

../redis-7.0.0/src (1)

Contents

- [|- acl.c](#)
- [|- adlist.c](#)
- [|- adlist.h](#)
- [|- ae.c](#)
- [|- ae.h](#)
- [|- ae_epoll.c](#)
- [|- ae_evport.c](#)
- [|- ae_kqueue.c](#)
- [|- ae_select.c](#)
- [|- anet.c](#)
- [|- anet.h](#)
- [|- aof.c](#)
- [|- asciilogo.h](#)
- [|- atomicvar.h](#)
- [|- bio.c](#)
- [|- bio.h](#)
- [|- bitops.c](#)
- [|- blocked.c](#)
- [|- call_reply.c](#)
- [|- call_reply.h](#)
- [|- childinfo.c](#)
- [|- cli_common.c](#)
- [|- cli_common.h](#)
- [|- cluster.c](#)
- [|- cluster.h](#)
- [/commands](#)
- [|- commands.c](#)
- [|- config.c](#)

[/acl.c](#)

[to top](#)

```

/*
 * Copyright (c) 2018, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "server.h"
#include "sha256.h"
#include <fcntl.h>
#include <ctype.h>

/*
=====
 * Global state for ACLs
 * =====*/

rax *Users; /* Table mapping usernames to user structures. */

user *DefaultUser; /* Global reference to the default user.
                    Every new connection is associated to it, if no
                    AUTH or HELLO is used to authenticate with a
                    different user. */

list *UsersToLoad; /* This is a list of users found in the configuration file
                    that we'll need to load in the final stage of Redis
                    initialization, after all the modules are already
                    loaded. Every list element is a NULL terminated

```

```

        array of SDS pointers: the first is the user name,
        all the remaining pointers are ACL rules in the same
        format as ACLSetUser(). */
list *ACLog;      /* Our security log, the user is able to inspect that
                   using the ACL LOG command .*/

static rax *commandId = NULL; /* Command name to id mapping */

static unsigned long nextid = 0; /* Next command id that has not been assigned
*/

struct ACLCategoryItem {
    const char *name;
    uint64_t flag;
} ACLCommandCategories[] = { /* See redis.conf for details on each category. */
    {"keyspace", ACL_CATEGORY_KEYSPACE},
    {"read", ACL_CATEGORY_READ},
    {"write", ACL_CATEGORY_WRITE},
    {"set", ACL_CATEGORY_SET},
    {"sortedset", ACL_CATEGORY_SORTEDSET},
    {"list", ACL_CATEGORY_LIST},
    {"hash", ACL_CATEGORY_HASH},
    {"string", ACL_CATEGORY_STRING},
    {"bitmap", ACL_CATEGORY_BITMAP},
    {"hyperloglog", ACL_CATEGORY_HYPERLOGLOG},
    {"geo", ACL_CATEGORY_GEO},
    {"stream", ACL_CATEGORY_STREAM},
    {"pubsub", ACL_CATEGORY_PUBSUB},
    {"admin", ACL_CATEGORY_ADMIN},
    {"fast", ACL_CATEGORY_FAST},
    {"slow", ACL_CATEGORY_SLOW},
    {"blocking", ACL_CATEGORY_BLOCKING},
    {"dangerous", ACL_CATEGORY_DANGEROUS},
    {"connection", ACL_CATEGORY_CONNECTION},
    {"transaction", ACL_CATEGORY_TRANSACTION},
    {"scripting", ACL_CATEGORY_SCRIPTING},
    {NULL,0} /* Terminator. */
};

struct ACLUserFlag {
    const char *name;
    uint64_t flag;
} ACLUserFlags[] = {
    /* Note: the order here dictates the emitted order at ACLDescribeUser */
    {"on", USER_FLAG_ENABLED},
    {"off", USER_FLAG_DISABLED},
    {"nopass", USER_FLAG_NOPASS},
    {"skip-sanitize-payload", USER_FLAG_SANITIZE_PAYLOAD_SKIP},
    {"sanitize-payload", USER_FLAG_SANITIZE_PAYLOAD},
    {NULL,0} /* Terminator. */
};

```

```

struct ACLSelectorFlags {
    const char *name;
    uint64_t flag;
} ACLSelectorFlags[] = {
    /* Note: the order here dictates the emitted order at ACLDescribeUser */
    {"allkeys", SELECTOR_FLAG_ALLKEYS},
    {"allchannels", SELECTOR_FLAG_ALLCHANNELS},
    {"allcommands", SELECTOR_FLAG_ALLCOMMANDS},
    {NULL,0} /* Terminator. */
};

/* ACL selectors are private and not exposed outside of acl.c. */
typedef struct {
    uint32_t flags; /* See SELECTOR_FLAG_* */
    /* The bit in allowed_commands is set if this user has the right to
     * execute this command.
     *
     * If the bit for a given command is NOT set and the command has
     * allowed first-args, Redis will also check allowed_firstargs in order to
     * understand if the command can be executed. */
    uint64_t allowed_commands[USER_COMMAND_BITS_COUNT/64];
    /* allowed_firstargs is used by ACL rules to block access to a command
    unless a
     * specific argv[1] is given.
     *
     * For each command ID (corresponding to the command bit set in
    allowed_commands),
     * This array points to an array of SDS strings, terminated by a NULL
    pointer,
     * with all the first-args that are allowed for this command. When no
    first-arg
     * matching is used, the field is just set to NULL to avoid allocating
     * USER_COMMAND_BITS_COUNT pointers. */
    sds **allowed_firstargs;
    list *patterns; /* A list of allowed key patterns. If this field is NULL
                     the user cannot mention any key in a command, unless
                     the flag ALLKEYS is set in the user. */
    list *channels; /* A list of allowed Pub/Sub channel patterns. If this
                     field is NULL the user cannot mention any channel in a
                     `PUBLISH` or [P][UNSUBSCRIBE] command, unless the flag
                     ALLCHANNELS is set in the user. */
} aclSelector;

void ACLResetFirstArgsForCommand(aclSelector *selector, unsigned long id);
void ACLResetFirstArgs(aclSelector *selector);
void ACLAddAllowedFirstArg(aclSelector *selector, unsigned long id, const char
*sub);
void ACLFreeLogEntry(void *le);
int ACLSetSelector(aclSelector *selector, const char *op, size_t olen);

```

```

/* The length of the string representation of a hashed password. */
#define HASH_PASSWORD_LEN SHA256_BLOCK_SIZE*2

/*
=====
* Helper functions for the rest of the ACL implementation
* =====*/

/* Return zero if strings are the same, non-zero if they are not.
* The comparison is performed in a way that prevents an attacker to obtain
* information about the nature of the strings just monitoring the execution
* time of the function.
*
* Note that limiting the comparison length to strings up to 512 bytes we
* can avoid leaking any information about the password length and any
* possible branch misprediction related leak.
*/
int time_independent_strcmp(char *a, char *b) {
    char bufa[CONFIG_AUTHPASS_MAX_LEN], bufb[CONFIG_AUTHPASS_MAX_LEN];
    /* The above two strlen perform len(a) + len(b) operations where either
     * a or b are fixed (our password) length, and the difference is only
     * relative to the length of the user provided string, so no information
     * leak is possible in the following two lines of code. */
    unsigned int alen = strlen(a);
    unsigned int blen = strlen(b);
    unsigned int j;
    int diff = 0;

    /* We can't compare strings longer than our static buffers.
     * Note that this will never pass the first test in practical circumstances
     * so there is no info leak. */
    if (alen > sizeof(bufa) || blen > sizeof(bufb)) return 1;

    memset(bufa,0,sizeof(bufa)); /* Constant time. */
    memset(bufb,0,sizeof(bufb)); /* Constant time. */
    /* Again the time of the following two copies is proportional to
     * len(a) + len(b) so no info is leaked. */
    memcpy(bufa,a,alen);
    memcpy(bufb,b,blen);

    /* Always compare all the chars in the two buffers without
     * conditional expressions. */
    for (j = 0; j < sizeof(bufa); j++) {
        diff |= (bufa[j] ^ bufb[j]);
    }
    /* Length must be equal as well. */
    diff |= alen ^ blen;
    return diff; /* If zero strings are the same. */
}

/* Given an SDS string, returns the SHA256 hex representation as a

```

```

    * new SDS string. */
sds ACLHashPassword(unsigned char *cleartext, size_t len) {
    SHA256_CTX ctx;
    unsigned char hash[SHA256_BLOCK_SIZE];
    char hex[HASH_PASSWORD_LEN];
    char *cset = "0123456789abcdef";

    sha256_init(&ctx);
    sha256_update(&ctx, (unsigned char *)cleartext, len);
    sha256_final(&ctx, hash);

    for (int j = 0; j < SHA256_BLOCK_SIZE; j++) {
        hex[j*2] = cset[((hash[j]&0xF0)>>4)];
        hex[j*2+1] = cset[(hash[j]&0xF)];
    }
    return sdsnewlen(hex, HASH_PASSWORD_LEN);
}

/* Given a hash and the hash length, returns C_OK if it is a valid password
 * hash, or C_ERR otherwise. */
int ACLCheckPasswordHash(unsigned char *hash, int hashlen) {
    if (hashlen != HASH_PASSWORD_LEN) {
        return C_ERR;
    }

    /* Password hashes can only be characters that represent
     * hexadecimal values, which are numbers and lowercase
     * characters 'a' through 'f'. */
    for(int i = 0; i < HASH_PASSWORD_LEN; i++) {
        char c = hash[i];
        if ((c < 'a' || c > 'f') && (c < '0' || c > '9')) {
            return C_ERR;
        }
    }
    return C_OK;
}

/*
=====
 * Low level ACL API
 * =====*/

/* Return 1 if the specified string contains spaces or null characters.
 * We do this for usernames and key patterns for simpler rewriting of
 * ACL rules, presentation on ACL list, and to avoid subtle security bugs
 * that may arise from parsing the rules in presence of escapes.
 * The function returns 0 if the string has no spaces. */
int ACLStringHasSpaces(const char *s, size_t len) {
    for (size_t i = 0; i < len; i++) {
        if (isspace(s[i]) || s[i] == 0) return 1;
    }
}

```

```

    return 0;
}

/* Given the category name the command returns the corresponding flag, or
 * zero if there is no match. */
uint64_t ACLGetCommandCategoryFlagByName(const char *name) {
    for (int j = 0; ACLCommandCategories[j].flag != 0; j++) {
        if (!strcasecmp(name, ACLCommandCategories[j].name)) {
            return ACLCommandCategories[j].flag;
        }
    }
    return 0; /* No match. */
}

/* Method for searching for a user within a list of user definitions. The
 * list contains an array of user arguments, and we are only
 * searching the first argument, the username, for a match. */
int ACLListMatchLoadedUser(void *definition, void *user) {
    sds *user_definition = definition;
    return sdscmp(user_definition[0], user) == 0;
}

/* Method for passwords/pattern comparison used for the user->passwords list
 * so that we can search for items with listSearchKey(). */
int ACLListMatchSds(void *a, void *b) {
    return sdscmp(a, b) == 0;
}

/* Method to free list elements from ACL users password/patterns lists. */
void ACLListFreeSds(void *item) {
    sdsfree(item);
}

/* Method to duplicate list elements from ACL users password/patterns lists. */
void *ACLListDupSds(void *item) {
    return sdsdup(item);
}

/* Structure used for handling key patterns with different key
 * based permissions. */
typedef struct {
    int flags; /* The CMD_KEYS_* flags for this key pattern */
    sds pattern; /* The pattern to match keys against */
} keyPattern;

/* Create a new key pattern. */
keyPattern *ACLKeyPatternCreate(sds pattern, int flags) {
    keyPattern *new = (keyPattern *) zmalloc(sizeof(keyPattern));
    new->pattern = pattern;
    new->flags = flags;
    return new;
}

```

```

}

/* Free a key pattern and internal structures. */
void ACLKeyPatternFree(keyPattern *pattern) {
    sdsfree(pattern->pattern);
    zfree(pattern);
}

/* Method for passwords/pattern comparison used for the user->passwords list
 * so that we can search for items with listSearchKey(). */
int ACLListMatchKeyPattern(void *a, void *b) {
    return sdscmp(((keyPattern *) a)->pattern, ((keyPattern *) b)->pattern) ==
0;
}

/* Method to free list elements from ACL users password/patterns lists. */
void ACLListFreeKeyPattern(void *item) {
    ACLKeyPatternFree(item);
}

/* Method to duplicate list elements from ACL users password/patterns lists. */
void *ACLListDupKeyPattern(void *item) {
    keyPattern *old = (keyPattern *) item;
    return ACLKeyPatternCreate(sdsdup(old->pattern), old->flags);
}

/* Append the string representation of a key pattern onto the
 * provided base string. */
sds sdsCatPatternString(sds base, keyPattern *pat) {
    if (pat->flags == ACL_ALL_PERMISSION) {
        base = sdscatlen(base, "~", 1);
    } else if (pat->flags == ACL_READ_PERMISSION) {
        base = sdscatlen(base, "%R~", 3);
    } else if (pat->flags == ACL_WRITE_PERMISSION) {
        base = sdscatlen(base, "%W~", 3);
    } else {
        serverPanic("Invalid key pattern flag detected");
    }
    return sdscatsds(base, pat->pattern);
}

/* Create an empty selector with the provided set of initial
 * flags. The selector will be default have no permissions. */
aclSelector *ACLCreateSelector(int flags) {
    aclSelector *selector = zmalloc(sizeof(aclSelector));
    selector->flags = flags | server.acl_pubsub_default;
    selector->patterns = listCreate();
    selector->channels = listCreate();
    selector->allowed_firstargs = NULL;

    listSetMatchMethod(selector->patterns, ACLListMatchKeyPattern);
}

```



```

listSetFreeMethod(selector->patterns,ACLListFreeKeyPattern);
listSetDupMethod(selector->patterns,ACLListDupKeyPattern);
listSetMatchMethod(selector->channels,ACLListMatchSds);
listSetFreeMethod(selector->channels,ACLListFreeSds);
listSetDupMethod(selector->channels,ACLListDupSds);
memset(selector->allowed_commands,0,sizeof(selector->allowed_commands));

return selector;
}

/* Cleanup the provided selector, including all interior structures. */
void ACLFreeSelector(aclSelector *selector) {
    listRelease(selector->patterns);
    listRelease(selector->channels);
    ACLResetFirstArgs(selector);
    zfree(selector);
}

/* Create an exact copy of the provided selector. */
aclSelector *ACLCopySelector(aclSelector *src) {
    aclSelector *dst = zmalloc(sizeof(aclSelector));
    dst->flags = src->flags;
    dst->patterns = listDup(src->patterns);
    dst->channels = listDup(src->channels);
    memcpy(dst->allowed_commands,src->allowed_commands,
        sizeof(dst->allowed_commands));
    dst->allowed_firstargs = NULL;
    /* Copy the allowed first-args array of array of SDS strings. */
    if (src->allowed_firstargs) {
        for (int j = 0; j < USER_COMMAND_BITS_COUNT; j++) {
            if (!(src->allowed_firstargs[j])) continue;
            for (int i = 0; src->allowed_firstargs[j][i]; i++) {
                ACLAddAllowedFirstArg(dst, j, src->allowed_firstargs[j][i]);
            }
        }
    }
    return dst;
}

/* List method for freeing a selector */
void ACLListFreeSelector(void *a) {
    ACLFreeSelector((aclSelector *) a);
}

/* List method for duplicating a selector */
void *ACLListDuplicateSelector(void *src) {
    return ACLCopySelector((aclSelector *)src);
}

/* All users have an implicit root selector which
 * provides backwards compatibility to the old ACLs-

```

```

    * permissions. */
aclSelector *ACLUserGetRootSelector(user *u) {
    serverAssert(listLength(u->selectors));
    aclSelector *s = (aclSelector *) listNodeValue(listFirst(u->selectors));
    serverAssert(s->flags & SELECTOR_FLAG_ROOT);
    return s;
}

/* Create a new user with the specified name, store it in the list
 * of users (the Users global radix tree), and returns a reference to
 * the structure representing the user.
 *
 * If the user with such name already exists NULL is returned. */
user *ACLCreateUser(const char *name, size_t namelen) {
    if (raxFind(Users, (unsigned char *)name, namelen) != raxNotFound) return
    NULL;
    user *u = zmalloc(sizeof(*u));
    u->name = sdsnewlen(name, namelen);
    u->flags = USER_FLAG_DISABLED;
    u->passwords = listCreate();
    listSetMatchMethod(u->passwords, ACLListMatchSds);
    listSetFreeMethod(u->passwords, ACLListFreeSds);
    listSetDupMethod(u->passwords, ACLListDupSds);

    u->selectors = listCreate();
    listSetFreeMethod(u->selectors, ACLListFreeSelector);
    listSetDupMethod(u->selectors, ACLListDuplicateSelector);

    /* Add the initial root selector */
    aclSelector *s = ACLCreateSelector(SELECTOR_FLAG_ROOT);
    listAddNodeHead(u->selectors, s);

    raxInsert(Users, (unsigned char *)name, namelen, u, NULL);
    return u;
}

/* This function should be called when we need an unlinked "fake" user
 * we can use in order to validate ACL rules or for other similar reasons.
 * The user will not get linked to the Users radix tree. The returned
 * user should be released with ACLFreeUser() as usually. */
user *ACLCreateUnlinkedUser(void) {
    char username[64];
    for (int j = 0; j < 64; j++) {
        snprintf(username, sizeof(username), "__fakeuser:%d__", j);
        user *fakeuser = ACLCreateUser(username, strlen(username));
        if (fakeuser == NULL) continue;
        int retval = raxRemove(Users, (unsigned char *)username,
                               strlen(username), NULL);
        serverAssert(retval != 0);
        return fakeuser;
    }
}

```

```

}

/* Release the memory used by the user structure. Note that this function
 * will not remove the user from the Users global radix tree. */
void ACLFreeUser(user *u) {
    sdsfree(u->name);
    listRelease(u->passwords);
    listRelease(u->selectors);
    zfree(u);
}

/* When a user is deleted we need to cycle the active
 * connections in order to kill all the pending ones that
 * are authenticated with such user. */
void ACLFreeUserAndKillClients(user *u) {
    listIter li;
    listNode *ln;
    listRewind(server.clients,&li);
    while ((ln = listNext(&li)) != NULL) {
        client *c = listNodeValue(ln);
        if (c->user == u) {
            /* We'll free the connection asynchronously, so
             * in theory to set a different user is not needed.
             * However if there are bugs in Redis, soon or later
             * this may result in some security hole: it's much
             * more defensive to set the default user and put
             * it in non authenticated mode. */
            c->user = DefaultUser;
            c->authenticated = 0;
            /* We will write replies to this client later, so we can't
             * close it directly even if async. */
            if (c == server.current_client) {
                c->flags |= CLIENT_CLOSE_AFTER_COMMAND;
            } else {
                freeClientAsync(c);
            }
        }
    }
    ACLFreeUser(u);
}

/* Copy the user ACL rules from the source user 'src' to the destination
 * user 'dst' so that at the end of the process they'll have exactly the
 * same rules (but the names will continue to be the original ones). */
void ACLCopyUser(user *dst, user *src) {
    listRelease(dst->passwords);
    listRelease(dst->selectors);
    dst->passwords = listDup(src->passwords);
    dst->selectors = listDup(src->selectors);
    dst->flags = src->flags;
}

```

```

/* Free all the users registered in the radix tree 'users' and free the
 * radix tree itself. */
void ACLFreeUsersSet(rax *users) {
    raxFreeWithCallback(users, (void*)(void*))ACLFreeUserAndKillClients);
}

/* Given a command ID, this function set by reference 'word' and 'bit'
 * so that user->allowed_commands[word] will address the right word
 * where the corresponding bit for the provided ID is stored, and
 * so that user->allowed_commands[word]&bit will identify that specific
 * bit. The function returns C_ERR in case the specified ID overflows
 * the bitmap in the user representation. */
int ACLGetCommandBitCoordinates(uint64_t id, uint64_t *word, uint64_t *bit) {
    if (id >= USER_COMMAND_BITS_COUNT) return C_ERR;
    *word = id / sizeof(uint64_t) / 8;
    *bit = 1ULL << (id % (sizeof(uint64_t) * 8));
    return C_OK;
}

/* Check if the specified command bit is set for the specified user.
 * The function returns 1 if the bit is set or 0 if it is not.
 * Note that this function does not check the ALLCOMMANDS flag of the user
 * but just the lowlevel bitmask.
 *
 * If the bit overflows the user internal representation, zero is returned
 * in order to disallow the execution of the command in such edge case. */
int ACLGetSelectorCommandBit(const aclSelector *selector, unsigned long id) {
    uint64_t word, bit;
    if (ACLGetCommandBitCoordinates(id, &word, &bit) == C_ERR) return 0;
    return (selector->allowed_commands[word] & bit) != 0;
}

/* When +@all or allcommands is given, we set a reserved bit as well that we
 * can later test, to see if the user has the right to execute "future
 * commands",
 * that is, commands loaded later via modules. */
int ACLSelectorCanExecuteFutureCommands(aclSelector *selector) {
    return ACLGetSelectorCommandBit(selector, USER_COMMAND_BITS_COUNT-1);
}

/* Set the specified command bit for the specified user to 'value' (0 or 1).
 * If the bit overflows the user internal representation, no operation
 * is performed. As a side effect of calling this function with a value of
 * zero, the user flag ALLCOMMANDS is cleared since it is no longer possible
 * to skip the command bit explicit test. */
void ACLSetSelectorCommandBit(aclSelector *selector, unsigned long id, int
value) {
    uint64_t word, bit;
    if (ACLGetCommandBitCoordinates(id, &word, &bit) == C_ERR) return;
    if (value) {

```

```

        selector->allowed_commands[word] |= bit;
    } else {
        selector->allowed_commands[word] &= ~bit;
        selector->flags &= ~SELECTOR_FLAG_ALLCOMMANDS;
    }
}

/* This function is used to allow/block a specific command.
 * Allowing/blocking a container command also applies for its subcommands */
void ACLChangeSelectorPerm(aclSelector *selector, struct redisCommand *cmd, int
allow) {
    unsigned long id = cmd->id;
    ACLSetSelectorCommandBit(selector,id,allow);
    ACLResetFirstArgsForCommand(selector,id);
    if (cmd->subcommands_dict) {
        dictEntry *de;
        dictIterator *di = dictGetSafeIterator(cmd->subcommands_dict);
        while((de = dictNext(di)) != NULL) {
            struct redisCommand *sub = (struct redisCommand *)dictGetVal(de);
            ACLSetSelectorCommandBit(selector,sub->id,allow);
        }
        dictReleaseIterator(di);
    }
}

void ACLSetSelectorCommandBitsForCategoryLogic(dict *commands, aclSelector
*selector, uint64_t cflag, int value) {
    dictIterator *di = dictGetIterator(commands);
    dictEntry *de;
    while ((de = dictNext(di)) != NULL) {
        struct redisCommand *cmd = dictGetVal(de);
        if (cmd->flags & CMD_MODULE) continue; /* Ignore modules commands. */
        if (cmd->acl_categories & cflag) {
            ACLChangeSelectorPerm(selector,cmd,value);
        }
        if (cmd->subcommands_dict) {
            ACLSetSelectorCommandBitsForCategoryLogic(cmd->subcommands_dict,
selector, cflag, value);
        }
    }
    dictReleaseIterator(di);
}

/* This is like ACLSetSelectorCommandBit(), but instead of setting the
specified
 * ID, it will check all the commands in the category specified as argument,
 * and will set all the bits corresponding to such commands to the specified
 * value. Since the category passed by the user may be non existing, the
 * function returns C_ERR if the category was not found, or C_OK if it was
 * found and the operation was performed. */
int ACLSetSelectorCommandBitsForCategory(aclSelector *selector, const char

```

```

*category, int value) {
    uint64_t cflag = ACLGetCommandCategoryFlagByName(category);
    if (!cflag) return C_ERR;
    ACLSetSelectorCommandBitsForCategoryLogic(server.orig_commands, selector,
cflag, value);
    return C_OK;
}

```

```

void ACLCountCategoryBitsForCommands(dict *commands, aclSelector *selector,
unsigned long *on, unsigned long *off, uint64_t cflag) {
    dictIterator *di = dictGetIterator(commands);
    dictEntry *de;
    while ((de = dictNext(di)) != NULL) {
        struct redisCommand *cmd = dictGetVal(de);
        if (cmd->acl_categories & cflag) {
            if (ACLGetSelectorCommandBit(selector, cmd->id))
                (*on)++;
            else
                (*off)++;
        }
        if (cmd->subcommands_dict) {
            ACLCountCategoryBitsForCommands(cmd->subcommands_dict, selector,
on, off, cflag);
        }
    }
    dictReleaseIterator(di);
}

```

```

/* Return the number of commands allowed (on) and denied (off) for the user 'u'
 * in the subset of commands flagged with the specified category name.
 * If the category name is not valid, C_ERR is returned, otherwise C_OK is
 * returned and on and off are populated by reference. */

```

```

int ACLCountCategoryBitsForSelector(aclSelector *selector, unsigned long *on,
unsigned long *off,
                                const char *category)
{
    uint64_t cflag = ACLGetCommandCategoryFlagByName(category);
    if (!cflag) return C_ERR;

    *on = *off = 0;
    ACLCountCategoryBitsForCommands(server.orig_commands, selector, on, off,
cflag);
    return C_OK;
}

```

```

sds ACLDescribeSelectorCommandRulesSingleCommands(aclSelector *selector,
aclSelector *fake_selector,
            sds rules, dict *commands) {
    dictIterator *di = dictGetIterator(commands);
    dictEntry *de;
    while ((de = dictNext(di)) != NULL) {

```

```

    struct redisCommand *cmd = dictGetVal(de);
    int userbit = ACLGetSelectorCommandBit(selector,cmd->id);
    int fakebit = ACLGetSelectorCommandBit(fake_selector,cmd->id);
    if (userbit != fakebit) {
        rules = sdscatlen(rules, userbit ? "+" : "-", 1);
        rules = sdscatsds(rules,cmd->fullname);
        rules = sdscatlen(rules," ",1);
        ACLChangeSelectorPerm(fake_selector,cmd,userbit);
    }

    if (cmd->subcommands_dict)
        rules =
ACLDescribeSelectorCommandRulesSingleCommands(selector,fake_selector,rules,cmd->subcommands_dict);

    /* Emit the first-args if there are any. */
    if (userbit == 0 && selector->allowed_firstargs &&
        selector->allowed_firstargs[cmd->id])
    {
        for (int j = 0; selector->allowed_firstargs[cmd->id][j]; j++) {
            rules = sdscatlen(rules,"+",1);
            rules = sdscatsds(rules,cmd->fullname);
            rules = sdscatlen(rules,"|",1);
            rules = sdscatsds(rules,selector->allowed_firstargs[cmd->id]
[j]);
            rules = sdscatlen(rules," ",1);
        }
    }
    dictReleaseIterator(di);
    return rules;
}

/* This function returns an SDS string representing the specified selector ACL
 * rules related to command execution, in the same format you could set them
 * back using ACL SETUSER. The function will return just the set of rules
 * needed
 * to recreate the user commands bitmap, without including other user flags
 * such
 * as on/off, passwords and so forth. The returned string always starts with
 * the +@all or -@all rule, depending on the user bitmap, and is followed, if
 * needed, by the other rules needed to narrow or extend what the user can do.
 */
sds ACLDescribeSelectorCommandRules(aclSelector *selector) {
    sds rules = sdsempty();
    int additive; /* If true we start from -@all and add, otherwise if
                   false we start from +@all and remove. */

    /* This code is based on a trick: as we generate the rules, we apply
     * them to a fake user, so that as we go we still know what are the
     * bit differences we should try to address by emitting more rules. */

```

```

aclSelector fs = {0};
aclSelector *fake_selector = &fs;

/* Here we want to understand if we should start with +@all and remove
 * the commands corresponding to the bits that are not set in the user
 * commands bitmap, or the contrary. Note that semantically the two are
 * different. For instance starting with +@all and subtracting, the user
 * will be able to execute future commands, while -@all and adding will
just
 * allow the user the run the selected commands and/or categories.
 * How do we test for that? We use the trick of a reserved command ID bit
 * that is set only by +@all (and its alias "allcommands"). */
if (ACLSelectorCanExecuteFutureCommands(selector)) {
    additive = 0;
    rules = sdscat(rules, "+@all ");
    ACLSetSelector(fake_selector, "+@all", -1);
} else {
    additive = 1;
    rules = sdscat(rules, "-@all ");
    ACLSetSelector(fake_selector, "-@all", -1);
}

/* Attempt to find a good approximation for categories and commands
 * based on the current bits used, by looping over the category list
 * and applying the best fit each time. Often a set of categories will not
 * perfectly match the set of commands into it, so at the end we do a
 * final pass adding/removing the single commands needed to make the bitmap
 * exactly match. A temp user is maintained to keep track of categories
 * already applied. */
aclSelector ts = {0};
aclSelector *temp_selector = &ts;

/* Keep track of the categories that have been applied, to prevent
 * applying them twice. */
char applied[sizeof(ACLCommandCategories)/sizeof(ACLCommandCategories[0])];
memset(applied, 0, sizeof(applied));

memcpy(temp_selector->allowed_commands,
        selector->allowed_commands,
        sizeof(selector->allowed_commands));
while (1) {
    int best = -1;
    unsigned long mindiff = INT_MAX, maxsame = 0;
    for (int j = 0; ACLCommandCategories[j].flag != 0; j++) {
        if (applied[j]) continue;

        unsigned long on, off, diff, same;

        ACLCountCategoryBitsForSelector(temp_selector, &on, &off, ACLCommandCategories[j].nam
        /* Check if the current category is the best this loop:
         * * It has more commands in common with the user than commands

```



```

        *   that are different.
        * AND EITHER
        * * It has the fewest number of differences
        *   than the best match we have found so far.
        * * OR it matches the fewest number of differences
        *   that we've seen but it has more in common. */
diff = additive ? off : on;
same = additive ? on : off;
if (same > diff &&
    ((diff < mindiff) || (diff == mindiff && same > maxsame)))
{
    best = j;
    mindiff = diff;
    maxsame = same;
}
}

/* We didn't find a match */
if (best == -1) break;

sds op = sdsnewlen(additive ? "+@" : "-@", 2);
op = sdscat(op, ACLCommandCategories[best].name);
ACLSetSelector(fake_selector, op, -1);

sds invop = sdsnewlen(additive ? "-@" : "+@", 2);
invop = sdscat(invop, ACLCommandCategories[best].name);
ACLSetSelector(temp_selector, invop, -1);

rules = sdscatsds(rules, op);
rules = sdscatlen(rules, " ", 1);
sdsfree(op);
sdsfree(invop);

applied[best] = 1;
}

/* Fix the final ACLs with single commands differences. */
rules =
ACLDDescribeSelectorCommandRulesSingleCommands(selector, fake_selector, rules, server.

/* Trim the final useless space. */
sdsrange(rules, 0, -2);

/* This is technically not needed, but we want to verify that now the
 * predicted bitmap is exactly the same as the user bitmap, and abort
 * otherwise, because aborting is better than a security risk in this
 * code path. */
if (memcmp(fake_selector->allowed_commands,
            selector->allowed_commands,
            sizeof(selector->allowed_commands)) != 0)
{

```

```

        serverLog(LL_WARNING,
            "CRITICAL ERROR: User ACLs don't match final bitmap: '%s'",
            rules);
        serverPanic("No bitmap match in ACLDescribeSelectorCommandRules()");
    }
    return rules;
}

```

```

sds ACLDescribeSelector(aclSelector *selector) {
    listIter li;
    listNode *ln;
    sds res = sdsempty();
    /* Key patterns. */
    if (selector->flags & SELECTOR_FLAG_ALLKEYS) {
        res = sdscatlen(res, "~* ", 3);
    } else {
        listRewind(selector->patterns, &li);
        while((ln = listNext(&li))) {
            keyPattern *thispat = (keyPattern *)listNodeValue(ln);
            res = sdsCatPatternString(res, thispat);
            res = sdscatlen(res, " ", 1);
        }
    }
}

```

```

/* Pub/sub channel patterns. */
if (selector->flags & SELECTOR_FLAG_ALLCHANNELS) {
    res = sdscatlen(res, "&* ", 3);
} else {
    res = sdscatlen(res, "resetchannels ", 14);
    listRewind(selector->channels, &li);
    while((ln = listNext(&li))) {
        sds thispat = listNodeValue(ln);
        res = sdscatlen(res, "&", 1);
        res = sdscatsds(res, thispat);
        res = sdscatlen(res, " ", 1);
    }
}
}

```

```

/* Command rules. */
sds rules = ACLDescribeSelectorCommandRules(selector);
res = sdscatsds(res, rules);
sdsfree(rules);
return res;
}

```

```

/* This is similar to ACLDescribeSelectorCommandRules(), however instead of
 * describing just the user command rules, everything is described: user
 * flags, keys, passwords and finally the command rules obtained via
 * the ACLDescribeSelectorCommandRules() function. This is the function we call
 * when we want to rewrite the configuration files describing ACLs and
 * in order to show users with ACL LIST. */

```

```

sds ACLDescribeUser(user *u) {
    sds res = sdsempty();

    /* Flags. */
    for (int j = 0; ACLUserFlags[j].flag; j++) {
        if (u->flags & ACLUserFlags[j].flag) {
            res = sdscat(res, ACLUserFlags[j].name);
            res = sdscatlen(res, " ", 1);
        }
    }

    /* Passwords. */
    listIter li;
    listNode *ln;
    listRewind(u->passwords, &li);
    while((ln = listNext(&li))) {
        sds thispass = listNodeValue(ln);
        res = sdscatlen(res, "#", 1);
        res = sdscatsds(res, thispass);
        res = sdscatlen(res, " ", 1);
    }

    /* Selectors (Commands and keys) */
    listRewind(u->selectors, &li);
    while((ln = listNext(&li))) {
        aclSelector *selector = (aclSelector *) listNodeValue(ln);
        sds default_perm = ACLDescribeSelector(selector);
        if (selector->flags & SELECTOR_FLAG_ROOT) {
            res = sdscatfmt(res, "%s", default_perm);
        } else {
            res = sdscatfmt(res, " (%s)", default_perm);
        }
        sdsfree(default_perm);
    }
    return res;
}

/* Get a command from the original command table, that is not affected
 * by the command renaming operations: we base all the ACL work from that
 * table, so that ACLs are valid regardless of command renaming. */
struct redisCommand *ACLLookupCommand(const char *name) {
    struct redisCommand *cmd;
    sds sdsname = sdsnew(name);
    cmd = lookupCommandBySdsLogic(server.orig_commands, sdsname);
    sdsfree(sdsname);
    return cmd;
}

/* Flush the array of allowed first-args for the specified user
 * and command ID. */
void ACLResetFirstArgsForCommand(aclSelector *selector, unsigned long id) {

```

```

    if (selector->allowed_firstargs && selector->allowed_firstargs[id]) {
        for (int i = 0; selector->allowed_firstargs[id][i]; i++)
            sdsfree(selector->allowed_firstargs[id][i]);
        zfree(selector->allowed_firstargs[id]);
        selector->allowed_firstargs[id] = NULL;
    }
}

/* Flush the entire table of first-args. This is useful on +@all, -@all
 * or similar to return back to the minimal memory usage (and checks to do)
 * for the user. */
void ACLResetFirstArgs(acSelector *selector) {
    if (selector->allowed_firstargs == NULL) return;
    for (int j = 0; j < USER_COMMAND_BITS_COUNT; j++) {
        if (selector->allowed_firstargs[j]) {
            for (int i = 0; selector->allowed_firstargs[j][i]; i++)
                sdsfree(selector->allowed_firstargs[j][i]);
            zfree(selector->allowed_firstargs[j]);
        }
    }
    zfree(selector->allowed_firstargs);
    selector->allowed_firstargs = NULL;
}

/* Add a first-arg to the list of subcommands for the user 'u' and
 * the command id specified. */
void ACLAddAllowedFirstArg(acSelector *selector, unsigned long id, const char
*sub) {
    /* If this is the first first-arg to be configured for
     * this user, we have to allocate the first-args array. */
    if (selector->allowed_firstargs == NULL) {
        selector->allowed_firstargs = zcalloc(USER_COMMAND_BITS_COUNT *
sizeof(sds*));
    }

    /* We also need to enlarge the allocation pointing to the
     * null terminated SDS array, to make space for this one.
     * To start check the current size, and while we are here
     * make sure the first-arg is not already specified inside. */
    long items = 0;
    if (selector->allowed_firstargs[id]) {
        while(selector->allowed_firstargs[id][items]) {
            /* If it's already here do not add it again. */
            if (!strcasecmp(selector->allowed_firstargs[id][items],sub))
                return;
            items++;
        }
    }

    /* Now we can make space for the new item (and the null term). */
    items += 2;

```

```

    selector->allowed_firstargs[id] = zrealloc(selector->allowed_firstargs[id],
sizeof(sds)*items);
    selector->allowed_firstargs[id][items-2] = sdsnew(sub);
    selector->allowed_firstargs[id][items-1] = NULL;
}

/* Create an ACL selector from the given ACL operations, which should be
 * a list of space separate ACL operations that starts and ends
 * with parentheses.
 *
 * If any of the operations are invalid, NULL will be returned instead
 * and errno will be set corresponding to the interior error. */
aclSelector *aclCreateSelectorFromOpSet(const char *opset, size_t opsetlen) {
    serverAssert(opset[0] == '(' && opset[opsetlen - 1] == ');
    aclSelector *s = ACLCreateSelector(0);

    int argc = 0;
    sds trimmed = sdsnewlen(opset + 1, opsetlen - 2);
    sds *argv = sdssplitargs(trimmed, &argc);
    for (int i = 0; i < argc; i++) {
        if (ACLSetSelector(s, argv[i], sdslen(argv[i])) == C_ERR) {
            ACLFreeSelector(s);
            s = NULL;
            goto cleanup;
        }
    }

cleanup:
    sdsfreesplitres(argv, argc);
    sdsfree(trimmed);
    return s;
}

/* Set a selector's properties with the provided 'op'.
 *
 * +<command>    Allow the execution of that command.
 *               May be used with `|` for allowing subcommands (e.g
"+config|get")
 * -<command>    Disallow the execution of that command.
 *               May be used with `|` for blocking subcommands (e.g "-
config|set")
 * +@<category> Allow the execution of all the commands in such category
 *               with valid categories are like @admin, @set, @sortedset, ...
 *               and so forth, see the full list in the server.c file where
 *               the Redis command table is described and defined.
 *               The special category @all means all the commands, but currently
 *               present in the server, and that will be loaded in the future
 *               via modules.
 * +<command>|first-arg    Allow a specific first argument of an otherwise
 *                           disabled command. Note that this form is not
 *                           allowed as negative like -SELECT|1, but

```

```

*                only additive starting with "+".
* allcommands    Alias for +@all. Note that it implies the ability to execute
*                all the future commands loaded via the modules system.
* nocommands     Alias for -@all.
* ~<pattern>     Add a pattern of keys that can be mentioned as part of
*                commands. For instance ~* allows all the keys. The pattern
*                is a glob-style pattern like the one of KEYS.
*                It is possible to specify multiple patterns.
* %R~<pattern>   Add key read pattern that specifies which keys can be read
*                from.
* %W~<pattern>   Add key write pattern that specifies which keys can be
*                written to.
* allkeys        Alias for ~*
* resetkeys      Flush the list of allowed keys patterns.
* &<pattern>     Add a pattern of channels that can be mentioned as part of
*                Pub/Sub commands. For instance &* allows all the channels. The
*                pattern is a glob-style pattern like the one of PSUBSCRIBE.
*                It is possible to specify multiple patterns.
* allchannels    Alias for &*
* resetchannels  Flush the list of allowed channel patterns.
*/
int ACLSetSelector(aclSelector *selector, const char* op, size_t olen) {
    if (!strcasecmp(op,"allkeys") ||
        !strcasecmp(op,"~*"))
    {
        selector->flags |= SELECTOR_FLAG_ALLKEYS;
        listEmpty(selector->patterns);
    } else if (!strcasecmp(op,"resetkeys")) {
        selector->flags &= ~SELECTOR_FLAG_ALLKEYS;
        listEmpty(selector->patterns);
    } else if (!strcasecmp(op,"allchannels") ||
        !strcasecmp(op,"&*))
    {
        selector->flags |= SELECTOR_FLAG_ALLCHANNELS;
        listEmpty(selector->channels);
    } else if (!strcasecmp(op,"resetchannels")) {
        selector->flags &= ~SELECTOR_FLAG_ALLCHANNELS;
        listEmpty(selector->channels);
    } else if (!strcasecmp(op,"allcommands") ||
        !strcasecmp(op,"+@all"))
    {
        memset(selector->allowed_commands,255,sizeof(selector->allowed_commands));
        selector->flags |= SELECTOR_FLAG_ALLCOMMANDS;
        ACLResetFirstArgs(selector);
    } else if (!strcasecmp(op,"nocommands") ||
        !strcasecmp(op,"-@all"))
    {
        memset(selector->allowed_commands,0,sizeof(selector->allowed_commands));
        selector->flags &= ~SELECTOR_FLAG_ALLCOMMANDS;
    }
}

```

```

        ACLResetFirstArgs(selector);
    } else if (op[0] == '~' || op[0] == '%') {
        if (selector->flags & SELECTOR_FLAG_ALLKEYS) {
            errno = EEXIST;
            return C_ERR;
        }
        int flags = 0;
        size_t offset = 1;
        if (op[0] == '%') {
            for (; offset < olen; offset++) {
                if (toupper(op[offset]) == 'R' && !(flags &
ACL_READ_PERMISSION)) {
                    flags |= ACL_READ_PERMISSION;
                } else if (toupper(op[offset]) == 'W' && !(flags &
ACL_WRITE_PERMISSION)) {
                    flags |= ACL_WRITE_PERMISSION;
                } else if (op[offset] == '~') {
                    offset++;
                    break;
                } else {
                    errno = EINVAL;
                    return C_ERR;
                }
            }
        } else {
            flags = ACL_ALL_PERMISSION;
        }

        if (ACLStringHasSpaces(op+offset, olen-offset)) {
            errno = EINVAL;
            return C_ERR;
        }

        keyPattern *newpat = ACLKeyPatternCreate(sdsnewlen(op+offset, olen-
offset), flags);
        listNode *ln = listSearchKey(selector->patterns, newpat);
        /* Avoid re-adding the same key pattern multiple times. */
        if (ln == NULL) {
            listAddNodeTail(selector->patterns, newpat);
        } else {
            ((keyPattern *)listNodeValue(ln))->flags |= flags;
            ACLKeyPatternFree(newpat);
        }
        selector->flags &= ~SELECTOR_FLAG_ALLKEYS;
    } else if (op[0] == '&') {
        if (selector->flags & SELECTOR_FLAG_ALLCHANNELS) {
            errno = EISDIR;
            return C_ERR;
        }
        if (ACLStringHasSpaces(op+1, olen-1)) {
            errno = EINVAL;
            return C_ERR;
        }
    }

```

```

}
sds newpat = sdsnewlen(op+1,oplen-1);
listNode *ln = listSearchKey(selector->channels,newpat);
/* Avoid re-adding the same channel pattern multiple times. */
if (ln == NULL)
    listAddNodeTail(selector->channels,newpat);
else
    sdsfree(newpat);
selector->flags &= ~SELECTOR_FLAG_ALLCHANNELS;
} else if (op[0] == '+' && op[1] != '@') {
    if (strchr(op,'|') == NULL) {
        struct redisCommand *cmd = ACLLookupCommand(op+1);
        if (cmd == NULL) {
            errno = ENOENT;
            return C_ERR;
        }
        ACLChangeSelectorPerm(selector,cmd,1);
    } else {
        /* Split the command and subcommand parts. */
        char *copy = zstrdup(op+1);
        char *sub = strchr(copy,'|');
        sub[0] = '\0';
        sub++;

        struct redisCommand *cmd = ACLLookupCommand(copy);

        /* Check if the command exists. We can't check the
         * first-arg to see if it is valid. */
        if (cmd == NULL) {
            zfree(copy);
            errno = ENOENT;
            return C_ERR;
        }

        /* We do not support allowing first-arg of a subcommand */
        if (cmd->parent) {
            zfree(copy);
            errno = ECHILD;
            return C_ERR;
        }

        /* The subcommand cannot be empty, so things like DEBUG|
         * are syntax errors of course. */
        if (strlen(sub) == 0) {
            zfree(copy);
            errno = EINVAL;
            return C_ERR;
        }

        if (cmd->subcommands_dict) {
            /* If user is trying to allow a valid subcommand we can just

```



```

add its unique ID */
    cmd = ACLLookupCommand(op+1);
    if (cmd == NULL) {
        zfree(copy);
        errno = ENOENT;
        return C_ERR;
    }
    ACLChangeSelectorPerm(selector,cmd,1);
} else {
    /* If user is trying to use the ACL mech to block SELECT except
SELECT 0 or
    * block DEBUG except DEBUG OBJECT (DEBUG subcommands are not
considered
    * subcommands for now) we use the allowed_firstargs mechanism.
*/

    /* Add the first-arg to the list of valid ones. */
    serverLog(LL_WARNING, "Deprecation warning: Allowing a first
arg of an otherwise "
                                "blocked command is a misuse of ACL and
may get disabled "
                                "in the future (offender: +%s)", op+1);
    ACLAddAllowedFirstArg(selector,cmd->id,sub);
}

    zfree(copy);
}
} else if (op[0] == '-' && op[1] != '@') {
    struct redisCommand *cmd = ACLLookupCommand(op+1);
    if (cmd == NULL) {
        errno = ENOENT;
        return C_ERR;
    }
    ACLChangeSelectorPerm(selector,cmd,0);
} else if ((op[0] == '+' || op[0] == '-') && op[1] == '@') {
    int bitval = op[0] == '+' ? 1 : 0;
    if (ACLSetSelectorCommandBitsForCategory(selector,op+2,bitval) ==
C_ERR) {
        errno = ENOENT;
        return C_ERR;
    }
} else {
    errno = EINVAL;
    return C_ERR;
}
return C_OK;
}

/* Set user properties according to the string "op". The following
* is a description of what different strings will do:
*

```

```

* on          Enable the user: it is possible to authenticate as this user.
* off         Disable the user: it's no longer possible to authenticate
*             with this user, however the already authenticated connections
*             will still work.
* ><password> Add this password to the list of valid password for the user.
*             For example >mypass will add "mypass" to the list.
*             This directive clears the "nopass" flag (see later).
* #<hash>     Add this password hash to the list of valid hashes for
*             the user. This is useful if you have previously computed
*             the hash, and don't want to store it in plaintext.
*             This directive clears the "nopass" flag (see later).
* <<password> Remove this password from the list of valid passwords.
* !<hash>     Remove this hashed password from the list of valid passwords.
*             This is useful when you want to remove a password just by
*             hash without knowing its plaintext version at all.
* nopass      All the set passwords of the user are removed, and the user
*             is flagged as requiring no password: it means that every
*             password will work against this user. If this directive is
*             used for the default user, every new connection will be
*             immediately authenticated with the default user without
*             any explicit AUTH command required. Note that the "resetpass"
*             directive will clear this condition.
* resetpass   Flush the list of allowed passwords. Moreover removes the
*             "nopass" status. After "resetpass" the user has no associated
*             passwords and there is no way to authenticate without adding
*             some password (or setting it as "nopass" later).
* reset       Performs the following actions: resetpass, resetkeys, off,
*             -@all. The user returns to the same state it has immediately
*             after its creation.
* (<options>) Create a new selector with the options specified within the
*             parentheses and attach it to the user. Each option should be
*             space separated. The first character must be ( and the last
*             character must be ).
* clearselectors Remove all of the currently attached selectors.
*             Note this does not change the "root" user
permissions,
*             which are the permissions directly applied onto the
*             user (outside the parentheses).
*
* Selector options can also be specified by this function, in which case
* they update the root selector for the user.
*
* The 'op' string must be null terminated. The 'oplen' argument should
* specify the length of the 'op' string in case the caller requires to pass
* binary data (for instance the >password form may use a binary password).
* Otherwise the field can be set to -1 and the function will use strlen()
* to determine the length.
*
* The function returns C_OK if the action to perform was understood because
* the 'op' string made sense. Otherwise C_ERR is returned if the operation
* is unknown or has some syntax error.

```

```

*
* When an error is returned, errno is set to the following values:
*
* EINVAL: The specified opcode is not understood or the key/channel pattern is
*         invalid (contains non allowed characters).
* ENOENT: The command name or command category provided with + or - is not
*         known.
* EEXIST: You are adding a key pattern after "*" was already added. This is
*         almost surely an error on the user side.
* EISDIR: You are adding a channel pattern after "*" was already added. This
is
*         almost surely an error on the user side.
* ENODEV: The password you are trying to remove from the user does not exist.
* EBADMSG: The hash you are trying to add is not a valid hash.
* ECHILD: Attempt to allow a specific first argument of a subcommand
*/
int ACLSetUser(user *u, const char *op, ssize_t oplen) {
    if (oplen == -1) oplen = strlen(op);
    if (oplen == 0) return C_OK; /* Empty string is a no-operation. */
    if (!strcasecmp(op,"on")) {
        u->flags |= USER_FLAG_ENABLED;
        u->flags &= ~USER_FLAG_DISABLED;
    } else if (!strcasecmp(op,"off")) {
        u->flags |= USER_FLAG_DISABLED;
        u->flags &= ~USER_FLAG_ENABLED;
    } else if (!strcasecmp(op,"skip-sanitize-payload")) {
        u->flags |= USER_FLAG_SANITIZE_PAYLOAD_SKIP;
        u->flags &= ~USER_FLAG_SANITIZE_PAYLOAD;
    } else if (!strcasecmp(op,"sanitize-payload")) {
        u->flags &= ~USER_FLAG_SANITIZE_PAYLOAD_SKIP;
        u->flags |= USER_FLAG_SANITIZE_PAYLOAD;
    } else if (!strcasecmp(op,"nopass")) {
        u->flags |= USER_FLAG_NOPASS;
        listEmpty(u->passwords);
    } else if (!strcasecmp(op,"resetpass")) {
        u->flags &= ~USER_FLAG_NOPASS;
        listEmpty(u->passwords);
    } else if (op[0] == '>' || op[0] == '#') {
        sds newpass;
        if (op[0] == '>') {
            newpass = ACLHashPassword((unsigned char*)op+1,oplen-1);
        } else {
            if (ACLCheckPasswordHash((unsigned char*)op+1,oplen-1) == C_ERR) {
                errno = EBADMSG;
                return C_ERR;
            }
            newpass = sdsnewlen(op+1,oplen-1);
        }

        listNode *ln = listSearchKey(u->passwords,newpass);
        /* Avoid re-adding the same password multiple times. */

```

```

    if (ln == NULL)
        listAddNodeTail(u->passwords,newpass);
    else
        sdsfree(newpass);
    u->flags &= ~USER_FLAG_NOPASS;
} else if (op[0] == '<' || op[0] == '!') {
    sds delpass;
    if (op[0] == '<') {
        delpass = ACLHashPassword((unsigned char*)op+1,oplen-1);
    } else {
        if (ACLCheckPasswordHash((unsigned char*)op+1,oplen-1) == C_ERR) {
            errno = EBADMSG;
            return C_ERR;
        }
        delpass = sdsnewlen(op+1,oplen-1);
    }
    listNode *ln = listSearchKey(u->passwords,delpass);
    sdsfree(delpass);
    if (ln) {
        listDelNode(u->passwords,ln);
    } else {
        errno = ENODEV;
        return C_ERR;
    }
} else if (op[0] == '(' && op[oplen - 1] == ')') {
    aclSelector *selector = aclCreateSelectorFromOpSet(op, oplen);
    if (!selector) {
        /* No errno set, propagate it from interior error. */
        return C_ERR;
    }
    listAddNodeTail(u->selectors, selector);
    return C_OK;
} else if (!strcasecmp(op,"clearselectors")) {
    listIter li;
    listNode *ln;
    listRewind(u->selectors,&li);
    /* There has to be a root selector */
    serverAssert(listNext(&li));
    while((ln = listNext(&li))) {
        listDelNode(u->selectors, ln);
    }
    return C_OK;
} else if (!strcasecmp(op,"reset")) {
    serverAssert(ACLSetUser(u,"resetpass",-1) == C_OK);
    serverAssert(ACLSetUser(u,"resetkeys",-1) == C_OK);
    serverAssert(ACLSetUser(u,"resetchannels",-1) == C_OK);
    if (server.acl_pubsub_default & SELECTOR_FLAG_ALLCHANNELS)
        serverAssert(ACLSetUser(u,"allchannels",-1) == C_OK);
    serverAssert(ACLSetUser(u,"off",-1) == C_OK);
    serverAssert(ACLSetUser(u,"sanitize-payload",-1) == C_OK);
    serverAssert(ACLSetUser(u,"clearselectors",-1) == C_OK);
}

```

```

        serverAssert(ACLSetUser(u,"-@all",-1) == C_OK);
    } else {
        aclSelector *selector = ACLUserGetRootSelector(u);
        if (ACLSetSelector(selector, op, opLen) == C_ERR) {
            return C_ERR;
        }
    }
    return C_OK;
}

/* Return a description of the error that occurred in ACLSetUser() according to
 * the errno value set by the function on error. */
const char *ACLSetUserStringError(void) {
    const char *errmsg = "Wrong format";
    if (errno == ENOENT)
        errmsg = "Unknown command or category name in ACL";
    else if (errno == EINVAL)
        errmsg = "Syntax error";
    else if (errno == EEXIST)
        errmsg = "Adding a pattern after the * pattern (or the "
            "'allkeys' flag) is not valid and does not have any "
            "effect. Try 'resetkeys' to start with an empty "
            "list of patterns";
    else if (errno == EISDIR)
        errmsg = "Adding a pattern after the * pattern (or the "
            "'allchannels' flag) is not valid and does not have any "
            "effect. Try 'resetchannels' to start with an empty "
            "list of channels";
    else if (errno == ENODEV)
        errmsg = "The password you are trying to remove from the user does "
            "not exist";
    else if (errno == EBADMSG)
        errmsg = "The password hash must be exactly 64 characters and contain "
            "only lowercase hexadecimal characters";
    else if (errno == EALREADY)
        errmsg = "Duplicate user found. A user can only be defined once in "
            "config files";
    else if (errno == ECHILD)
        errmsg = "Allowing first-arg of a subcommand is not supported";
    return errmsg;
}

/* Create the default user, this has special permissions. */
user *ACLCreateDefaultUser(void) {
    user *new = ACLCreateUser("default",7);
    ACLSetUser(new,"+@all",-1);
    ACLSetUser(new,"~*", -1);
    ACLSetUser(new,"&*", -1);
    ACLSetUser(new,"on",-1);
    ACLSetUser(new,"nopass",-1);
    return new;
}

```

```

}

/* Initialization of the ACL subsystem. */
void ACLInit(void) {
    Users = raxNew();
    UsersToLoad = listCreate();
    listSetMatchMethod(UsersToLoad, ACLListMatchLoadedUser);
    ACLLog = listCreate();
    DefaultUser = ACLCreateDefaultUser();
}

/* Check the username and password pair and return C_OK if they are valid,
 * otherwise C_ERR is returned and errno is set to:
 *
 * * EINVAL: if the username-password do not match.
 * * ENOENT: if the specified user does not exist at all.
 */
int ACLCheckUserCredentials(robj *username, robj *password) {
    user *u = ACLGetUserByName(username->ptr,sdslen(username->ptr));
    if (u == NULL) {
        errno = ENOENT;
        return C_ERR;
    }

    /* Disabled users can't login. */
    if (u->flags & USER_FLAG_DISABLED) {
        errno = EINVAL;
        return C_ERR;
    }

    /* If the user is configured to don't require any password, we
     * are already fine here. */
    if (u->flags & USER_FLAG_NOPASS) return C_OK;

    /* Check all the user passwords for at least one to match. */
    listIter li;
    listNode *ln;
    listRewind(u->passwords,&li);
    sds hashed = ACLHashPassword(password->ptr,sdslen(password->ptr));
    while((ln = listNext(&li))) {
        sds thispass = listNodeValue(ln);
        if (!time_independent_strcmp(hashed, thispass)) {
            sdsfree(hashed);
            return C_OK;
        }
    }
    sdsfree(hashed);

    /* If we reached this point, no password matched. */
    errno = EINVAL;
    return C_ERR;
}

```

```

}

/* This is like ACLCheckUserCredentials(), however if the user/pass
 * are correct, the connection is put in authenticated state and the
 * connection user reference is populated.
 *
 * The return value is C_OK or C_ERR with the same meaning as
 * ACLCheckUserCredentials(). */
int ACLAuthenticateUser(client *c, robj *username, robj *password) {
    if (ACLCheckUserCredentials(username,password) == C_OK) {
        c->authenticated = 1;
        c->user = ACLGetUserByName(username->ptr,sdslen(username->ptr));
        moduleNotifyUserChanged(c);
        return C_OK;
    } else {
        addACLLogEntry(c,ACL_DENIED_AUTH,(c->flags & CLIENT_MULTI) ?
ACL_LOG_CTX_MULTI : ACL_LOG_CTX_TOPLEVEL,0,username->ptr,NULL);
        return C_ERR;
    }
}

/* For ACL purposes, every user has a bitmap with the commands that such
 * user is allowed to execute. In order to populate the bitmap, every command
 * should have an assigned ID (that is used to index the bitmap). This function
 * creates such an ID: it uses sequential IDs, reusing the same ID for the same
 * command name, so that a command retains the same ID in case of modules that
 * are unloaded and later reloaded.
 *
 * The function does not take ownership of the 'cmdname' SDS string.
 */
unsigned long ACLGetCommandID(sds cmdname) {
    sds lowername = sdsdup(cmdname);
    sdstolower(lowername);
    if (commandId == NULL) commandId = raxNew();
    void *id = raxFind(commandId,(unsigned char*)lowername,sdslen(lowername));
    if (id != raxNotFound) {
        sdsfree(lowername);
        return (unsigned long)id;
    }
    raxInsert(commandId,(unsigned char*)lowername,strlen(lowername),
        (void*)nextid,NULL);
    sdsfree(lowername);
    unsigned long thisid = nextid;
    nextid++;

    /* We never assign the last bit in the user commands bitmap structure,
     * this way we can later check if this bit is set, understanding if the
     * current ACL for the user was created starting with a +@all to add all
     * the possible commands and just subtracting other single commands or
     * categories, or if, instead, the ACL was created just adding commands
     * and command categories from scratch, not allowing future commands by

```

```

    * default (loaded via modules). This is useful when rewriting the ACLs
    * with ACL SAVE. */
    if (nextid == USER_COMMAND_BITS_COUNT-1) nextid++;
    return thisid;
}

/* Clear command id table and reset nextid to 0. */
void ACLClearCommandID(void) {
    if (commandId) raxFree(commandId);
    commandId = NULL;
    nextid = 0;
}

/* Return an username by its name, or NULL if the user does not exist. */
user *ACLGetUserByName(const char *name, size_t namelen) {
    void *myuser = raxFind(Users, (unsigned char *)name, namelen);
    if (myuser == raxNotFound) return NULL;
    return myuser;
}

/*
=====
* ACL permission checks
* =====*/

/* Check if the key can be accessed by the selector.
*
* If the selector can access the key, ACL_OK is returned, otherwise
* ACL_DENIED_KEY is returned. */
static int ACLSelectorCheckKey(aclSelector *selector, const char *key, int
keylen, int keyspec_flags) {
    /* The selector can access any key */
    if (selector->flags & SELECTOR_FLAG_ALLKEYS) return ACL_OK;

    listIter li;
    listNode *ln;
    listRewind(selector->patterns, &li);

    int key_flags = 0;
    if (keyspec_flags & CMD_KEY_ACCESS) key_flags |= ACL_READ_PERMISSION;
    if (keyspec_flags & CMD_KEY_INSERT) key_flags |= ACL_WRITE_PERMISSION;
    if (keyspec_flags & CMD_KEY_DELETE) key_flags |= ACL_WRITE_PERMISSION;
    if (keyspec_flags & CMD_KEY_UPDATE) key_flags |= ACL_WRITE_PERMISSION;

    /* Test this key against every pattern. */
    while((ln = listNext(&li))) {
        keyPattern *pattern = listNodeValue(ln);
        if ((pattern->flags & key_flags) != key_flags)
            continue;
        size_t plen = sdslen(pattern->pattern);
        if (stringmatchlen(pattern->pattern, plen, key, keylen, 0))

```



```

        return ACL_OK;
    }
    return ACL_DENIED_KEY;
}

/* Checks if the provided selector selector has access specified in flags
 * to all keys in the keyspace. For example, CMD_KEY_READ access requires
 * either
 * * '%R~*', '~*', or allkeys to be granted to the selector. Returns 1 if all
 * * the access flags are satisfied with this selector or 0 otherwise.
 */
static int ACLSelectorHasUnrestrictedKeyAccess(aclSelector *selector, int
flags) {
    /* The selector can access any key */
    if (selector->flags & SELECTOR_FLAG_ALLKEYS) return 1;

    listIter li;
    listNode *ln;
    listRewind(selector->patterns,&li);

    int access_flags = 0;
    if (flags & CMD_KEY_ACCESS) access_flags |= ACL_READ_PERMISSION;
    if (flags & CMD_KEY_INSERT) access_flags |= ACL_WRITE_PERMISSION;
    if (flags & CMD_KEY_DELETE) access_flags |= ACL_WRITE_PERMISSION;
    if (flags & CMD_KEY_UPDATE) access_flags |= ACL_WRITE_PERMISSION;

    /* Test this key against every pattern. */
    while((ln = listNext(&li))) {
        keyPattern *pattern = listNodeValue(ln);
        if ((pattern->flags & access_flags) != access_flags)
            continue;
        if (!strcmp(pattern->pattern,"*")) {
            return 1;
        }
    }
    return 0;
}

/* Checks a channel against a provided list of channels. The is_pattern
 * argument should only be used when subscribing (not when publishing)
 * and controls whether the input channel is evaluated as a channel pattern
 * (like in PSUBSCRIBE) or a plain channel name (like in SUBSCRIBE).
 *
 * Note that a plain channel name like in PUBLISH or SUBSCRIBE can be
 * matched against ACL channel patterns, but the pattern provided in PSUBSCRIBE
 * can only be matched as a literal against an ACL pattern (using plain string
 * compare). */
static int ACLCheckChannelAgainstList(list *reference, const char *channel, int
channellen, int is_pattern) {
    listIter li;
    listNode *ln;

```

```

listRewind(reference, &li);
while((ln = listNext(&li))) {
    sds pattern = listNodeValue(ln);
    size_t plen = sdslen(pattern);
    /* Channel patterns are matched literally against the channels in
     * the list. Regular channels perform pattern matching. */
    if ((is_pattern && !strcmp(pattern,channel)) ||
        (!is_pattern && stringmatchlen(pattern,plen,channel,channellen,0)))
    {
        return ACL_OK;
    }
}
return ACL_DENIED_CHANNEL;
}

/* To prevent duplicate calls to getKeysResult, a cache is maintained
 * in between calls to the various selectors. */
typedef struct {
    int keys_init;
    getKeysResult keys;
} aclKeyResultCache;

void initACLKeyResultCache(aclKeyResultCache *cache) {
    cache->keys_init = 0;
}

void cleanupACLKeyResultCache(aclKeyResultCache *cache) {
    if (cache->keys_init) getKeysFreeResult(&(cache->keys));
}

/* Check if the command is ready to be executed according to the
 * ACLs associated with the specified selector.
 *
 * If the selector can execute the command ACL_OK is returned, otherwise
 * ACL_DENIED_CMD, ACL_DENIED_KEY, or ACL_DENIED_CHANNEL is returned: the first
in case the
 * command cannot be executed because the selector is not allowed to run such
 * command, the second and third if the command is denied because the selector
is trying
 * to access a key or channel that are not among the specified patterns. */
static int ACLSelectorCheckCmd(aclSelector *selector, struct redisCommand *cmd,
robj **argv, int argc, int *keyidxptr, aclKeyResultCache *cache) {
    uint64_t id = cmd->id;
    int ret;
    if (!(selector->flags & SELECTOR_FLAG_ALLCOMMANDS) && !(cmd->flags &
CMD_NO_AUTH)) {
        /* If the bit is not set we have to check further, in case the
         * command is allowed just with that specific first argument. */
        if (ACLGetSelectorCommandBit(selector,id) == 0) {
            /* Check if the first argument matches. */

```

```

        if (argc < 2 ||
            selector->allowed_firstargs == NULL ||
            selector->allowed_firstargs[id] == NULL)
        {
            return ACL_DENIED_CMD;
        }

        long subid = 0;
        while (1) {
            if (selector->allowed_firstargs[id][subid] == NULL)
                return ACL_DENIED_CMD;
            int idx = cmd->parent ? 2 : 1;
            if (!strcasecmp(argv[idx]->ptr, selector->allowed_firstargs[id]
[subid]))
                break; /* First argument match found. Stop here. */
            subid++;
        }
    }

    /* Check if the user can execute commands explicitly touching the keys
    * mentioned in the command arguments. */
    if (!(selector->flags & SELECTOR_FLAG_ALLKEYS) && doesCommandHaveKeys(cmd))
    {
        if (!(cache->keys_init)) {
            cache->keys = (getKeyResult) GETKEYS_RESULT_INIT;
            getKeyFromCommandWithSpecs(cmd, argv, argc, GET_KEYSPEC_DEFAULT, &
(cache->keys));
            cache->keys_init = 1;
        }
        getKeyResult *result = &(cache->keys);
        keyReference *resultidx = result->keys;
        for (int j = 0; j < result->numkeys; j++) {
            int idx = resultidx[j].pos;
            ret = ACLSelectorCheckKey(selector, argv[idx]->ptr,
sdslen(argv[idx]->ptr), resultidx[j].flags);
            if (ret != ACL_OK) {
                if (keyidxptr) *keyidxptr = resultidx[j].pos;
                return ret;
            }
        }
    }

    /* Check if the user can execute commands explicitly touching the channels
    * mentioned in the command arguments */
    const int channel_flags = CMD_CHANNEL_PUBLISH | CMD_CHANNEL_SUBSCRIBE;
    if (!(selector->flags & SELECTOR_FLAG_ALLCHANNELS) &&
doesCommandHaveChannelsWithFlags(cmd, channel_flags)) {
        getKeyResult channels = (getKeyResult) GETKEYS_RESULT_INIT;
        getChannelsFromCommand(cmd, argv, argc, &channels);
        keyReference *channelref = channels.keys;

```

```

        for (int j = 0; j < channels.numkeys; j++) {
            int idx = channelref[j].pos;
            if (!(channelref[j].flags & channel_flags)) continue;
            int is_pattern = channelref[j].flags & CMD_CHANNEL_PATTERN;
            int ret = ACLCheckChannelAgainstList(selector->channels, argv[idx]-
>ptr, sdslen(argv[idx]->ptr), is_pattern);
            if (ret != ACL_OK) {
                if (keyidxptr) *keyidxptr = channelref[j].pos;
                getKeysFreeResult(&channels);
                return ret;
            }
        }
        getKeysFreeResult(&channels);
    }
    return ACL_OK;
}

```

```

/* Check if the key can be accessed by the client according to
 * the ACLs associated with the specified user according to the
 * keyspec access flags.
 */

```

```

/* If the user can access the key, ACL_OK is returned, otherwise
 * ACL_DENIED_KEY is returned. */

```

```

int ACLUserCheckKeyPerm(user *u, const char *key, int keylen, int flags) {
    listIter li;
    listNode *ln;

    /* If there is no associated user, the connection can run anything. */
    if (u == NULL) return ACL_OK;

    /* Check all of the selectors */
    listRewind(u->selectors,&li);
    while((ln = listNext(&li))) {
        aclSelector *s = (aclSelector *) listNodeValue(ln);
        if (ACLSelectorCheckKey(s, key, keylen, flags) == ACL_OK) {
            return ACL_OK;
        }
    }
    return ACL_DENIED_KEY;
}

```

```

/* Checks if the user can execute the given command with the added restriction
 * it must also have the access specified in flags to any key in the key space.
 * For example, CMD_KEY_READ access requires either '%R~*', '~*', or allkeys to
 be
 * granted in addition to the access required by the command. Returns 1
 * if the user has access or 0 otherwise.
 */

```

```

int ACLUserCheckCmdWithUnrestrictedKeyAccess(user *u, struct redisCommand *cmd,
robj **argv, int argc, int flags) {
    listIter li;

```

```

listNode *ln;
int local_idxptr;

/* If there is no associated user, the connection can run anything. */
if (u == NULL) return 1;

/* For multiple selectors, we cache the key result in between selector
 * calls to prevent duplicate lookups. */
aclKeyResultCache cache;
initACLKeyResultCache(&cache);

/* Check each selector sequentially */
listRewind(u->selectors,&li);
while((ln = listNext(&li))) {
    aclSelector *s = (aclSelector *) listNodeValue(ln);
    int acl_retval = ACLSelectorCheckCmd(s, cmd, argv, argc, &local_idxptr,
&cache);
    if (acl_retval == ACL_OK && ACLSelectorHasUnrestrictedKeyAccess(s,
flags)) {
        cleanupACLKeyResultCache(&cache);
        return 1;
    }
}
cleanupACLKeyResultCache(&cache);
return 0;
}

/* Check if the channel can be accessed by the client according to
 * the ACLs associated with the specified user.
 *
 * If the user can access the key, ACL_OK is returned, otherwise
 * ACL_DENIED_CHANNEL is returned. */
int ACLUserCheckChannelPerm(user *u, sds channel, int is_pattern) {
    listIter li;
    listNode *ln;

    /* If there is no associated user, the connection can run anything. */
    if (u == NULL) return ACL_OK;

    /* Check all of the selectors */
    listRewind(u->selectors,&li);
    while((ln = listNext(&li))) {
        aclSelector *s = (aclSelector *) listNodeValue(ln);
        /* The selector can run any keys */
        if (s->flags & SELECTOR_FLAG_ALLCHANNELS) return ACL_OK;

        /* Otherwise, loop over the selectors list and check each channel */
        if (ACLCheckChannelAgainstList(s->channels, channel, sdslen(channel),
is_pattern) == ACL_OK) {
            return ACL_OK;
        }
    }
}

```

```

    }
    return ACL_DENIED_CHANNEL;
}

/* Lower level API that checks if a specified user is able to execute a given
command. */
int ACLCheckAllUserCommandPerm(user *u, struct redisCommand *cmd, robj **argv,
int argc, int *idxptr) {
    listIter li;
    listNode *ln;

    /* If there is no associated user, the connection can run anything. */
    if (u == NULL) return ACL_OK;

    /* We have to pick a single error to log, the logic for picking is as
follows:
    * 1) If no selector can execute the command, return the command.
    * 2) Return the last key or channel that no selector could match. */
    int relevant_error = ACL_DENIED_CMD;
    int local_idxptr = 0, last_idx = 0;

    /* For multiple selectors, we cache the key result in between selector
    * calls to prevent duplicate lookups. */
    aclKeyResultCache cache;
    initACLKeyResultCache(&cache);

    /* Check each selector sequentially */
    listRewind(u->selectors,&li);
    while((ln = listNext(&li))) {
        aclSelector *s = (aclSelector *) listNodeValue(ln);
        int acl_retval = ACLSelectorCheckCmd(s, cmd, argv, argc, &local_idxptr,
&cache);
        if (acl_retval == ACL_OK) {
            cleanupACLKeyResultCache(&cache);
            return ACL_OK;
        }
        if (acl_retval > relevant_error ||
            (acl_retval == relevant_error && local_idxptr > last_idx))
        {
            relevant_error = acl_retval;
            last_idx = local_idxptr;
        }
    }

    *idxptr = last_idx;
    cleanupACLKeyResultCache(&cache);
    return relevant_error;
}

/* High level API for checking if a client can execute the queued up command */
int ACLCheckAllPerm(client *c, int *idxptr) {

```

```

    return ACLCheckAllUserCommandPerm(c->user, c->cmd, c->argv, c->argc,
idxptr);
}

/* Check if the user's existing pub/sub clients violate the ACL pub/sub
 * permissions specified via the upcoming argument, and kill them if so. */
void ACLKillPubsubClientsIfNeeded(user *new, user *original) {
    listIter li, lpi;
    listNode *ln, *lpn;
    robj *o;
    int kill = 0;

    /* First optimization is we check if any selector has all channel
     * permissions. */
    listRewind(new->selectors,&li);
    while((ln = listNext(&li))) {
        aclSelector *s = (aclSelector *) listNodeValue(ln);
        if (s->flags & SELECTOR_FLAG_ALLCHANNELS) return;
    }

    /* Second optimization is to check if the new list of channels
     * is a strict superset of the original. This is done by
     * created an "upcoming" list of all channels that are in
     * the new user and checking each of the existing channels
     * against it. */
    list *upcoming = listCreate();
    listRewind(new->selectors,&li);
    while((ln = listNext(&li))) {
        aclSelector *s = (aclSelector *) listNodeValue(ln);
        listRewind(s->channels, &lpi);
        while((lpn = listNext(&lpi))) {
            listAddNodeTail(upcoming, listNodeValue(lpn));
        }
    }

    int match = 1;
    listRewind(original->selectors,&li);
    while((ln = listNext(&li)) && match) {
        aclSelector *s = (aclSelector *) listNodeValue(ln);
        listRewind(s->channels, &lpi);
        while((lpn = listNext(&lpi)) && match) {
            if (!listSearchKey(upcoming, listNodeValue(lpn))) {
                match = 0;
                break;
            }
        }
    }

    if (match) {
        /* All channels were matched, no need to kill clients. */
        listRelease(upcoming);
    }
}

```

```

        return;
    }

    /* Permissions have changed, so we need to iterate through all
     * the clients and disconnect those that are no longer valid.
     * Scan all connected clients to find the user's pub/subs. */
    listRewind(server.clients,&li);
    while ((ln = listNext(&li)) != NULL) {
        client *c = listNodeValue(ln);
        kill = 0;

        if (c->user == original && getClientType(c) == CLIENT_TYPE_PUBSUB) {
            /* Check for pattern violations. */
            listRewind(c->pubsub_patterns,&lpi);
            while (!kill && ((lpn = listNext(&lpi)) != NULL)) {

                o = lpn->value;
                int res = ACLCheckChannelAgainstList(upcoming, o->ptr,
sdslen(o->ptr), 1);
                kill = (res == ACL_DENIED_CHANNEL);
            }
            /* Check for channel violations. */
            if (!kill) {
                /* Check for global channels violation. */
                dictIterator *di = dictGetIterator(c->pubsub_channels);

                dictEntry *de;
                while (!kill && ((de = dictNext(di)) != NULL)) {
                    o = dictGetKey(de);
                    int res = ACLCheckChannelAgainstList(upcoming, o->ptr,
sdslen(o->ptr), 0);
                    kill = (res == ACL_DENIED_CHANNEL);
                }
                dictReleaseIterator(di);

                /* Check for shard channels violation. */
                di = dictGetIterator(c->pubsubshard_channels);
                while (!kill && ((de = dictNext(di)) != NULL)) {
                    o = dictGetKey(de);
                    int res = ACLCheckChannelAgainstList(upcoming, o->ptr,
sdslen(o->ptr), 0);
                    kill = (res == ACL_DENIED_CHANNEL);
                }

                dictReleaseIterator(di);
            }

            /* Kill it. */
            if (kill) {
                freeClient(c);
            }
        }
    }
}

```



```

    }
    }
    listRelease(upcoming);
}

/*
=====
* ACL loading / saving functions
* =====*/

/* Selector definitions should be sent as a single argument, however
* we will be lenient and try to find selector definitions spread
* across multiple arguments since it makes for a simpler user experience
* for ACL SETUSER as well as when loading from conf files.
*
* This function takes in an array of ACL operators, excluding the username,
* and merges selector operations that are spread across multiple arguments.
The return
* value is a new SDS array, with length set to the passed in merged_argc.
Arguments
* that are untouched are still duplicated. If there is an unmatched
parenthesis, NULL
* is returned and invalid_idx is set to the argument with the start of the
opening
* parenthesis. */
sds *ACLMergeSelectorArguments(sds *argv, int argc, int *merged_argc, int
*invalid_idx) {
    *merged_argc = 0;
    int open_bracket_start = -1;

    sds *acl_args = (sds *) zmalloc(sizeof(sds) * argc);

    sds selector = NULL;
    for (int j = 0; j < argc; j++) {
        char *op = argv[j];

        if (op[0] == '(' && op[sdslen(op) - 1] != ')') {
            selector = sdsdup(argv[j]);
            open_bracket_start = j;
            continue;
        }

        if (open_bracket_start != -1) {
            selector = sdscatfmt(selector, " %s", op);
            if (op[sdslen(op) - 1] == ')') {
                open_bracket_start = -1;
                acl_args[*merged_argc] = selector;
                (*merged_argc)++;
            }
            continue;
        }
    }
}

```

```

    }

    acl_args[*merged_argc] = sdsdup(argv[j]);
    (*merged_argc)++;
}

if (open_bracket_start != -1) {
    for (int i = 0; i < *merged_argc; i++) sdsfree(acl_args[i]);
    zfree(acl_args);
    sdsfree(selector);
    if (invalid_idx) *invalid_idx = open_bracket_start;
    return NULL;
}

return acl_args;
}

/* Given an argument vector describing a user in the form:
 *
 *      user <username> ... ACL rules and flags ...
 *
 * this function validates, and if the syntax is valid, appends
 * the user definition to a list for later loading.
 *
 * The rules are tested for validity and if there obvious syntax errors
 * the function returns C_ERR and does nothing, otherwise C_OK is returned
 * and the user is appended to the list.
 *
 * Note that this function cannot stop in case of commands that are not found
 * and, in that case, the error will be emitted later, because certain
 * commands may be defined later once modules are loaded.
 *
 * When an error is detected and C_ERR is returned, the function populates
 * by reference (if not set to NULL) the argc_err argument with the index
 * of the argv vector that caused the error. */
int ACLAppendUserForLoading(sds *argv, int argc, int *argc_err) {
    if (argc < 2 || strcasecmp(argv[0], "user")) {
        if (argc_err) *argc_err = 0;
        return C_ERR;
    }

    if (listSearchKey(UsersToLoad, argv[1])) {
        if (argc_err) *argc_err = 1;
        errno = EALREADY;
        return C_ERR;
    }

    /* Try to apply the user rules in a fake user to see if they
     * are actually valid. */
    user *fakeuser = ACLCreateUnlinkedUser();

```

```

    /* Merged selectors before trying to process */
    int merged_argc;
    sds *acl_args = ACLMergeSelectorArguments(argv + 2, argc - 2, &merged_argc,
    argc_err);

    if (!acl_args) {
        return C_ERR;
    }

    for (int j = 0; j < merged_argc; j++) {
        if (ACLSetUser(fakeuser,acl_args[j],sdslen(acl_args[j])) == C_ERR) {
            if (errno != ENOENT) {
                ACLFreeUser(fakeuser);
                if (argc_err) *argc_err = j;
                for (int i = 0; i < merged_argc; i++) sdsfree(acl_args[i]);
                zfree(acl_args);
                return C_ERR;
            }
        }
    }

    /* Rules look valid, let's append the user to the list. */
    sds *copy = zmalloc(sizeof(sds)*(merged_argc + 2));
    copy[0] = sdsdup(argv[1]);
    for (int j = 0; j < merged_argc; j++) copy[j+1] = sdsdup(acl_args[j]);
    copy[merged_argc + 1] = NULL;
    listAddNodeTail(UsersToLoad,copy);
    ACLFreeUser(fakeuser);
    for (int i = 0; i < merged_argc; i++) sdsfree(acl_args[i]);
    zfree(acl_args);
    return C_OK;
}

/* This function will load the configured users appended to the server
 * configuration via ACLAppendUserForLoading(). On loading errors it will
 * log an error and return C_ERR, otherwise C_OK will be returned. */
int ACLLoadConfiguredUsers(void) {
    listIter li;
    listNode *ln;
    listRewind(UsersToLoad,&li);
    while ((ln = listNext(&li)) != NULL) {
        sds *aclrules = listNodeValue(ln);
        sds username = aclrules[0];

        if (ACLStringHasSpaces(aclrules[0],sdslen(aclrules[0]))) {
            serverLog(LL_WARNING,"Spaces not allowed in ACL usernames");
            return C_ERR;
        }

        user *u = ACLCreateUser(username,sdslen(username));
        if (!u) {

```

```

        /* Only valid duplicate user is the default one. */
        serverAssert(!strcmp(username, "default"));
        u = ACLGetUserByName("default",7);
        ACLSetUser(u,"reset",-1);
    }

    /* Load every rule defined for this user. */
    for (int j = 1; aclrules[j]; j++) {
        if (ACLSetUser(u,aclrules[j],sdslen(aclrules[j])) != C_OK) {
            const char *errmsg = ACLSetUserStringError();
            serverLog(LL_WARNING,"Error loading ACL rule '%s' for "
                        "the user named '%s': %s",
                        aclrules[j],aclrules[0],errmsg);
            return C_ERR;
        }
    }

    /* Having a disabled user in the configuration may be an error,
     * warn about it without returning any error to the caller. */
    if (u->flags & USER_FLAG_DISABLED) {
        serverLog(LL_NOTICE, "The user '%s' is disabled (there is no "
                        "'on' modifier in the user description). Make
"
                        "sure this is not a configuration error.",
                        aclrules[0]);
    }
}

return C_OK;
}

/* This function loads the ACL from the specified filename: every line
 * is validated and should be either empty or in the format used to specify
 * users in the redis.conf configuration or in the ACL file, that is:
 *
 * user <username> ... rules ...
 *
 * Note that this function considers comments starting with '#' as errors
 * because the ACL file is meant to be rewritten, and comments would be
 * lost after the rewrite. Yet empty lines are allowed to avoid being too
 * strict.
 *
 * One important part of implementing ACL LOAD, that uses this function, is
 * to avoid ending with broken rules if the ACL file is invalid for some
 * reason, so the function will attempt to validate the rules before loading
 * each user. For every line that will be found broken the function will
 * collect an error message.
 *
 * IMPORTANT: If there is at least a single error, nothing will be loaded
 * and the rules will remain exactly as they were.
 *
 * At the end of the process, if no errors were found in the whole file then

```

```

* NULL is returned. Otherwise an SDS string describing in a single line
* a description of all the issues found is returned. */
sds ACLLoadFromFile(const char *filename) {
    FILE *fp;
    char buf[1024];

    /* Open the ACL file. */
    if ((fp = fopen(filename,"r")) == NULL) {
        sds errors = sdscatprintf(sdsempy(),
            "Error loading ACLs, opening file '%s': %s",
            filename, strerror(errno));
        return errors;
    }

    /* Load the whole file as a single string in memory. */
    sds acs = sdsempy();
    while(fgets(buf,sizeof(buf),fp) != NULL)
        acs = sdscat(acs,buf);
    fclose(fp);

    /* Split the file into lines and attempt to load each line. */
    int totlines;
    sds *lines, errors = sdsempy();
    lines = sdssplitlen(acs,strlen(acs),"\n",1,&totlines);
    sdsfree(acs);

    /* We do all the loading in a fresh instance of the Users radix tree,
     * so if there are errors loading the ACL file we can rollback to the
     * old version. */
    rax *old_users = Users;
    Users = raxNew();

    /* Load each line of the file. */
    for (int i = 0; i < totlines; i++) {
        sds *argv;
        int argc;
        int linenum = i+1;

        lines[i] = sdstrim(lines[i]," \t\r\n");

        /* Skip blank lines */
        if (lines[i][0] == '\0') continue;

        /* Split into arguments */
        argv = sdssplitlen(lines[i],sdslen(lines[i])," ",1,&argc);
        if (argv == NULL) {
            errors = sdscatprintf(errors,
                "%s:%d: unbalanced quotes in acl line. ",
                server.acl_filename, linenum);
            continue;
        }
    }
}

```

```

/* Skip this line if the resulting command vector is empty. */
if (argc == 0) {
    sdsfreesplitres(argv,argc);
    continue;
}

/* The line should start with the "user" keyword. */
if (strcmp(argv[0],"user") || argc < 2) {
    errors = sdscatprintf(errors,
        "%s:%d should start with user keyword followed "
        "by the username. ", server.acl_filename,
        linenum);
    sdsfreesplitres(argv,argc);
    continue;
}

/* Spaces are not allowed in usernames. */
if (ACLStringHasSpaces(argv[1],sdslen(argv[1]))) {
    errors = sdscatprintf(errors,
        "'%s:%d: username '%s' contains invalid characters. ",
        server.acl_filename, linenum, argv[1]);
    sdsfreesplitres(argv,argc);
    continue;
}

user *u = ACLCreateUser(argv[1],sdslen(argv[1]));

/* If the user already exists we assume it's an error and abort. */
if (!u) {
    errors = sdscatprintf(errors,"WARNING: Duplicate user '%s' found on
line %d. ", argv[1], linenum);
    sdsfreesplitres(argv,argc);
    continue;
}

/* Finally process the options and validate they can
 * be cleanly applied to the user. If any option fails
 * to apply, the other values won't be applied since
 * all the pending changes will get dropped. */
int merged_argc;
sds *acl_args = ACLMergeSelectorArguments(argv + 2, argc - 2,
&merged_argc, NULL);
if (!acl_args) {
    errors = sdscatprintf(errors,
        "%s:%d: Unmatched parenthesis in selector definition.",
        server.acl_filename, linenum);
}

int j;
for (j = 0; j < merged_argc; j++) {

```

```

        acl_args[j] = sdstrim(acl_args[j], "\t\r\n");
        if (ACLSetUser(u, acl_args[j], sdslen(acl_args[j])) != C_OK) {
            const char *errmsg = ACLSetUserStringError();
            errors = sdscatprintf(errors,
                                  "%s:%d: %s. ",
                                  server.acl_filename, linenum, errmsg);
            continue;
        }
    }

    for (int i = 0; i < merged_argc; i++) sdsfree(acl_args[i]);
    zfree(acl_args);

    /* Apply the rule to the new users set only if so far there
     * are no errors, otherwise it's useless since we are going
     * to discard the new users set anyway. */
    if (sdslen(errors) != 0) {
        sdsfreesplitres(argv, argc);
        continue;
    }

    sdsfreesplitres(argv, argc);
}

sdsfreesplitres(lines, totlines);

/* Check if we found errors and react accordingly. */
if (sdslen(errors) == 0) {
    /* The default user pointer is referenced in different places: instead
     * of replacing such occurrences it is much simpler to copy the new
     * default user configuration in the old one. */
    user *new_default = ACLGetUserByName("default", 7);
    if (!new_default) {
        new_default = ACLCreateDefaultUser();
    }

    ACLCopyUser(DefaultUser, new_default);
    ACLFreeUser(new_default);
    raxInsert(Users, (unsigned char*)"default", 7, DefaultUser, NULL);
    raxRemove(old_users, (unsigned char*)"default", 7, NULL);
    ACLFreeUsersSet(old_users);
    sdsfree(errors);
    return NULL;
} else {
    ACLFreeUsersSet(Users);
    Users = old_users;
    errors = sdscat(errors, "WARNING: ACL errors detected, no change to the
previously active ACL rules was performed");
    return errors;
}
}

```

```

/* Generate a copy of the ACLs currently in memory in the specified filename.
 * Returns C_OK on success or C_ERR if there was an error during the I/O.
 * When C_ERR is returned a log is produced with hints about the issue. */
int ACLSaveToFile(const char *filename) {
    sds acl = sdsempty();
    int fd = -1;
    sds tmpfilename = NULL;
    int retval = C_ERR;

    /* Let's generate an SDS string containing the new version of the
     * ACL file. */
    raxIterator ri;
    raxStart(&ri,Users);
    raxSeek(&ri,"^",NULL,0);
    while(raxNext(&ri)) {
        user *u = ri.data;
        /* Return information in the configuration file format. */
        sds user = sdsnew("user ");
        user = sdscatsds(user,u->name);
        user = sdscatlen(user," ",1);
        sds descr = ACLDescribeUser(u);
        user = sdscatsds(user,descr);
        sdsfree(descr);
        acl = sdscatsds(acl,user);
        acl = sdscatlen(acl,"\n",1);
        sdsfree(user);
    }
    raxStop(&ri);

    /* Create a temp file with the new content. */
    tmpfilename = sdsnew(filename);
    tmpfilename = sdscatfmt(tmpfilename,".tmp-%i-%I",
        (int)getpid(),(int)mstime());
    if ((fd = open(tmpfilename,O_WRONLY|O_CREAT,0644)) == -1) {
        serverLog(LL_WARNING,"Opening temp ACL file for ACL SAVE: %s",
            strerror(errno));
        goto cleanup;
    }

    /* Write it. */
    if (write(fd,acl,sdslen(acl)) != (ssize_t)sdslen(acl)) {
        serverLog(LL_WARNING,"Writing ACL file for ACL SAVE: %s",
            strerror(errno));
        goto cleanup;
    }
    close(fd); fd = -1;

    /* Let's replace the new file with the old one. */
    if (rename(tmpfilename,filename) == -1) {
        serverLog(LL_WARNING,"Renaming ACL file for ACL SAVE: %s",

```



```

        strerror(errno));
    goto cleanup;
}
sdsfree(tmpfilename); tmpfilename = NULL;
retval = C_OK; /* If we reached this point, everything is fine. */

cleanup:
    if (fd != -1) close(fd);
    if (tmpfilename) unlink(tmpfilename);
    sdsfree(tmpfilename);
    sdsfree(acl);
    return retval;
}

/* This function is called once the server is already running, modules are
 * loaded, and we are ready to start, in order to load the ACLs either from
 * the pending list of users defined in redis.conf, or from the ACL file.
 * The function will just exit with an error if the user is trying to mix
 * both the loading methods. */
void ACLLoadUsersAtStartup(void) {
    if (server.acl_filename[0] != '\0' && listLength(UsersToLoad) != 0) {
        serverLog(LL_WARNING,
            "Configuring Redis with users defined in redis.conf and at "
            "the same setting an ACL file path is invalid. This setup "
            "is very likely to lead to configuration errors and security "
            "holes, please define either an ACL file or declare users "
            "directly in your redis.conf, but not both.");
        exit(1);
    }

    if (ACLLoadConfiguredUsers() == C_ERR) {
        serverLog(LL_WARNING,
            "Critical error while loading ACLs. Exiting.");
        exit(1);
    }

    if (server.acl_filename[0] != '\0') {
        sds errors = ACLLoadFromFile(server.acl_filename);
        if (errors) {
            serverLog(LL_WARNING,
                "Aborting Redis startup because of ACL errors: %s", errors);
            sdsfree(errors);
            exit(1);
        }
    }
}

/*
=====
 * ACL log
 * =====*/

```

```

#define ACL_LOG_GROUPING_MAX_TIME_DELTA 60000

/* This structure defines an entry inside the ACL log. */
typedef struct ACLLogEntry {
    uint64_t count;        /* Number of times this happened recently. */
    int reason;            /* Reason for denying the command. ACL_DENIED_*. */
    int context;           /* Toplevel, Lua or MULTI/EXEC? ACL_LOG_CTX_*. */
    sds object;            /* The key name or command name. */
    sds username;          /* User the client is authenticated with. */
    mstime_t ctime;        /* Milliseconds time of last update to this entry. */
    sds cinfo;             /* Client info (last client if updated). */
} ACLLogEntry;

/* This function will check if ACL entries 'a' and 'b' are similar enough
 * that we should actually update the existing entry in our ACL log instead
 * of creating a new one. */
int ACLLogMatchEntry(ACLLogEntry *a, ACLLogEntry *b) {
    if (a->reason != b->reason) return 0;
    if (a->context != b->context) return 0;
    mstime_t delta = a->ctime - b->ctime;
    if (delta < 0) delta = -delta;
    if (delta > ACL_LOG_GROUPING_MAX_TIME_DELTA) return 0;
    if (sdscmp(a->object,b->object) != 0) return 0;
    if (sdscmp(a->username,b->username) != 0) return 0;
    return 1;
}

/* Release an ACL log entry. */
void ACLFreeLogEntry(void *leptr) {
    ACLLogEntry *le = leptr;
    sdsfree(le->object);
    sdsfree(le->username);
    sdsfree(le->cinfo);
    zfree(le);
}

/* Adds a new entry in the ACL log, making sure to delete the old entry
 * if we reach the maximum length allowed for the log. This function attempts
 * to find similar entries in the current log in order to bump the counter of
 * the log entry instead of creating many entries for very similar ACL
 * rules issues.
 *
 * The argpos argument is used when the reason is ACL_DENIED_KEY or
 * ACL_DENIED_CHANNEL, since it allows the function to log the key or channel
 * name that caused the problem.
 *
 * The last 2 arguments are a manual override to be used, instead of any of the
 * automatic
 * ones which depend on the client and reason arguments (use NULL for default).
 */

```

```

    * If `object` is not NULL, this functions takes over it.
    */
void addACLLogEntry(client *c, int reason, int context, int argpos, sds
username, sds object) {
    /* Create a new entry. */
    struct ACLLogEntry *le = zmalloc(sizeof(*le));
    le->count = 1;
    le->reason = reason;
    le->username = sdsdup(username ? username : c->user->name);
    le->ctime = mstime();

    if (object) {
        le->object = object;
    } else {
        switch(reason) {
            case ACL_DENIED_CMD: le->object = sdsdup(c->cmd->fullname); break;
            case ACL_DENIED_KEY: le->object = sdsdup(c->argv[argpos]->ptr);
break;
            case ACL_DENIED_CHANNEL: le->object = sdsdup(c->argv[argpos]->ptr);
break;
            case ACL_DENIED_AUTH: le->object = sdsdup(c->argv[0]->ptr); break;
            default: le->object = sdsempty();
        }
    }

    client *realclient = c;
    if (realclient->flags & CLIENT_SCRIPT) realclient = server.script_caller;

    le->cinfo = catClientInfoString(sdsempty(),realclient);
    le->context = context;

    /* Try to match this entry with past ones, to see if we can just
     * update an existing entry instead of creating a new one. */
    long toscan = 10; /* Do a limited work trying to find duplicated. */
    listIter li;
    listNode *ln;
    listRewind(ACLLog,&li);
    ACLLogEntry *match = NULL;
    while (toscan-- && (ln = listNext(&li)) != NULL) {
        ACLLogEntry *current = listNodeValue(ln);
        if (ACLLogMatchEntry(current,le) {
            match = current;
            listDelNode(ACLLog,ln);
            listAddNodeHead(ACLLog,current);
            break;
        }
    }

    /* If there is a match update the entry, otherwise add it as a
     * new one. */
    if (match) {

```

```

    /* We update a few fields of the existing entry and bump the
     * counter of events for this entry. */
    sdsfree(match->cinfo);
    match->cinfo = le->cinfo;
    match->ctime = le->ctime;
    match->count++;

    /* Release the old entry. */
    le->cinfo = NULL;
    ACLFreeLogEntry(le);
} else {
    /* Add it to our list of entries. We'll have to trim the list
     * to its maximum size. */
    listAddNodeHead(ACLLog, le);
    while(listLength(ACLLog) > server.acllog_max_len) {
        listNode *ln = listLast(ACLLog);
        ACLLogEntry *le = listNodeValue(ln);
        ACLFreeLogEntry(le);
        listDelNode(ACLLog, ln);
    }
}
}

const char* getAcLErrorMessage(int acl_res) {
    /* Notice that a variant of this code also exists on aclCommand so
     * it also need to be updated on changed. */
    switch (acl_res) {
    case ACL_DENIED_CMD:
        return "can't run this command or subcommand";
    case ACL_DENIED_KEY:
        return "can't access at least one of the keys mentioned in the command
arguments";
    case ACL_DENIED_CHANNEL:
        return "can't publish to the channel mentioned in the command";
    default:
        return "lacking the permissions for the command";
    }
    serverPanic("Reached deadcode on getAcLErrorMessage");
}

/*
=====
 * ACL related commands
 * =====*/

/* ACL CAT category */
void aclCatWithFlags(client *c, dict *commands, uint64_t cflag, int *arraylen)
{
    dictEntry *de;
    dictIterator *di = dictGetIterator(commands);

```

```

while ((de = dictNext(di)) != NULL) {
    struct redisCommand *cmd = dictGetVal(de);
    if (cmd->flags & CMD_MODULE) continue;
    if (cmd->acl_categories & cflag) {
        addReplyBulkCBuffer(c, cmd->fullname, sdslen(cmd->fullname));
        (*arraylen)++;
    }

    if (cmd->subcommands_dict) {
        aclCatWithFlags(c, cmd->subcommands_dict, cflag, arraylen);
    }
}
dictReleaseIterator(di);
}

/* Add the formatted response from a single selector to the ACL GETUSER
 * response. This function returns the number of fields added.
 *
 * Setting verbose to 1 means that the full qualifier for key and channel
 * permissions are shown.
 */
int aclAddReplySelectorDescription(client *c, aclSelector *s) {
    listIter li;
    listNode *ln;

    /* Commands */
    addReplyBulkCString(c,"commands");
    sds cmddescr = ACLDescribeSelectorCommandRules(s);
    addReplyBulkSds(c,cmddescr);

    /* Key patterns */
    addReplyBulkCString(c,"keys");
    if (s->flags & SELECTOR_FLAG_ALLKEYS) {
        addReplyBulkCBuffer(c,"~*",2);
    } else {
        sds dsl = sdsempty();
        listRewind(s->patterns,&li);
        while((ln = listNext(&li))) {
            keyPattern *thispat = (keyPattern *) listNodeValue(ln);
            if (ln != listFirst(s->patterns)) dsl = sdscat(dsl, " ");
            dsl = sdsCatPatternString(dsl, thispat);
        }
        addReplyBulkSds(c, dsl);
    }

    /* Pub/sub patterns */
    addReplyBulkCString(c,"channels");
    if (s->flags & SELECTOR_FLAG_ALLCHANNELS) {
        addReplyBulkCBuffer(c,"&*",2);
    } else {
        sds dsl = sdsempty();

```

```

        listRewind(s->channels,&li);
        while((ln = listNext(&li)) {
            sds thispat = listNodeValue(ln);
            if (ln != listFirst(s->channels)) dsl = sdscat(dsl, " ");
            dsl = sdscatfmt(dsl, "%S", thispat);
        }
        addReplyBulkSds(c, dsl);
    }
    return 3;
}

/* ACL -- show and modify the configuration of ACL users.
 * ACL HELP
 * ACL LOAD
 * ACL SAVE
 * ACL LIST
 * ACL USERS
 * ACL CAT [<category>]
 * ACL SETUSER <username> ... acl rules ...
 * ACL DELUSER <username> [...]
 * ACL GETUSER <username>
 * ACL GENPASS [<bits>]
 * ACL WHOAMI
 * ACL LOG [<count> | RESET]
 */
void aclCommand(client *c) {
    char *sub = c->argv[1]->ptr;
    if (!strcasecmp(sub,"setuser") && c->argc >= 3) {
        /* Initially redact all of the arguments to not leak any information
         * about the user. */
        for (int j = 2; j < c->argc; j++) {
            redactClientCommandArgument(c, j);
        }

        sds username = c->argv[2]->ptr;
        /* Check username validity. */
        if (ACLStringHasSpaces(username,sdslen(username))) {
            addReplyErrorFormat(c,
                "Usernames can't contain spaces or null characters");
            return;
        }

        int merged_argc = 0, invalid_idx = 0;
        sds *temp_argv = zmalloc(c->argc * sizeof(sds));
        for (int i = 3; i < c->argc; i++) temp_argv[i-3] = c->argv[i]->ptr;
        sds *acl_args = ACLMergeSelectorArguments(temp_argv, c->argc - 3,
            &merged_argc, &invalid_idx);
        zfree(temp_argv);

        if (!acl_args) {
            addReplyErrorFormat(c,

```

```

        "Unmatched parenthesis in acl selector starting "
        "at '%s'.", (char *) c->argv[invalid_idx]->ptr);
    return;
}

/* Create a temporary user to validate and stage all changes against
 * before applying to an existing user or creating a new user. If all
 * arguments are valid the user parameters will all be applied
together.
 * If there are any errors then none of the changes will be applied. */
user *tempu = ACLCreateUnlinkedUser();
user *u = ACLGetUserByName(username, sdslen(username));
if (u) ACLCopyUser(tempu, u);

for (int j = 0; j < merged_argc; j++) {
    if (ACLSetUser(tempu, acl_args[j], sdslen(acl_args[j])) != C_OK) {
        const char *errmsg = ACLSetUserStringError();
        addReplyErrorFormat(c,
            "Error in ACL SETUSER modifier '%s': %s",
            (char*)acl_args[j], errmsg);
        goto setuser_cleanup;
    }
}

/* Existing pub/sub clients authenticated with the user may need to be
 * disconnected if (some of) their channel permissions were revoked. */
if (u) ACLKillPubsubClientsIfNeeded(tempu, u);

/* Overwrite the user with the temporary user we modified above. */
if (!u) u = ACLCreateUser(username, sdslen(username));
serverAssert(u != NULL);
ACLCopyUser(u, tempu);
addReply(c, shared.ok);
setuser_cleanup:
    ACLFreeUser(tempu);
    for (int i = 0; i < merged_argc; i++) sdsfree(acl_args[i]);
    zfree(acl_args);
    return;
} else if (!strcasecmp(sub, "deluser") && c->argc >= 3) {
    int deleted = 0;
    for (int j = 2; j < c->argc; j++) {
        sds username = c->argv[j]->ptr;
        if (!strcmp(username, "default")) {
            addReplyError(c, "The 'default' user cannot be removed");
            return;
        }
    }
}

for (int j = 2; j < c->argc; j++) {
    sds username = c->argv[j]->ptr;
    user *u;

```

```

        if (raxRemove(Users,(unsigned char*)username,
                      sdslen(username),
                      (void**)&u))
        {
            ACLFreeUserAndKillClients(u);
            deleted++;
        }
    }
    addReplyLongLong(c,deleted);
} else if (!strcasecmp(sub,"getuser") && c->argc == 3) {
    user *u = ACLGetUserByName(c->argv[2]->ptr,sdslen(c->argv[2]->ptr));
    if (u == NULL) {
        addReplyNull(c);
        return;
    }

    void *ufields = addReplyDeferredLen(c);
    int fields = 3;

    /* Flags */
    addReplyBulkCString(c,"flags");
    void *deflen = addReplyDeferredLen(c);
    int numflags = 0;
    for (int j = 0; ACLUserFlags[j].flag; j++) {
        if (u->flags & ACLUserFlags[j].flag) {
            addReplyBulkCString(c,ACLUserFlags[j].name);
            numflags++;
        }
    }
    setDeferredSetLen(c,deflen,numflags);

    /* Passwords */
    addReplyBulkCString(c,"passwords");
    addReplyArrayLen(c,listLength(u->passwords));
    listIter li;
    listNode *ln;
    listRewind(u->passwords,&li);
    while((ln = listNext(&li))) {
        sds thispass = listNodeValue(ln);
        addReplyBulkCBuffer(c,thispass,sdslen(thispass));
    }

    /* Include the root selector at the top level for backwards
compatibility */
    fields += aclAddReplySelectorDescription(c, ACLUserGetRootSelector(u));

    /* Describe all of the selectors on this user, including duplicating
the root selector */
    addReplyBulkCString(c,"selectors");
    addReplyArrayLen(c, listLength(u->selectors) - 1);
    listRewind(u->selectors,&li);
    serverAssert(listNext(&li));

```



```

        while((ln = listNext(&li))) {
            void *slen = addReplyDeferredLen(c);
            int sfields = aclAddReplySelectorDescription(c, (aclSelector
*)listNodeValue(ln));
            setDeferredMapLen(c, slen, sfields);
        }
        setDeferredMapLen(c, ufields, fields);
    } else if ((!strcasecmp(sub,"list") || !strcasecmp(sub,"users")) &&
        c->argc == 2)
    {
        int justnames = !strcasecmp(sub,"users");
        addReplyArrayLen(c, raxSize(Users));
        raxIterator ri;
        raxStart(&ri, Users);
        raxSeek(&ri, "^", NULL, 0);
        while(raxNext(&ri)) {
            user *u = ri.data;
            if (justnames) {
                addReplyBulkCBuffer(c, u->name, sdslen(u->name));
            } else {
                /* Return information in the configuration file format. */
                sds config = sdsnew("user ");
                config = sdscatsds(config, u->name);
                config = sdscatlen(config, " ", 1);
                sds descr = ACLDescribeUser(u);
                config = sdscatsds(config, descr);
                sdsfree(descr);
                addReplyBulkSds(c, config);
            }
        }
        raxStop(&ri);
    } else if (!strcasecmp(sub,"whoami") && c->argc == 2) {
        if (c->user != NULL) {
            addReplyBulkCBuffer(c, c->user->name, sdslen(c->user->name));
        } else {
            addReplyNull(c);
        }
    } else if (server.acl_filename[0] == '\\0' &&
        (!strcasecmp(sub,"load") || !strcasecmp(sub,"save")))
    {
        addReplyError(c, "This Redis instance is not configured to use an ACL
file. You may want to specify users via the ACL SETUSER command and then issue
a CONFIG REWRITE (assuming you have a Redis configuration file set) in order to
store users in the Redis configuration.");
        return;
    } else if (!strcasecmp(sub,"load") && c->argc == 2) {
        sds errors = ACLLoadFromFile(server.acl_filename);
        if (errors == NULL) {
            addReply(c, shared.ok);
        } else {
            addReplyError(c, errors);
        }
    }

```

```

        sdsfree(errors);
    }
} else if (!strcasecmp(sub,"save") && c->argc == 2) {
    if (ACLSaveToFile(server.acl_filename) == C_OK) {
        addReply(c,shared.ok);
    } else {
        addReplyError(c,"There was an error trying to save the ACLs. "
            "Please check the server logs for more "
            "information");
    }
} else if (!strcasecmp(sub,"cat") && c->argc == 2) {
    void *dl = addReplyDeferredLen(c);
    int j;
    for (j = 0; ACLCommandCategories[j].flag != 0; j++)
        addReplyBulkCString(c,ACLCommandCategories[j].name);
    setDeferredArrayLen(c,dl,j);
} else if (!strcasecmp(sub,"cat") && c->argc == 3) {
    uint64_t cflag = ACLGetCommandCategoryFlagByName(c->argv[2]->ptr);
    if (cflag == 0) {
        addReplyErrorFormat(c, "Unknown category '%.128s'", (char*)c-
>argv[2]->ptr);
        return;
    }
    int arraylen = 0;
    void *dl = addReplyDeferredLen(c);
    aclCatWithFlags(c, server.orig_commands, cflag, &arraylen);
    setDeferredArrayLen(c,dl,arraylen);
} else if (!strcasecmp(sub,"genpass") && (c->argc == 2 || c->argc == 3)) {
    #define GENPASS_MAX_BITS 4096
    char pass[GENPASS_MAX_BITS/8*2]; /* Hex representation. */
    long bits = 256; /* By default generate 256 bits passwords. */

    if (c->argc == 3 && getLongFromObjectOrReply(c,c->argv[2],&bits,NULL)
        != C_OK) return;

    if (bits <= 0 || bits > GENPASS_MAX_BITS) {
        addReplyErrorFormat(c,
            "ACL GENPASS argument must be the number of "
            "bits for the output password, a positive number "
            "up to %d",GENPASS_MAX_BITS);
        return;
    }

    long chars = (bits+3)/4; /* Round to number of characters to emit. */
    getRandomHexChars(pass,chars);
    addReplyBulkCBuffer(c,pass,chars);
} else if (!strcasecmp(sub,"log") && (c->argc == 2 || c->argc == 3)) {
    long count = 10; /* Number of entries to emit by default. */

    /* Parse the only argument that LOG may have: it could be either
     * the number of entries the user wants to display, or alternatively

```

```

    /* the "RESET" command in order to flush the old entries. */
    if (c->argc == 3) {
        if (!strcasecmp(c->argv[2]->ptr,"reset")) {
            listSetFreeMethod(ACLLog,ACLFreeLogEntry);
            listEmpty(ACLLog);
            listSetFreeMethod(ACLLog,NULL);
            addReply(c,shared.ok);
            return;
        } else if (getLongFromObjectOrReply(c,c->argv[2],&count,NULL)
                    != C_OK)
        {
            return;
        }
        if (count < 0) count = 0;
    }

    /* Fix the count according to the number of entries we got. */
    if ((size_t)count > listLength(ACLLog))
        count = listLength(ACLLog);

    addReplyArrayLen(c,count);
    listIter li;
    listNode *ln;
    listRewind(ACLLog,&li);
    mstime_t now = mstime();
    while (count-- && (ln = listNext(&li)) != NULL) {
        ACLLogEntry *le = listNodeValue(ln);
        addReplyMapLen(c,7);
        addReplyBulkCString(c,"count");
        addReplyLongLong(c,le->count);

        addReplyBulkCString(c,"reason");
        char *reasonstr;
        switch(le->reason) {
            case ACL_DENIED_CMD: reasonstr="command"; break;
            case ACL_DENIED_KEY: reasonstr="key"; break;
            case ACL_DENIED_CHANNEL: reasonstr="channel"; break;
            case ACL_DENIED_AUTH: reasonstr="auth"; break;
            default: reasonstr="unknown";
        }
        addReplyBulkCString(c,reasonstr);

        addReplyBulkCString(c,"context");
        char *ctxstr;
        switch(le->context) {
            case ACL_LOG_CTX_TOPLEVEL: ctxstr="toplevel"; break;
            case ACL_LOG_CTX_MULTII: ctxstr="multi"; break;
            case ACL_LOG_CTX_LUA: ctxstr="lua"; break;
            case ACL_LOG_CTX_MODULE: ctxstr="module"; break;
            default: ctxstr="unknown";
        }
    }

```

```

        addReplyBulkCString(c,ctxstr);

        addReplyBulkCString(c,"object");
        addReplyBulkCBuffer(c,le->object,sdslen(le->object));
        addReplyBulkCString(c,"username");
        addReplyBulkCBuffer(c,le->username,sdslen(le->username));
        addReplyBulkCString(c,"age-seconds");
        double age = (double)(now - le->ctime)/1000;
        addReplyDouble(c,age);
        addReplyBulkCString(c,"client-info");
        addReplyBulkCBuffer(c,le->cinfo,sdslen(le->cinfo));
    }
} else if (!strcasecmp(sub,"dryrun") && c->argc >= 4) {
    struct redisCommand *cmd;
    user *u = ACLGetUserByName(c->argv[2]->ptr,sdslen(c->argv[2]->ptr));
    if (u == NULL) {
        addReplyErrorFormat(c, "User '%s' not found", (char *)c->argv[2]-
>ptr);
        return;
    }

    if ((cmd = lookupCommand(c->argv + 3, c->argc - 3)) == NULL) {
        addReplyErrorFormat(c, "Command '%s' not found", (char *)c-
>argv[3]->ptr);
        return;
    }

    if ((cmd->arity > 0 && cmd->arity != c->argc-3) ||
        (c->argc-3 < -cmd->arity))
    {
        addReplyErrorFormat(c,"wrong number of arguments for '%s' command",
cmd->fullname);
        return;
    }

    int idx;
    int result = ACLCheckAllUserCommandPerm(u, cmd, c->argv + 3, c->argc -
3, &idx);
    /* Notice that a variant of this code also exists on getAclErrorMessage
so
    * it also need to be updated on changed. */
    if (result != ACL_OK) {
        sds err = sdsempty();
        if (result == ACL_DENIED_CMD) {
            err = sdscatfmt(err, "This user has no permissions to run "
                "the '%s' command", cmd->fullname);
        } else if (result == ACL_DENIED_KEY) {
            err = sdscatfmt(err, "This user has no permissions to access "
                "the '%s' key", c->argv[idx + 3]->ptr);
        } else if (result == ACL_DENIED_CHANNEL) {
            err = sdscatfmt(err, "This user has no permissions to access "

```

```

        "the '%s' channel", c->argv[idx + 3]->ptr);
    } else {
        serverPanic("Invalid permission result");
    }
    addReplyBulkSds(c, err);
    return;
}

addReply(c, shared.ok);
} else if (c->argc == 2 && !strcasecmp(sub, "help")) {
    const char *help[] = {
"CAT [<category>]",
"    List all commands that belong to <category>, or all command categories",
"    when no category is specified.",
"DELUSER <username> [<username> ...]",
"    Delete a list of users.",
"DRYRUN <username> <command> [<arg> ...]",
"    Returns whether the user can execute the given command without executing",
"    the command.",
"GETUSER <username>",
"    Get the user's details.",
"GENPASS [<bits>]",
"    Generate a secure 256-bit user password. The optional `bits` argument",
"    can",
"    be used to specify a different size.",
"LIST",
"    Show users details in config file format.",
"LOAD",
"    Reload users from the ACL file.",
"LOG [<count> | RESET]",
"    Show the ACL log entries.",
"SAVE",
"    Save the current config to the ACL file.",
"SETUSER <username> <attribute> [<attribute> ...]",
"    Create or modify a user with the specified attributes.",
"USERS",
"    List all the registered usernames.",
"WHOAMI",
"    Return the current connection username.",
NULL
    };
    addReplyHelp(c, help);
} else {
    addReplySubcommandSyntaxError(c);
}
}

void addReplyCommandCategories(client *c, struct redisCommand *cmd) {
    int flagcount = 0;
    void *flaglen = addReplyDeferredLen(c);
    for (int j = 0; ACLCommandCategories[j].flag != 0; j++) {

```

```

        if (cmd->acl_categories & ACLCommandCategories[j].flag) {
            addReplyStatusFormat(c, "@%s", ACLCommandCategories[j].name);
            flagcount++;
        }
    }
    setDeferredSetLen(c, flaglen, flagcount);
}

/* AUTH <password>
 * AUTH <username> <password> (Redis >= 6.0 form)
 *
 * When the user is omitted it means that we are trying to authenticate
 * against the default user. */
void authCommand(client *c) {
    /* Only two or three argument forms are allowed. */
    if (c->argc > 3) {
        addReplyErrorObject(c, shared.syntaxerr);
        return;
    }
    /* Always redact the second argument */
    redactClientCommandArgument(c, 1);

    /* Handle the two different forms here. The form with two arguments
     * will just use "default" as username. */
    robj *username, *password;
    if (c->argc == 2) {
        /* Mimic the old behavior of giving an error for the two argument
         * form if no password is configured. */
        if (DefaultUser->flags & USER_FLAG_NOPASS) {
            addReplyError(c, "AUTH <password> called without any password "
                           "configured for the default user. Are you sure "
                           "your configuration is correct?");
            return;
        }

        username = shared.default_username;
        password = c->argv[1];
    } else {
        username = c->argv[1];
        password = c->argv[2];
        redactClientCommandArgument(c, 2);
    }

    if (ACLAuthenticateUser(c, username, password) == C_OK) {
        addReply(c, shared.ok);
    } else {
        addReplyError(c, "-WRONGPASS invalid username-password pair or user is
disabled.");
    }
}

```

```
/* Set the password for the "default" ACL user. This implements supports for
 * requirepass config, so passing in NULL will set the user to be nopass. */
void ACLUpdateDefaultUserPassword(sds password) {
    ACLSetUser(DefaultUser,"resetpass",-1);
    if (password) {
        sds aclop = sdscatlen(sdsnew(">"), password, sdslen(password));
        ACLSetUser(DefaultUser,aclop,sdslen(aclop));
        sdsfree(aclop);
    } else {
        ACLSetUser(DefaultUser,"nopass",-1);
    }
}
```

/adlist.c

[to top](#)

```

/* adlist.c - A generic doubly linked list implementation
 *
 * Copyright (c) 2006-2010, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include <stdlib.h>
#include "adlist.h"
#include "zmalloc.h"

/* Create a new list. The created list can be freed with
 * listRelease(), but private value of every node need to be freed
 * by the user before to call listRelease(), or by setting a free method using
 * listSetFreeMethod.
 *
 * On error, NULL is returned. Otherwise the pointer to the new list. */
list *listCreate(void)
{
    struct list *list;

    if ((list = zmalloc(sizeof(*list))) == NULL)
        return NULL;
    list->head = list->tail = NULL;
    list->len = 0;
    list->dup = NULL;

```



```

    list->free = NULL;
    list->match = NULL;
    return list;
}

/* Remove all the elements from the list without destroying the list itself. */
void listEmpty(list *list)
{
    unsigned long len;
    listNode *current, *next;

    current = list->head;
    len = list->len;
    while(len--) {
        next = current->next;
        if (list->free) list->free(current->value);
        zfree(current);
        current = next;
    }
    list->head = list->tail = NULL;
    list->len = 0;
}

/* Free the whole list.
 *
 * This function can't fail. */
void listRelease(list *list)
{
    listEmpty(list);
    zfree(list);
}

/* Add a new node to the list, to head, containing the specified 'value'
 * pointer as value.
 *
 * On error, NULL is returned and no operation is performed (i.e. the
 * list remains unaltered).
 * On success the 'list' pointer you pass to the function is returned. */
list *listAddNodeHead(list *list, void *value)
{
    listNode *node;

    if ((node = zmalloc(sizeof(*node))) == NULL)
        return NULL;
    node->value = value;
    if (list->len == 0) {
        list->head = list->tail = node;
        node->prev = node->next = NULL;
    } else {
        node->prev = NULL;
        node->next = list->head;
    }
}

```

```

        list->head->prev = node;
        list->head = node;
    }
    list->len++;
    return list;
}

/* Add a new node to the list, to tail, containing the specified 'value'
 * pointer as value.
 *
 * On error, NULL is returned and no operation is performed (i.e. the
 * list remains unaltered).
 * On success the 'list' pointer you pass to the function is returned. */
list *listAddNodeTail(list *list, void *value)
{
    listNode *node;

    if ((node = zmalloc(sizeof(*node))) == NULL)
        return NULL;
    node->value = value;
    if (list->len == 0) {
        list->head = list->tail = node;
        node->prev = node->next = NULL;
    } else {
        node->prev = list->tail;
        node->next = NULL;
        list->tail->next = node;
        list->tail = node;
    }
    list->len++;
    return list;
}

list *listInsertNode(list *list, listNode *old_node, void *value, int after) {
    listNode *node;

    if ((node = zmalloc(sizeof(*node))) == NULL)
        return NULL;
    node->value = value;
    if (after) {
        node->prev = old_node;
        node->next = old_node->next;
        if (list->tail == old_node) {
            list->tail = node;
        }
    } else {
        node->next = old_node;
        node->prev = old_node->prev;
        if (list->head == old_node) {
            list->head = node;
        }
    }
}

```

```

    }
    if (node->prev != NULL) {
        node->prev->next = node;
    }
    if (node->next != NULL) {
        node->next->prev = node;
    }
    list->len++;
    return list;
}

/* Remove the specified node from the specified list.
 * It's up to the caller to free the private value of the node.
 *
 * This function can't fail. */
void listDelNode(list *list, listNode *node)
{
    if (node->prev)
        node->prev->next = node->next;
    else
        list->head = node->next;
    if (node->next)
        node->next->prev = node->prev;
    else
        list->tail = node->prev;
    if (list->free) list->free(node->value);
    zfree(node);
    list->len--;
}

/* Returns a list iterator 'iter'. After the initialization every
 * call to listNext() will return the next element of the list.
 *
 * This function can't fail. */
listIter *listGetIterator(list *list, int direction)
{
    listIter *iter;

    if ((iter = zmalloc(sizeof(*iter))) == NULL) return NULL;
    if (direction == AL_START_HEAD)
        iter->next = list->head;
    else
        iter->next = list->tail;
    iter->direction = direction;
    return iter;
}

/* Release the iterator memory */
void listReleaseIterator(listIter *iter) {
    zfree(iter);
}

```

```

/* Create an iterator in the list private iterator structure */
void listRewind(list *list, listIter *li) {
    li->next = list->head;
    li->direction = AL_START_HEAD;
}

void listRewindTail(list *list, listIter *li) {
    li->next = list->tail;
    li->direction = AL_START_TAIL;
}

/* Return the next element of an iterator.
 * It's valid to remove the currently returned element using
 * listDelNode(), but not to remove other elements.
 *
 * The function returns a pointer to the next element of the list,
 * or NULL if there are no more elements, so the classical usage
 * pattern is:
 *
 * iter = listGetIterator(list,<direction>);
 * while ((node = listNext(iter)) != NULL) {
 *     doSomethingWith(listNodeValue(node));
 * }
 *
 */
listNode *listNext(listIter *iter)
{
    listNode *current = iter->next;

    if (current != NULL) {
        if (iter->direction == AL_START_HEAD)
            iter->next = current->next;
        else
            iter->next = current->prev;
    }
    return current;
}

/* Duplicate the whole list. On out of memory NULL is returned.
 * On success a copy of the original list is returned.
 *
 * The 'Dup' method set with listSetDupMethod() function is used
 * to copy the node value. Otherwise the same pointer value of
 * the original node is used as value of the copied node.
 *
 * The original list both on success or error is never modified. */
list *listDup(list *orig)
{
    list *copy;
    listIter iter;

```

```

listNode *node;

if ((copy = listCreate()) == NULL)
    return NULL;
copy->dup = orig->dup;
copy->free = orig->free;
copy->match = orig->match;
listRewind(orig, &iter);
while((node = listNext(&iter)) != NULL) {
    void *value;

    if (copy->dup) {
        value = copy->dup(node->value);
        if (value == NULL) {
            listRelease(copy);
            return NULL;
        }
    } else {
        value = node->value;
    }

    if (listAddNodeTail(copy, value) == NULL) {
        /* Free value if dup succeed but listAddNodeTail failed. */
        if (copy->free) copy->free(value);

        listRelease(copy);
        return NULL;
    }
}
return copy;
}

/* Search the list for a node matching a given key.
 * The match is performed using the 'match' method
 * set with listSetMatchMethod(). If no 'match' method
 * is set, the 'value' pointer of every node is directly
 * compared with the 'key' pointer.
 *
 * On success the first matching node pointer is returned
 * (search starts from head). If no matching node exists
 * NULL is returned. */
listNode *listSearchKey(list *list, void *key)
{
    listIter iter;
    listNode *node;

    listRewind(list, &iter);
    while((node = listNext(&iter)) != NULL) {
        if (list->match) {
            if (list->match(node->value, key)) {
                return node;
            }
        }
    }
    return NULL;
}

```

```

    }
    } else {
        if (key == node->value) {
            return node;
        }
    }
}
return NULL;
}

/* Return the element at the specified zero-based index
 * where 0 is the head, 1 is the element next to head
 * and so on. Negative integers are used in order to count
 * from the tail, -1 is the last element, -2 the penultimate
 * and so on. If the index is out of range NULL is returned. */
listNode *listIndex(list *list, long index) {
    listNode *n;

    if (index < 0) {
        index = (-index)-1;
        n = list->tail;
        while(index-- && n) n = n->prev;
    } else {
        n = list->head;
        while(index-- && n) n = n->next;
    }
    return n;
}

/* Rotate the list removing the tail node and inserting it to the head. */
void listRotateTailToHead(list *list) {
    if (listLength(list) <= 1) return;

    /* Detach current tail */
    listNode *tail = list->tail;
    list->tail = tail->prev;
    list->tail->next = NULL;
    /* Move it as head */
    list->head->prev = tail;
    tail->prev = NULL;
    tail->next = list->head;
    list->head = tail;
}

/* Rotate the list removing the head node and inserting it to the tail. */
void listRotateHeadToTail(list *list) {
    if (listLength(list) <= 1) return;

    listNode *head = list->head;
    /* Detach current head */
    list->head = head->next;

```

```

list->head->prev = NULL;
/* Move it as tail */
list->tail->next = head;
head->next = NULL;
head->prev = list->tail;
list->tail = head;
}

/* Add all the elements of the list 'o' at the end of the
 * list 'l'. The list 'other' remains empty but otherwise valid. */
void listJoin(list *l, list *o) {
    if (o->len == 0) return;

    o->head->prev = l->tail;

    if (l->tail)
        l->tail->next = o->head;
    else
        l->head = o->head;

    l->tail = o->tail;
    l->len += o->len;

    /* Setup other as an empty list. */
    o->head = o->tail = NULL;
    o->len = 0;
}

```

/adlist.h

[to top](#)

```

/* adlist.h - A generic doubly linked list implementation
 *
 * Copyright (c) 2006-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef __ADLIST_H__
#define __ADLIST_H__

/* Node, List, and Iterator are the only data structures used currently. */

typedef struct listNode {
    struct listNode *prev;
    struct listNode *next;
    void *value;
} listNode;

typedef struct listIter {
    listNode *next;
    int direction;
} listIter;

typedef struct list {
    listNode *head;
    listNode *tail;
    void *(*dup)(void *ptr);

```



```

    void (*free)(void *ptr);
    int (*match)(void *ptr, void *key);
    unsigned long len;
} list;

/* Functions implemented as macros */
#define listLength(l) ((l)->len)
#define listFirst(l) ((l)->head)
#define listLast(l) ((l)->tail)
#define listPrevNode(n) ((n)->prev)
#define listNextNode(n) ((n)->next)
#define listNodeValue(n) ((n)->value)

#define listSetDupMethod(l,m) ((l)->dup = (m))
#define listSetFreeMethod(l,m) ((l)->free = (m))
#define listSetMatchMethod(l,m) ((l)->match = (m))

#define listGetDupMethod(l) ((l)->dup)
#define listGetFreeMethod(l) ((l)->free)
#define listGetMatchMethod(l) ((l)->match)

/* Prototypes */
list *listCreate(void);
void listRelease(list *list);
void listEmpty(list *list);
list *listAddNodeHead(list *list, void *value);
list *listAddNodeTail(list *list, void *value);
list *listInsertNode(list *list, listNode *old_node, void *value, int after);
void listDelNode(list *list, listNode *node);
listIter *listGetIterator(list *list, int direction);
listNode *listNext(listIter *iter);
void listReleaseIterator(listIter *iter);
list *listDup(list *orig);
listNode *listSearchKey(list *list, void *key);
listNode *listIndex(list *list, long index);
void listRewind(list *list, listIter *li);
void listRewindTail(list *list, listIter *li);
void listRotateTailToHead(list *list);
void listRotateHeadToTail(list *list);
void listJoin(list *l, list *o);

/* Directions for iterators */
#define AL_START_HEAD 0
#define AL_START_TAIL 1

#endif /* __ADLIST_H__ */

```

/ae.c

[to top](#)

```

/* A simple event-driven programming library. Originally I wrote this code
 * for the Jim's event-loop (Jim is a Tcl interpreter) but later translated
 * it in form of a library for easy reuse.
 *
 * Copyright (c) 2006-2010, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "ae.h"
#include "anet.h"
#include "redisassert.h"

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <poll.h>
#include <string.h>
#include <time.h>
#include <errno.h>

#include "zmalloc.h"
#include "config.h"

/* Include the best multiplexing layer supported by this system.

```

```

    * The following should be ordered by performances, descending. */
#ifdef HAVE_EVPORT
#include "ae_evport.c"
#else
    #ifdef HAVE_EPOLL
    #include "ae_epoll.c"
    #else
        #ifdef HAVE_KQUEUE
        #include "ae_kqueue.c"
        #else
            #include "ae_select.c"
        #endif
    #endif
#endif

aeEventLoop *aeCreateEventLoop(int setsize) {
    aeEventLoop *eventLoop;
    int i;

    monotonicInit();    /* just in case the calling app didn't initialize */

    if ((eventLoop = zmalloc(sizeof(*eventLoop))) == NULL) goto err;
    eventLoop->events = zmalloc(sizeof(aeFileEvent)*setsize);
    eventLoop->fired = zmalloc(sizeof(aeFiredEvent)*setsize);
    if (eventLoop->events == NULL || eventLoop->fired == NULL) goto err;
    eventLoop->setsize = setsize;
    eventLoop->timeEventHead = NULL;
    eventLoop->timeEventNextId = 0;
    eventLoop->stop = 0;
    eventLoop->maxfd = -1;
    eventLoop->beforesleep = NULL;
    eventLoop->aftersleep = NULL;
    eventLoop->flags = 0;
    if (aeApiCreate(eventLoop) == -1) goto err;
    /* Events with mask == AE_NONE are not set. So let's initialize the
     * vector with it. */
    for (i = 0; i < setsize; i++)
        eventLoop->events[i].mask = AE_NONE;
    return eventLoop;

err:
    if (eventLoop) {
        zfree(eventLoop->events);
        zfree(eventLoop->fired);
        zfree(eventLoop);
    }
    return NULL;
}

/* Return the current set size. */

```

```

int aeGetSetSize(aeEventLoop *eventLoop) {
    return eventLoop->setsize;
}

/* Tells the next iteration/s of the event processing to set timeout of 0. */
void aeSetDontWait(aeEventLoop *eventLoop, int noWait) {
    if (noWait)
        eventLoop->flags |= AE_DONT_WAIT;
    else
        eventLoop->flags &= ~AE_DONT_WAIT;
}

/* Resize the maximum set size of the event loop.
 * If the requested set size is smaller than the current set size, but
 * there is already a file descriptor in use that is >= the requested
 * set size minus one, AE_ERR is returned and the operation is not
 * performed at all.
 *
 * Otherwise AE_OK is returned and the operation is successful. */
int aeResizeSetSize(aeEventLoop *eventLoop, int setsize) {
    int i;

    if (setsize == eventLoop->setsize) return AE_OK;
    if (eventLoop->maxfd >= setsize) return AE_ERR;
    if (aeApiResize(eventLoop, setsize) == -1) return AE_ERR;

    eventLoop->events = zrealloc(eventLoop->events,
    sizeof(aeFileEvent)*setsize);
    eventLoop->fired = zrealloc(eventLoop->fired, sizeof(aeFiredEvent)*setsize);
    eventLoop->setsize = setsize;

    /* Make sure that if we created new slots, they are initialized with
     * an AE_NONE mask. */
    for (i = eventLoop->maxfd+1; i < setsize; i++)
        eventLoop->events[i].mask = AE_NONE;
    return AE_OK;
}

void aeDeleteEventLoop(aeEventLoop *eventLoop) {
    aeApiFree(eventLoop);
    zfree(eventLoop->events);
    zfree(eventLoop->fired);

    /* Free the time events list. */
    aeTimeEvent *next_te, *te = eventLoop->timeEventHead;
    while (te) {
        next_te = te->next;
        zfree(te);
        te = next_te;
    }
    zfree(eventLoop);
}

```

```

}

void aeStop(aeEventLoop *eventLoop) {
    eventLoop->stop = 1;
}

int aeCreateFileEvent(aeEventLoop *eventLoop, int fd, int mask,
    aeFileProc *proc, void *clientData)
{
    if (fd >= eventLoop->setsize) {
        errno = ERANGE;
        return AE_ERR;
    }
    aeFileEvent *fe = &eventLoop->events[fd];

    if (aeApiAddEvent(eventLoop, fd, mask) == -1)
        return AE_ERR;
    fe->mask |= mask;
    if (mask & AE_READABLE) fe->rfileProc = proc;
    if (mask & AE_WRITABLE) fe->wfileProc = proc;
    fe->clientData = clientData;
    if (fd > eventLoop->maxfd)
        eventLoop->maxfd = fd;
    return AE_OK;
}

void aeDeleteFileEvent(aeEventLoop *eventLoop, int fd, int mask)
{
    if (fd >= eventLoop->setsize) return;
    aeFileEvent *fe = &eventLoop->events[fd];
    if (fe->mask == AE_NONE) return;

    /* We want to always remove AE_BARRIER if set when AE_WRITABLE
     * is removed. */
    if (mask & AE_WRITABLE) mask |= AE_BARRIER;

    aeApiDelEvent(eventLoop, fd, mask);
    fe->mask = fe->mask & (~mask);
    if (fd == eventLoop->maxfd && fe->mask == AE_NONE) {
        /* Update the max fd */
        int j;

        for (j = eventLoop->maxfd-1; j >= 0; j--)
            if (eventLoop->events[j].mask != AE_NONE) break;
        eventLoop->maxfd = j;
    }
}

void *aeGetFileClientData(aeEventLoop *eventLoop, int fd) {
    if (fd >= eventLoop->setsize) return NULL;
    aeFileEvent *fe = &eventLoop->events[fd];

```

```

    if (fe->mask == AE_NONE) return NULL;

    return fe->clientData;
}

int aeGetFileEvents(aeEventLoop *eventLoop, int fd) {
    if (fd >= eventLoop->setsize) return 0;
    aeFileEvent *fe = &eventLoop->events[fd];

    return fe->mask;
}

long long aeCreateTimeEvent(aeEventLoop *eventLoop, long long milliseconds,
    aeTimeProc *proc, void *clientData,
    aeEventFinalizerProc *finalizerProc)
{
    long long id = eventLoop->timeEventNextId++;
    aeTimeEvent *te;

    te = zmalloc(sizeof(*te));
    if (te == NULL) return AE_ERR;
    te->id = id;
    te->when = getMonotonicUs() + milliseconds * 1000;
    te->timeProc = proc;
    te->finalizerProc = finalizerProc;
    te->clientData = clientData;
    te->prev = NULL;
    te->next = eventLoop->timeEventHead;
    te->refcount = 0;
    if (te->next)
        te->next->prev = te;
    eventLoop->timeEventHead = te;
    return id;
}

int aeDeleteTimeEvent(aeEventLoop *eventLoop, long long id)
{
    aeTimeEvent *te = eventLoop->timeEventHead;
    while(te) {
        if (te->id == id) {
            te->id = AE_DELETED_EVENT_ID;
            return AE_OK;
        }
        te = te->next;
    }
    return AE_ERR; /* NO event with the specified ID found */
}

/* How many microseconds until the first timer should fire.
 * If there are no timers, -1 is returned.
 */

```

```

* Note that's  $O(N)$  since time events are unsorted.
* Possible optimizations (not needed by Redis so far, but...):
* 1) Insert the event in order, so that the nearest is just the head.
*    Much better but still insertion or deletion of timers is  $O(N)$ .
* 2) Use a skiplist to have this operation as  $O(1)$  and insertion as  $O(\log(N))$ .
*/
static int64_t usUntilEarliestTimer(aeEventLoop *eventLoop) {
    aeTimeEvent *te = eventLoop->timeEventHead;
    if (te == NULL) return -1;

    aeTimeEvent *earliest = NULL;
    while (te) {
        if (!earliest || te->when < earliest->when)
            earliest = te;
        te = te->next;
    }

    monotime now = getMonotonicUs();
    return (now >= earliest->when) ? 0 : earliest->when - now;
}

/* Process time events */
static int processTimeEvents(aeEventLoop *eventLoop) {
    int processed = 0;
    aeTimeEvent *te;
    long long maxId;

    te = eventLoop->timeEventHead;
    maxId = eventLoop->timeEventNextId-1;
    monotime now = getMonotonicUs();
    while(te) {
        long long id;

        /* Remove events scheduled for deletion. */
        if (te->id == AE_DELETED_EVENT_ID) {
            aeTimeEvent *next = te->next;
            /* If a reference exists for this timer event,
             * don't free it. This is currently incremented
             * for recursive timerProc calls */
            if (te->refcount) {
                te = next;
                continue;
            }
            if (te->prev)
                te->prev->next = te->next;
            else
                eventLoop->timeEventHead = te->next;
            if (te->next)
                te->next->prev = te->prev;
            if (te->finalizerProc) {
                te->finalizerProc(eventLoop, te->clientData);
            }
        }
    }
}

```

```

        now = getMonotonicUs();
    }
    zfree(te);
    te = next;
    continue;
}

/* Make sure we don't process time events created by time events in
 * this iteration. Note that this check is currently useless: we always
 * add new timers on the head, however if we change the implementation
 * detail, this check may be useful again: we keep it here for future
 * defense. */
if (te->id > maxId) {
    te = te->next;
    continue;
}

if (te->when <= now) {
    int retval;

    id = te->id;
    te->refcount++;
    retval = te->timeProc(eventLoop, id, te->clientData);
    te->refcount--;
    processed++;
    now = getMonotonicUs();
    if (retval != AE_NOMORE) {
        te->when = now + retval * 1000;
    } else {
        te->id = AE_DELETED_EVENT_ID;
    }
}
te = te->next;
}
return processed;
}

/* Process every pending time event, then every pending file event
 * (that may be registered by time event callbacks just processed).
 * Without special flags the function sleeps until some file event
 * fires, or when the next time event occurs (if any).
 *
 * If flags is 0, the function does nothing and returns.
 * if flags has AE_ALL_EVENTS set, all the kind of events are processed.
 * if flags has AE_FILE_EVENTS set, file events are processed.
 * if flags has AE_TIME_EVENTS set, time events are processed.
 * if flags has AE_DONT_WAIT set, the function returns ASAP once all
 * the events that can be handled without a wait are processed.
 * if flags has AE_CALL_AFTER_SLEEP set, the aftersleep callback is called.
 * if flags has AE_CALL_BEFORE_SLEEP set, the before sleep callback is called.
 */

```



```

    * The function returns the number of events processed. */
int aeProcessEvents(aeEventLoop *eventLoop, int flags)
{
    int processed = 0, numevents;

    /* Nothing to do? return ASAP */
    if (!(flags & AE_TIME_EVENTS) && !(flags & AE_FILE_EVENTS)) return 0;

    /* Note that we want to call select() even if there are no
     * file events to process as long as we want to process time
     * events, in order to sleep until the next time event is ready
     * to fire. */
    if (eventLoop->maxfd != -1 ||
        ((flags & AE_TIME_EVENTS) && !(flags & AE_DONT_WAIT))) {
        int j;
        struct timeval tv, *tvp;
        int64_t usUntilTimer = -1;

        if (flags & AE_TIME_EVENTS && !(flags & AE_DONT_WAIT))
            usUntilTimer = usUntilEarliestTimer(eventLoop);

        if (usUntilTimer >= 0) {
            tv.tv_sec = usUntilTimer / 1000000;
            tv.tv_usec = usUntilTimer % 1000000;
            tvp = &tv;
        } else {
            /* If we have to check for events but need to return
             * ASAP because of AE_DONT_WAIT we need to set the timeout
             * to zero */
            if (flags & AE_DONT_WAIT) {
                tv.tv_sec = tv.tv_usec = 0;
                tvp = &tv;
            } else {
                /* Otherwise we can block */
                tvp = NULL; /* wait forever */
            }
        }
    }

    if (eventLoop->flags & AE_DONT_WAIT) {
        tv.tv_sec = tv.tv_usec = 0;
        tvp = &tv;
    }

    if (eventLoop->beforesleep != NULL && flags & AE_CALL_BEFORE_SLEEP)
        eventLoop->beforesleep(eventLoop);

    /* Call the multiplexing API, will return only on timeout or when
     * some event fires. */
    numevents = aeApiPoll(eventLoop, tvp);

    /* After sleep callback. */

```

```

if (eventLoop->aftersleep != NULL && flags & AE_CALL_AFTER_SLEEP)
    eventLoop->aftersleep(eventLoop);

for (j = 0; j < numevents; j++) {
    int fd = eventLoop->fired[j].fd;
    aeFileEvent *fe = &eventLoop->events[fd];
    int mask = eventLoop->fired[j].mask;
    int fired = 0; /* Number of events fired for current fd. */

    /* Normally we execute the readable event first, and the writable
     * event later. This is useful as sometimes we may be able
     * to serve the reply of a query immediately after processing the
     * query.
     *
     * However if AE_BARRIER is set in the mask, our application is
     * asking us to do the reverse: never fire the writable event
     * after the readable. In such a case, we invert the calls.
     * This is useful when, for instance, we want to do things
     * in the beforeSleep() hook, like fsyncing a file to disk,
     * before replying to a client. */
    int invert = fe->mask & AE_BARRIER;

    /* Note the "fe->mask & mask & ..." code: maybe an already
     * processed event removed an element that fired and we still
     * didn't process, so we check if the event is still valid.
     *
     * Fire the readable event if the call sequence is not
     * inverted. */
    if (!invert && fe->mask & mask & AE_READABLE) {
        fe->rfileProc(eventLoop,fd,fe->clientData,mask);
        fired++;
        fe = &eventLoop->events[fd]; /* Refresh in case of resize. */
    }

    /* Fire the writable event. */
    if (fe->mask & mask & AE_WRITABLE) {
        if (!fired || fe->wfileProc != fe->rfileProc) {
            fe->wfileProc(eventLoop,fd,fe->clientData,mask);
            fired++;
        }
    }
}

/* If we have to invert the call, fire the readable event now
 * after the writable one. */
if (invert) {
    fe = &eventLoop->events[fd]; /* Refresh in case of resize. */
    if ((fe->mask & mask & AE_READABLE) &&
        (!fired || fe->wfileProc != fe->rfileProc))
    {
        fe->rfileProc(eventLoop,fd,fe->clientData,mask);
        fired++;
    }
}

```

```

        }
    }

    processed++;
}
}

/* Check time events */
if (flags & AE_TIME_EVENTS)
    processed += processTimeEvents(eventLoop);

return processed; /* return the number of processed file/time events */
}

/* Wait for milliseconds until the given file descriptor becomes
 * writable/readable/exception */
int aeWait(int fd, int mask, long long milliseconds) {
    struct pollfd pfd;
    int retmask = 0, retval;

    memset(&pfd, 0, sizeof(pfd));
    pfd.fd = fd;
    if (mask & AE_READABLE) pfd.events |= POLLIN;
    if (mask & AE_WRITABLE) pfd.events |= POLLOUT;

    if ((retval = poll(&pfd, 1, milliseconds)) == 1) {
        if (pfd.revents & POLLIN) retmask |= AE_READABLE;
        if (pfd.revents & POLLOUT) retmask |= AE_WRITABLE;
        if (pfd.revents & POLLERR) retmask |= AE_READABLE;
        if (pfd.revents & POLLHUP) retmask |= AE_READABLE;
        return retmask;
    } else {
        return retval;
    }
}

void aeMain(aeEventLoop *eventLoop) {
    eventLoop->stop = 0;
    while (!eventLoop->stop) {
        aeProcessEvents(eventLoop, AE_ALL_EVENTS |
                        AE_CALL_BEFORE_SLEEP |
                        AE_CALL_AFTER_SLEEP);
    }
}

char *aeGetApiName(void) {
    return aeApiName();
}

void aeSetBeforeSleepProc(aeEventLoop *eventLoop, aeBeforeSleepProc
*beforesleep) {
    eventLoop->beforesleep = beforesleep;
}

```

```
}  
  
void aeSetAfterSleepProc(aeEventLoop *eventLoop, aeBeforeSleepProc *aftersleep)  
{  
    eventLoop->aftersleep = aftersleep;  
}
```

/ae.h

[to top](#)

```

/* A simple event-driven programming library. Originally I wrote this code
 * for the Jim's event-loop (Jim is a Tcl interpreter) but later translated
 * it in form of a library for easy reuse.
 *
 * Copyright (c) 2006-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef __AE_H__
#define __AE_H__

#include "monotonic.h"

#define AE_OK 0
#define AE_ERR -1

#define AE_NONE 0          /* No events registered. */
#define AE_READABLE 1     /* Fire when descriptor is readable. */
#define AE_WRITABLE 2     /* Fire when descriptor is writable. */
#define AE_BARRIER 4     /* With WRITABLE, never fire the event if the
                             READABLE event already fired in the same event
                             loop iteration. Useful when you want to persist
                             things to disk before sending replies, and want
                             to do that in a group fashion. */

#define AE_FILE_EVENTS (1<0)

```

```

#define AE_TIME_EVENTS (1<<1)
#define AE_ALL_EVENTS (AE_FILE_EVENTS|AE_TIME_EVENTS)
#define AE_DONT_WAIT (1<<2)
#define AE_CALL_BEFORE_SLEEP (1<<3)
#define AE_CALL_AFTER_SLEEP (1<<4)

#define AE_NOMORE -1
#define AE_DELETED_EVENT_ID -1

/* Macros */
#define AE_NOTUSED(V) ((void) V)

struct aeEventLoop;

/* Types and data structures */
typedef void aeFileProc(struct aeEventLoop *eventLoop, int fd, void
*clientData, int mask);
typedef int aeTimeProc(struct aeEventLoop *eventLoop, long long id, void
*clientData);
typedef void aeEventFinalizerProc(struct aeEventLoop *eventLoop, void
*clientData);
typedef void aeBeforeSleepProc(struct aeEventLoop *eventLoop);

/* File event structure */
typedef struct aeFileEvent {
    int mask; /* one of AE_(READABLE|WRITABLE|BARRIER) */
    aeFileProc *rfileProc;
    aeFileProc *wfileProc;
    void *clientData;
} aeFileEvent;

/* Time event structure */
typedef struct aeTimeEvent {
    long long id; /* time event identifier. */
    monotime when;
    aeTimeProc *timeProc;
    aeEventFinalizerProc *finalizerProc;
    void *clientData;
    struct aeTimeEvent *prev;
    struct aeTimeEvent *next;
    int refcount; /* refcount to prevent timer events from being
        * freed in recursive time event calls. */
} aeTimeEvent;

/* A fired event */
typedef struct aeFiredEvent {
    int fd;
    int mask;
} aeFiredEvent;

/* State of an event based program */

```

```

typedef struct aeEventLoop {
    int maxfd; /* highest file descriptor currently registered */
    int setsize; /* max number of file descriptors tracked */
    long long timeEventNextId;
    aeFileEvent *events; /* Registered events */
    aeFiredEvent *fired; /* Fired events */
    aeTimeEvent *timeEventHead;
    int stop;
    void *apidata; /* This is used for polling API specific data */
    aeBeforeSleepProc *beforesleep;
    aeBeforeSleepProc *aftersleep;
    int flags;
} aeEventLoop;

/* Prototypes */
aeEventLoop *aeCreateEventLoop(int setsize);
void aeDeleteEventLoop(aeEventLoop *eventLoop);
void aeStop(aeEventLoop *eventLoop);
int aeCreateFileEvent(aeEventLoop *eventLoop, int fd, int mask,
    aeFileProc *proc, void *clientData);
void aeDeleteFileEvent(aeEventLoop *eventLoop, int fd, int mask);
int aeGetFileEvents(aeEventLoop *eventLoop, int fd);
void *aeGetFileClientData(aeEventLoop *eventLoop, int fd);
long long aeCreateTimeEvent(aeEventLoop *eventLoop, long long milliseconds,
    aeTimeProc *proc, void *clientData,
    aeEventFinalizerProc *finalizerProc);
int aeDeleteTimeEvent(aeEventLoop *eventLoop, long long id);
int aeProcessEvents(aeEventLoop *eventLoop, int flags);
int aeWait(int fd, int mask, long long milliseconds);
void aeMain(aeEventLoop *eventLoop);
char *aeGetApiName(void);
void aeSetBeforeSleepProc(aeEventLoop *eventLoop, aeBeforeSleepProc
*beforesleep);
void aeSetAfterSleepProc(aeEventLoop *eventLoop, aeBeforeSleepProc
*aftersleep);
int aeGetSetSize(aeEventLoop *eventLoop);
int aeResizeSetSize(aeEventLoop *eventLoop, int setsize);
void aeSetDontWait(aeEventLoop *eventLoop, int noWait);

#endif

```

/ae_epoll.c

[to top](#)

```

/* Linux epoll(2) based ae.c module
 *
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include <sys/epoll.h>

typedef struct aeApiState {
    int epfd;
    struct epoll_event *events;
} aeApiState;

static int aeApiCreate(aeEventLoop *eventLoop) {
    aeApiState *state = zmalloc(sizeof(aeApiState));

    if (!state) return -1;
    state->events = zmalloc(sizeof(struct epoll_event)*eventLoop->setsize);
    if (!state->events) {
        zfree(state);
        return -1;
    }
    state->epfd = epoll_create(1024); /* 1024 is just a hint for the kernel */
    if (state->epfd == -1) {
        zfree(state->events);

```



```

        zfree(state);
        return -1;
    }
    anetCloexec(state->epfd);
    eventLoop->apidata = state;
    return 0;
}

static int aeApiResize(aeEventLoop *eventLoop, int setsize) {
    aeApiState *state = eventLoop->apidata;

    state->events = zrealloc(state->events, sizeof(struct
    epoll_event)*setsize);
    return 0;
}

static void aeApiFree(aeEventLoop *eventLoop) {
    aeApiState *state = eventLoop->apidata;

    close(state->epfd);
    zfree(state->events);
    zfree(state);
}

static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;
    struct epoll_event ee = {0}; /* avoid valgrind warning */
    /* If the fd was already monitored for some event, we need a MOD
     * operation. Otherwise we need an ADD operation. */
    int op = eventLoop->events[fd].mask == AE_NONE ?
        EPOLL_CTL_ADD : EPOLL_CTL_MOD;

    ee.events = 0;
    mask |= eventLoop->events[fd].mask; /* Merge old events */
    if (mask & AE_READABLE) ee.events |= EPOLLIN;
    if (mask & AE_WRITABLE) ee.events |= EPOLLOUT;
    ee.data.fd = fd;
    if (epoll_ctl(state->epfd,op,fd,&ee) == -1) return -1;
    return 0;
}

static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int delmask) {
    aeApiState *state = eventLoop->apidata;
    struct epoll_event ee = {0}; /* avoid valgrind warning */
    int mask = eventLoop->events[fd].mask & (~delmask);

    ee.events = 0;
    if (mask & AE_READABLE) ee.events |= EPOLLIN;
    if (mask & AE_WRITABLE) ee.events |= EPOLLOUT;
    ee.data.fd = fd;
    if (mask != AE_NONE) {

```

```

        epoll_ctl(state->epfd, EPOLL_CTL_MOD, fd, &ee);
    } else {
        /* Note, Kernel < 2.6.9 requires a non null event pointer even for
         * EPOLL_CTL_DEL. */
        epoll_ctl(state->epfd, EPOLL_CTL_DEL, fd, &ee);
    }
}

static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp) {
    aeApiState *state = eventLoop->apidata;
    int retval, numevents = 0;

    retval = epoll_wait(state->epfd, state->events, eventLoop->setsize,
        tvp ? (tvp->tv_sec*1000 + (tvp->tv_usec + 999)/1000) : -1);
    if (retval > 0) {
        int j;

        numevents = retval;
        for (j = 0; j < numevents; j++) {
            int mask = 0;
            struct epoll_event *e = state->events+j;

            if (e->events & EPOLLIN) mask |= AE_READABLE;
            if (e->events & EPOLLOUT) mask |= AE_WRITABLE;
            if (e->events & EPOLLERR) mask |= AE_WRITABLE|AE_READABLE;
            if (e->events & EPOLLHUP) mask |= AE_WRITABLE|AE_READABLE;
            eventLoop->fired[j].fd = e->data.fd;
            eventLoop->fired[j].mask = mask;
        }
    } else if (retval == -1 && errno != EINTR) {
        panic("aeApiPoll: epoll_wait, %s", strerror(errno));
    }

    return numevents;
}

static char *aeApiName(void) {
    return "epoll";
}

```

/ae_evport.c

[to top](#)

```

/* ae.c module for illumos event ports.
 *
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include <errno.h>
#include <port.h>
#include <poll.h>

#include <sys/types.h>
#include <sys/time.h>

#include <stdio.h>

static int evport_debug = 0;

/*
 * This file implements the ae API using event ports, present on Solaris-based
 * systems since Solaris 10. Using the event port interface, we associate file
 * descriptors with the port. Each association also includes the set of
poll(2)
 * events that the consumer is interested in (e.g., POLLIN and POLLOUT).
 *
 * There's one tricky piece to this implementation: when we return events via
 * aeApiPoll, the corresponding file descriptors become dissociated from the

```

```
* port. This is necessary because poll events are level-triggered, so if the
* fd didn't become dissociated, it would immediately fire another event since
* the underlying state hasn't changed yet. We must re-associate the file
* descriptor, but only after we know that our caller has actually read from
it.
```

```
* The ae API does not tell us exactly when that happens, but we do know that
* it must happen by the time aeApiPoll is called again. Our solution is to
* keep track of the last fds returned by aeApiPoll and re-associate them next
* time aeApiPoll is invoked.
```

```
*
* To summarize, in this module, each fd association is EITHER (a) represented
* only via the in-kernel association OR (b) represented by pending_fds and
* pending_masks. (b) is only true for the last fds we returned from
aeApiPoll,
```

```
* and only until we enter aeApiPoll again (at which point we restore the
* in-kernel association).
```

```
*/
```

```
#define MAX_EVENT_BATCHSZ 512
```

```
typedef struct aeApiState {
    int      portfd;                /* event port */
    uint_t   npending;              /* # of pending fds */
    int      pending_fds[MAX_EVENT_BATCHSZ]; /* pending fds */
    int      pending_masks[MAX_EVENT_BATCHSZ]; /* pending fds' masks */
} aeApiState;
```

```
static int aeApiCreate(aeEventLoop *eventLoop) {
    int i;
    aeApiState *state = zmalloc(sizeof(aeApiState));
    if (!state) return -1;

    state->portfd = port_create();
    if (state->portfd == -1) {
        zfree(state);
        return -1;
    }
    anetCloexec(state->portfd);

    state->npending = 0;

    for (i = 0; i < MAX_EVENT_BATCHSZ; i++) {
        state->pending_fds[i] = -1;
        state->pending_masks[i] = AE_NONE;
    }

    eventLoop->apidata = state;
    return 0;
}
```

```
static int aeApiResize(aeEventLoop *eventLoop, int setsize) {
    (void) eventLoop;
```

```

    (void) setsize;
    /* Nothing to resize here. */
    return 0;
}

static void aeApiFree(aeEventLoop *eventLoop) {
    aeApiState *state = eventLoop->apidata;

    close(state->portfd);
    zfree(state);
}

static int aeApiLookupPending(aeApiState *state, int fd) {
    uint_t i;

    for (i = 0; i < state->npending; i++) {
        if (state->pending_fds[i] == fd)
            return (i);
    }

    return (-1);
}

/*
 * Helper function to invoke port_associate for the given fd and mask.
 */
static int aeApiAssociate(const char *where, int portfd, int fd, int mask) {
    int events = 0;
    int rv, err;

    if (mask & AE_READABLE)
        events |= POLLIN;
    if (mask & AE_WRITABLE)
        events |= POLLOUT;

    if (evport_debug)
        fprintf(stderr, "%s: port_associate(%d, 0x%x) = ", where, fd, events);

    rv = port_associate(portfd, PORT_SOURCE_FD, fd, events,
        (void *) (uintptr_t) mask);
    err = errno;

    if (evport_debug)
        fprintf(stderr, "%d (%s)\n", rv, rv == 0 ? "no error" : strerror(err));

    if (rv == -1) {
        fprintf(stderr, "%s: port_associate: %s\n", where, strerror(err));

        if (err == EAGAIN)
            fprintf(stderr, "aeApiAssociate: event port limit exceeded.");
    }
}

```

```

    return rv;
}

static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;
    int fullmask, pfd;

    if (evport_debug)
        fprintf(stderr, "aeApiAddEvent: fd %d mask 0x%x\n", fd, mask);

    /*
     * Since port_associate's "events" argument replaces any existing events,
we
     * must be sure to include whatever events are already associated when
     * we call port_associate() again.
     */
    fullmask = mask | eventLoop->events[fd].mask;
    pfd = aeApiLookupPending(state, fd);

    if (pfd != -1) {
        /*
         * This fd was recently returned from aeApiPoll. It should be safe to
         * assume that the consumer has processed that poll event, but we play
         * it safer by simply updating pending_mask. The fd will be
         * re-associated as usual when aeApiPoll is called again.
         */
        if (evport_debug)
            fprintf(stderr, "aeApiAddEvent: adding to pending fd %d\n", fd);
        state->pending_masks[pfd] |= fullmask;
        return 0;
    }

    return (aeApiAssociate("aeApiAddEvent", state->portfd, fd, fullmask));
}

static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;
    int fullmask, pfd;

    if (evport_debug)
        fprintf(stderr, "del fd %d mask 0x%x\n", fd, mask);

    pfd = aeApiLookupPending(state, fd);

    if (pfd != -1) {
        if (evport_debug)
            fprintf(stderr, "deleting event from pending fd %d\n", fd);

        /*
         * This fd was just returned from aeApiPoll, so it's not currently

```

```

    * associated with the port. All we need to do is update
    * pending_mask appropriately.
    */
state->pending_masks[pfd] &= ~mask;

if (state->pending_masks[pfd] == AE_NONE)
    state->pending_fds[pfd] = -1;

return;
}

/*
 * The fd is currently associated with the port. Like with the add case
 * above, we must look at the full mask for the file descriptor before
 * updating that association. We don't have a good way of knowing what the
 * events are without looking into the eventLoop state directly. We rely
on
 * the fact that our caller has already updated the mask in the eventLoop.
 */

fullmask = eventLoop->events[fd].mask;
if (fullmask == AE_NONE) {
    /*
     * We're removing *all* events, so use port_dissociate to remove the
     * association completely. Failure here indicates a bug.
     */
    if (evport_debug)
        fprintf(stderr, "aeApiDelEvent: port_dissociate(%d)\n", fd);

    if (port_dissociate(state->portfd, PORT_SOURCE_FD, fd) != 0) {
        perror("aeApiDelEvent: port_dissociate");
        abort(); /* will not return */
    }
} else if (aeApiAssociate("aeApiDelEvent", state->portfd, fd,
fullmask) != 0) {
    /*
     * ENOMEM is a potentially transient condition, but the kernel won't
     * generally return it unless things are really bad. EAGAIN indicates
     * we've reached a resource limit, for which it doesn't make sense to
     * retry (counter-intuitively). All other errors indicate a bug. In
any
     * of these cases, the best we can do is to abort.
     */
    abort(); /* will not return */
}
}

static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp) {
    aeApiState *state = eventLoop->apidata;
    struct timespec timeout, *tsp;
    uint_t mask, i;

```

```

uint_t nevents;
port_event_t event[MAX_EVENT_BATCHSZ];

/*
 * If we've returned fd events before, we must re-associate them with the
 * port now, before calling port_get(). See the block comment at the top
of
 * this file for an explanation of why.
 */
for (i = 0; i < state->npending; i++) {
    if (state->pending_fds[i] == -1)
        /* This fd has since been deleted. */
        continue;

    if (aeApiAssociate("aeApiPoll", state->portfd,
        state->pending_fds[i], state->pending_masks[i]) != 0) {
        /* See aeApiDelEvent for why this case is fatal. */
        abort();
    }

    state->pending_masks[i] = AE_NONE;
    state->pending_fds[i] = -1;
}

state->npending = 0;

if (tvp != NULL) {
    timeout.tv_sec = tvp->tv_sec;
    timeout.tv_nsec = tvp->tv_usec * 1000;
    tsp = &timeout;
} else {
    tsp = NULL;
}

/*
 * port_getn can return with errno == ETIME having returned some events
(!).
 * So if we get ETIME, we check nevents, too.
 */
nevents = 1;
if (port_getn(state->portfd, event, MAX_EVENT_BATCHSZ, &nevents,
    tsp) == -1 && (errno != ETIME || nevents == 0)) {
    if (errno == ETIME || errno == EINTR)
        return 0;

    /* Any other error indicates a bug. */
    panic("aeApiPoll: port_getn, %s", strerror(errno));
}

state->npending = nevents;

```



```

    for (i = 0; i < nevents; i++) {
        mask = 0;
        if (event[i].portev_events & POLLIN)
            mask |= AE_READABLE;
        if (event[i].portev_events & POLLOUT)
            mask |= AE_WRITABLE;

        eventLoop->fired[i].fd = event[i].portev_object;
        eventLoop->fired[i].mask = mask;

        if (evport_debug)
            fprintf(stderr, "aeApiPoll: fd %d mask 0x%x\n",
                (int)event[i].portev_object, mask);

        state->pending_fds[i] = event[i].portev_object;
        state->pending_masks[i] = (uintptr_t)event[i].portev_user;
    }

    return nevents;
}

static char *aeApiName(void) {
    return "evport";
}

```

/ae_kqueue.c

[to top](#)

```

/* Kqueue(2)-based ae.c module
 *
 * Copyright (C) 2009 Harish Mallipeddi - harish.mallipeddi@gmail.com
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>

typedef struct aeApiState {
    int kqfd;
    struct kevent *events;

    /* Events mask for merge read and write event.
     * To reduce memory consumption, we use 2 bits to store the mask
     * of an event, so that 1 byte will store the mask of 4 events. */
    char *eventsMask;
} aeApiState;

#define EVENT_MASK_MALLOC_SIZE(sz) (((sz) + 3) / 4)
#define EVENT_MASK_OFFSET(fd) ((fd) % 4 * 2)
#define EVENT_MASK_ENCODE(fd, mask) ((mask) & 0x3) << EVENT_MASK_OFFSET(fd))

static inline int getEventMask(const char *eventsMask, int fd) {

```

```

    return (eventsMask[fd/4] >> EVENT_MASK_OFFSET(fd)) & 0x3;
}

static inline void addEventMask(char *eventsMask, int fd, int mask) {
    eventsMask[fd/4] |= EVENT_MASK_ENCODE(fd, mask);
}

static inline void resetEventMask(char *eventsMask, int fd) {
    eventsMask[fd/4] &= ~EVENT_MASK_ENCODE(fd, 0x3);
}

static int aeApiCreate(aeEventLoop *eventLoop) {
    aeApiState *state = zmalloc(sizeof(aeApiState));

    if (!state) return -1;
    state->events = zmalloc(sizeof(struct kevent)*eventLoop->setsize);
    if (!state->events) {
        zfree(state);
        return -1;
    }
    state->kqfd = kqueue();
    if (state->kqfd == -1) {
        zfree(state->events);
        zfree(state);
        return -1;
    }
    anetCloexec(state->kqfd);
    state->eventsMask = zmalloc(EVENT_MASK_MALLOC_SIZE(eventLoop->setsize));
    memset(state->eventsMask, 0, EVENT_MASK_MALLOC_SIZE(eventLoop->setsize));
    eventLoop->apidata = state;
    return 0;
}

static int aeApiResize(aeEventLoop *eventLoop, int setsize) {
    aeApiState *state = eventLoop->apidata;

    state->events = zrealloc(state->events, sizeof(struct kevent)*setsize);
    state->eventsMask = zrealloc(state->eventsMask,
EVENT_MASK_MALLOC_SIZE(setsize));
    memset(state->eventsMask, 0, EVENT_MASK_MALLOC_SIZE(setsize));
    return 0;
}

static void aeApiFree(aeEventLoop *eventLoop) {
    aeApiState *state = eventLoop->apidata;

    close(state->kqfd);
    zfree(state->events);
    zfree(state->eventsMask);
    zfree(state);
}

```

```

static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;
    struct kevent ke;

    if (mask & AE_READABLE) {
        EV_SET(&ke, fd, EVFILT_READ, EV_ADD, 0, 0, NULL);
        if (kevent(state->kqfd, &ke, 1, NULL, 0, NULL) == -1) return -1;
    }
    if (mask & AE_WRITABLE) {
        EV_SET(&ke, fd, EVFILT_WRITE, EV_ADD, 0, 0, NULL);
        if (kevent(state->kqfd, &ke, 1, NULL, 0, NULL) == -1) return -1;
    }
    return 0;
}

static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;
    struct kevent ke;

    if (mask & AE_READABLE) {
        EV_SET(&ke, fd, EVFILT_READ, EV_DELETE, 0, 0, NULL);
        kevent(state->kqfd, &ke, 1, NULL, 0, NULL);
    }
    if (mask & AE_WRITABLE) {
        EV_SET(&ke, fd, EVFILT_WRITE, EV_DELETE, 0, 0, NULL);
        kevent(state->kqfd, &ke, 1, NULL, 0, NULL);
    }
}

static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp) {
    aeApiState *state = eventLoop->apidata;
    int retval, numevents = 0;

    if (tvp != NULL) {
        struct timespec timeout;
        timeout.tv_sec = tvp->tv_sec;
        timeout.tv_nsec = tvp->tv_usec * 1000;
        retval = kevent(state->kqfd, NULL, 0, state->events, eventLoop-
>setsize,
                        &timeout);
    } else {
        retval = kevent(state->kqfd, NULL, 0, state->events, eventLoop-
>setsize,
                        NULL);
    }

    if (retval > 0) {
        int j;

        /* Normally we execute the read event first and then the write event.

```

```

    * When the barrier is set, we will do it reverse.
    *
    * However, under kqueue, read and write events would be separate
    * events, which would make it impossible to control the order of
    * reads and writes. So we store the event's mask we've got and merge
    * the same fd events later. */
for (j = 0; j < retval; j++) {
    struct kevent *e = state->events+j;
    int fd = e->ident;
    int mask = 0;

    if (e->filter == EVFILT_READ) mask = AE_READABLE;
    else if (e->filter == EVFILT_WRITE) mask = AE_WRITABLE;
    addEventMask(state->eventsMask, fd, mask);
}

/* Re-traversal to merge read and write events, and set the fd's mask
to
    * 0 so that events are not added again when the fd is encountered
again. */
numevents = 0;
for (j = 0; j < retval; j++) {
    struct kevent *e = state->events+j;
    int fd = e->ident;
    int mask = getEventMask(state->eventsMask, fd);

    if (mask) {
        eventLoop->fired[numevents].fd = fd;
        eventLoop->fired[numevents].mask = mask;
        resetEventMask(state->eventsMask, fd);
        numevents++;
    }
}
} else if (retval == -1 && errno != EINTR) {
    panic("aeApiPoll: kevent, %s", strerror(errno));
}

return numevents;
}

static char *aeApiName(void) {
    return "kqueue";
}

```

/ae_select.c

[to top](#)

```

/* Select()-based ae.c module.
 *
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include <sys/select.h>
#include <string.h>

typedef struct aeApiState {
    fd_set rfdsets, wfdsets;
    /* We need to have a copy of the fd sets as it's not safe to reuse
     * FD sets after select(). */
    fd_set _rfdsets, _wfdsets;
} aeApiState;

static int aeApiCreate(aeEventLoop *eventLoop) {
    aeApiState *state = zmalloc(sizeof(aeApiState));

    if (!state) return -1;
    FD_ZERO(&state->rfdsets);
    FD_ZERO(&state->wfdsets);
    eventLoop->apidata = state;
    return 0;
}

```

```

static int aeApiResize(aeEventLoop *eventLoop, int setsize) {
    /* Just ensure we have enough room in the fd_set type. */
    if (setsize >= FD_SETSIZE) return -1;
    return 0;
}

static void aeApiFree(aeEventLoop *eventLoop) {
    zfree(eventLoop->apidata);
}

static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;

    if (mask & AE_READABLE) FD_SET(fd,&state->rfd);
    if (mask & AE_WRITABLE) FD_SET(fd,&state->wfd);
    return 0;
}

static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int mask) {
    aeApiState *state = eventLoop->apidata;

    if (mask & AE_READABLE) FD_CLR(fd,&state->rfd);
    if (mask & AE_WRITABLE) FD_CLR(fd,&state->wfd);
}

static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp) {
    aeApiState *state = eventLoop->apidata;
    int retval, j, numevents = 0;

    memcpy(&state->_rfd,&state->rfd,sizeof(fd_set));
    memcpy(&state->_wfd,&state->wfd,sizeof(fd_set));

    retval = select(eventLoop->maxfd+1,
                    &state->_rfd,&state->_wfd,NULL,tvp);
    if (retval > 0) {
        for (j = 0; j <= eventLoop->maxfd; j++) {
            int mask = 0;
            aeFileEvent *fe = &eventLoop->events[j];

            if (fe->mask == AE_NONE) continue;
            if (fe->mask & AE_READABLE && FD_ISSET(j,&state->_rfd))
                mask |= AE_READABLE;
            if (fe->mask & AE_WRITABLE && FD_ISSET(j,&state->_wfd))
                mask |= AE_WRITABLE;
            eventLoop->fired[numevents].fd = j;
            eventLoop->fired[numevents].mask = mask;
            numevents++;
        }
    } else if (retval == -1 && errno != EINTR) {
        panic("aeApiPoll: select, %s", strerror(errno));
    }
}

```

```
    }

    return numevents;
}

static char *aeApiName(void) {
    return "select";
}
```

/anet.c

[to top](#)


```

/* anet.c -- Basic TCP socket stuff made a bit less boring
 *
 * Copyright (c) 2006-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "fmacros.h"

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <netdb.h>
#include <errno.h>
#include <stdarg.h>
#include <stdio.h>

#include "anet.h"
#include "config.h"

```

```

#define UNUSED(x) (void)(x)

static void anetSetError(char *err, const char *fmt, ...)
{
    va_list ap;

    if (!err) return;
    va_start(ap, fmt);
    vsnprintf(err, ANET_ERR_LEN, fmt, ap);
    va_end(ap);
}

int anetSetBlock(char *err, int fd, int non_block) {
    int flags;

    /* Set the socket blocking (if non_block is zero) or non-blocking.
     * Note that fcntl(2) for F_GETFL and F_SETFL can't be
     * interrupted by a signal. */
    if ((flags = fcntl(fd, F_GETFL)) == -1) {
        anetSetError(err, "fcntl(F_GETFL): %s", strerror(errno));
        return ANET_ERR;
    }

    /* Check if this flag has been set or unset, if so,
     * then there is no need to call fcntl to set/unset it again. */
    if (!(flags & O_NONBLOCK) == !!non_block)
        return ANET_OK;

    if (non_block)
        flags |= O_NONBLOCK;
    else
        flags &= ~O_NONBLOCK;

    if (fcntl(fd, F_SETFL, flags) == -1) {
        anetSetError(err, "fcntl(F_SETFL,O_NONBLOCK): %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}

int anetNonBlock(char *err, int fd) {
    return anetSetBlock(err, fd, 1);
}

int anetBlock(char *err, int fd) {
    return anetSetBlock(err, fd, 0);
}

/* Enable the FD_CLOEXEC on the given fd to avoid fd leaks.
 * This function should be invoked for fd's on specific places

```

```

    * where fork + execve system calls are called. */
int anetCloexec(int fd) {
    int r;
    int flags;

    do {
        r = fcntl(fd, F_GETFD);
    } while (r == -1 && errno == EINTR);

    if (r == -1 || (r & FD_CLOEXEC))
        return r;

    flags = r | FD_CLOEXEC;

    do {
        r = fcntl(fd, F_SETFD, flags);
    } while (r == -1 && errno == EINTR);

    return r;
}

/* Set TCP keep alive option to detect dead peers. The interval option
 * is only used for Linux as we are using Linux-specific APIs to set
 * the probe send time, interval, and count. */
int anetKeepAlive(char *err, int fd, int interval)
{
    int val = 1;

    if (setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &val, sizeof(val)) == -1)
    {
        anetSetError(err, "setsockopt SO_KEEPALIVE: %s", strerror(errno));
        return ANET_ERR;
    }

#ifdef __linux__
    /* Default settings are more or less garbage, with the keepalive time
     * set to 7200 by default on Linux. Modify settings to make the feature
     * actually useful. */

    /* Send first probe after interval. */
    val = interval;
    if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPIDL, &val, sizeof(val)) < 0) {
        anetSetError(err, "setsockopt TCP_KEEPIDL: %s\n", strerror(errno));
        return ANET_ERR;
    }

    /* Send next probes after the specified interval. Note that we set the
     * delay as interval / 3, as we send three probes before detecting
     * an error (see the next setsockopt call). */
    val = interval/3;
    if (val == 0) val = 1;

```

```

    if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPINTVL, &val, sizeof(val)) < 0) {
        anetSetError(err, "setsockopt TCP_KEEPINTVL: %s\n", strerror(errno));
        return ANET_ERR;
    }

    /* Consider the socket in error state after three we send three ACK
     * probes without getting a reply. */
    val = 3;
    if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPCNT, &val, sizeof(val)) < 0) {
        anetSetError(err, "setsockopt TCP_KEEPCNT: %s\n", strerror(errno));
        return ANET_ERR;
    }
#else
    ((void) interval); /* Avoid unused var warning for non Linux systems. */
#endif

    return ANET_OK;
}

static int anetSetTcpNoDelay(char *err, int fd, int val)
{
    if (setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, &val, sizeof(val)) == -1)
    {
        anetSetError(err, "setsockopt TCP_NODELAY: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}

int anetEnableTcpNoDelay(char *err, int fd)
{
    return anetSetTcpNoDelay(err, fd, 1);
}

int anetDisableTcpNoDelay(char *err, int fd)
{
    return anetSetTcpNoDelay(err, fd, 0);
}

/* Set the socket send timeout (SO_SNDTIMEO socket option) to the specified
 * number of milliseconds, or disable it if the 'ms' argument is zero. */
int anetSendTimeout(char *err, int fd, long long ms) {
    struct timeval tv;

    tv.tv_sec = ms/1000;
    tv.tv_usec = (ms%1000)*1000;
    if (setsockopt(fd, SOL_SOCKET, SO_SNDTIMEO, &tv, sizeof(tv)) == -1) {
        anetSetError(err, "setsockopt SO_SNDTIMEO: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}

```

```

}

/* Set the socket receive timeout (SO_RCVTIMEO socket option) to the specified
 * number of milliseconds, or disable it if the 'ms' argument is zero. */
int anetRecvTimeout(char *err, int fd, long long ms) {
    struct timeval tv;

    tv.tv_sec = ms/1000;
    tv.tv_usec = (ms%1000)*1000;
    if (setsockopt(fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) == -1) {
        anetSetError(err, "setsockopt SO_RCVTIMEO: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}

/* Resolve the hostname "host" and set the string representation of the
 * IP address into the buffer pointed by "ipbuf".
 *
 * If flags is set to ANET_IP_ONLY the function only resolves hostnames
 * that are actually already IPv4 or IPv6 addresses. This turns the function
 * into a validating / normalizing function. */
int anetResolve(char *err, char *host, char *ipbuf, size_t ipbuf_len,
                int flags)
{
    struct addrinfo hints, *info;
    int rv;

    memset(&hints, 0, sizeof(hints));
    if (flags & ANET_IP_ONLY) hints.ai_flags = AI_NUMERICHOST;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM; /* specify socktype to avoid dups */

    if ((rv = getaddrinfo(host, NULL, &hints, &info)) != 0) {
        anetSetError(err, "%s", gai_strerror(rv));
        return ANET_ERR;
    }
    if (info->ai_family == AF_INET) {
        struct sockaddr_in *sa = (struct sockaddr_in *)info->ai_addr;
        inet_ntop(AF_INET, &(sa->sin_addr), ipbuf, ipbuf_len);
    } else {
        struct sockaddr_in6 *sa = (struct sockaddr_in6 *)info->ai_addr;
        inet_ntop(AF_INET6, &(sa->sin6_addr), ipbuf, ipbuf_len);
    }

    freeaddrinfo(info);
    return ANET_OK;
}

static int anetSetReuseAddr(char *err, int fd) {
    int yes = 1;

```

```

/* Make sure connection-intensive things like the redis benchmark
 * will be able to close/open sockets a zillion of times */
if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes)) == -1) {
    anetSetError(err, "setsockopt SO_REUSEADDR: %s", strerror(errno));
    return ANET_ERR;
}
return ANET_OK;
}

static int anetCreateSocket(char *err, int domain) {
    int s;
    if ((s = socket(domain, SOCK_STREAM, 0)) == -1) {
        anetSetError(err, "creating socket: %s", strerror(errno));
        return ANET_ERR;
    }

    /* Make sure connection-intensive things like the redis benchmark
     * will be able to close/open sockets a zillion of times */
    if (anetSetReuseAddr(err,s) == ANET_ERR) {
        close(s);
        return ANET_ERR;
    }
    return s;
}

#define ANET_CONNECT_NONE 0
#define ANET_CONNECT_NONBLOCK 1
#define ANET_CONNECT_BE_BINDING 2 /* Best effort binding. */
static int anetTcpGenericConnect(char *err, const char *addr, int port,
                                const char *source_addr, int flags)
{
    int s = ANET_ERR, rv;
    char portstr[6]; /* strlen("65535") + 1; */
    struct addrinfo hints, *servinfo, *bservinfo, *p, *b;

    snprintf(portstr, sizeof(portstr), "%d", port);
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((rv = getaddrinfo(addr, portstr, &hints, &servinfo)) != 0) {
        anetSetError(err, "%s", gai_strerror(rv));
        return ANET_ERR;
    }
    for (p = servinfo; p != NULL; p = p->ai_next) {
        /* Try to create the socket and to connect it.
         * If we fail in the socket() call, or on connect(), we retry with
         * the next entry in servinfo. */
        if ((s = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1)
            continue;
        if (anetSetReuseAddr(err,s) == ANET_ERR) goto error;
    }
}

```

```

if (flags & ANET_CONNECT_NONBLOCK && anetNonBlock(err,s) != ANET_OK)
    goto error;
if (source_addr) {
    int bound = 0;
    /* Using getaddrinfo saves us from self-determining IPv4 vs IPv6 */
    if ((rv = getaddrinfo(source_addr, NULL, &hints, &bserverinfo)) != 0)
    {
        anetSetError(err, "%s", gai_strerror(rv));
        goto error;
    }
    for (b = bserverinfo; b != NULL; b = b->ai_next) {
        if (bind(s,b->ai_addr,b->ai_addrlen) != -1) {
            bound = 1;
            break;
        }
    }
    freeaddrinfo(bserverinfo);
    if (!bound) {
        anetSetError(err, "bind: %s", strerror(errno));
        goto error;
    }
}
if (connect(s,p->ai_addr,p->ai_addrlen) == -1) {
    /* If the socket is non-blocking, it is ok for connect() to
     * return an EINPROGRESS error here. */
    if (errno == EINPROGRESS && flags & ANET_CONNECT_NONBLOCK)
        goto end;
    close(s);
    s = ANET_ERR;
    continue;
}

/* If we ended an iteration of the for loop without errors, we
 * have a connected socket. Let's return to the caller. */
goto end;
}
if (p == NULL)
    anetSetError(err, "creating socket: %s", strerror(errno));

error:
    if (s != ANET_ERR) {
        close(s);
        s = ANET_ERR;
    }

end:
    freeaddrinfo(servinfo);

/* Handle best effort binding: if a binding address was used, but it is
 * not possible to create a socket, try again without a binding address. */
if (s == ANET_ERR && source_addr && (flags & ANET_CONNECT_BE_BINDING)) {

```

```

        return anetTcpGenericConnect(err,addr,port,NULL,flags);
    } else {
        return s;
    }
}

int anetTcpNonBlockConnect(char *err, const char *addr, int port)
{
    return anetTcpGenericConnect(err,addr,port,NULL,ANET_CONNECT_NONBLOCK);
}

int anetTcpNonBlockBestEffortBindConnect(char *err, const char *addr, int port,
                                          const char *source_addr)
{
    return anetTcpGenericConnect(err,addr,port,source_addr,
                                  ANET_CONNECT_NONBLOCK|ANET_CONNECT_BE_BINDING);
}

int anetUnixGenericConnect(char *err, const char *path, int flags)
{
    int s;
    struct sockaddr_un sa;

    if ((s = anetCreateSocket(err,AF_LOCAL)) == ANET_ERR)
        return ANET_ERR;

    sa.sun_family = AF_LOCAL;
    strncpy(sa.sun_path,path,sizeof(sa.sun_path)-1);
    if (flags & ANET_CONNECT_NONBLOCK) {
        if (anetNonBlock(err,s) != ANET_OK) {
            close(s);
            return ANET_ERR;
        }
    }
    if (connect(s,(struct sockaddr*)&sa,sizeof(sa)) == -1) {
        if (errno == EINPROGRESS &&
            flags & ANET_CONNECT_NONBLOCK)
            return s;

        anetSetError(err, "connect: %s", strerror(errno));
        close(s);
        return ANET_ERR;
    }
    return s;
}

static int anetListen(char *err, int s, struct sockaddr *sa, socklen_t len, int
backlog) {
    if (bind(s,sa,len) == -1) {
        anetSetError(err, "bind: %s", strerror(errno));
        close(s);
    }
}

```



```

        return ANET_ERR;
    }

    if (listen(s, backlog) == -1) {
        anetSetError(err, "listen: %s", strerror(errno));
        close(s);
        return ANET_ERR;
    }
    return ANET_OK;
}

static int anetV6Only(char *err, int s) {
    int yes = 1;
    if (setsockopt(s, IPPROTO_IPV6, IPV6_V6ONLY, &yes, sizeof(yes)) == -1) {
        anetSetError(err, "setsockopt: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}

static int _anetTcpServer(char *err, int port, char *bindaddr, int af, int
backlog)
{
    int s = -1, rv;
    char _port[6]; /* strlen("65535") */
    struct addrinfo hints, *servinfo, *p;

    snprintf(_port, 6, "%d", port);
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = af;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; /* No effect if bindaddr != NULL */
    if (bindaddr && !strcmp("*", bindaddr))
        bindaddr = NULL;
    if (af == AF_INET6 && bindaddr && !strcmp(":::", bindaddr))
        bindaddr = NULL;

    if ((rv = getaddrinfo(bindaddr, _port, &hints, &servinfo)) != 0) {
        anetSetError(err, "%s", gai_strerror(rv));
        return ANET_ERR;
    }
    for (p = servinfo; p != NULL; p = p->ai_next) {
        if ((s = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1)
            continue;

        if (af == AF_INET6 && anetV6Only(err, s) == ANET_ERR) goto error;
        if (anetSetReuseAddr(err, s) == ANET_ERR) goto error;
        if (anetListen(err, s, p->ai_addr, p->ai_addrlen, backlog) == ANET_ERR) s =
ANET_ERR;
        goto end;
    }
}

```

```

    if (p == NULL) {
        anetSetError(err, "unable to bind socket, errno: %d", errno);
        goto error;
    }

error:
    if (s != -1) close(s);
    s = ANET_ERR;
end:
    freeaddrinfo(servinfo);
    return s;
}

int anetTcpServer(char *err, int port, char *bindaddr, int backlog)
{
    return _anetTcpServer(err, port, bindaddr, AF_INET, backlog);
}

int anetTcp6Server(char *err, int port, char *bindaddr, int backlog)
{
    return _anetTcpServer(err, port, bindaddr, AF_INET6, backlog);
}

int anetUnixServer(char *err, char *path, mode_t perm, int backlog)
{
    int s;
    struct sockaddr_un sa;

    if (strlen(path) > sizeof(sa.sun_path)-1) {
        anetSetError(err, "unix socket path too long (%zu), must be under %zu",
            strlen(path), sizeof(sa.sun_path));
        return ANET_ERR;
    }
    if ((s = anetCreateSocket(err, AF_LOCAL)) == ANET_ERR)
        return ANET_ERR;

    memset(&sa, 0, sizeof(sa));
    sa.sun_family = AF_LOCAL;
    strncpy(sa.sun_path, path, sizeof(sa.sun_path)-1);
    if (anetListen(err, s, (struct sockaddr*)&sa, sizeof(sa), backlog) == ANET_ERR)
        return ANET_ERR;
    if (perm)
        chmod(sa.sun_path, perm);
    return s;
}

/* Accept a connection and also make sure the socket is non-blocking, and
CLOEXEC.
 * returns the new socket FD, or -1 on error. */
static int anetGenericAccept(char *err, int s, struct sockaddr *sa, socklen_t
*len) {

```

```

    int fd;
    do {
        /* Use the accept4() call on linux to simultaneously accept and
         * set a socket as non-blocking. */
#ifdef HAVE_ACCEPT4
        fd = accept4(s, sa, len,  SOCK_NONBLOCK | SOCK_CLOEXEC);
#else
        fd = accept(s,sa,len);
#endif
    } while(fd == -1 && errno == EINTR);
    if (fd == -1) {
        anetSetError(err, "accept: %s", strerror(errno));
        return ANET_ERR;
    }
#ifdef HAVE_ACCEPT4
    if (anetCloexec(fd) == -1) {
        anetSetError(err, "anetCloexec: %s", strerror(errno));
        close(fd);
        return ANET_ERR;
    }
    if (anetNonBlock(err, fd) != ANET_OK) {
        close(fd);
        return ANET_ERR;
    }
#endif
    return fd;
}

/* Accept a connection and also make sure the socket is non-blocking, and
CLOEXEC.
 * returns the new socket FD, or -1 on error. */
int anetTcpAccept(char *err, int serversock, char *ip, size_t ip_len, int
*port) {
    int fd;
    struct sockaddr_storage sa;
    socklen_t salen = sizeof(sa);
    if ((fd = anetGenericAccept(err,serversock,(struct sockaddr*)&sa,&salen))
== ANET_ERR)
        return ANET_ERR;

    if (sa.ss_family == AF_INET) {
        struct sockaddr_in *s = (struct sockaddr_in *)&sa;
        if (ip) inet_ntop(AF_INET,(void*)&(s->sin_addr),ip,ip_len);
        if (port) *port = ntohs(s->sin_port);
    } else {
        struct sockaddr_in6 *s = (struct sockaddr_in6 *)&sa;
        if (ip) inet_ntop(AF_INET6,(void*)&(s->sin6_addr),ip,ip_len);
        if (port) *port = ntohs(s->sin6_port);
    }
    return fd;
}

```

```

/* Accept a connection and also make sure the socket is non-blocking, and
CLOEXEC.
 * returns the new socket FD, or -1 on error. */
int anetUnixAccept(char *err, int s) {
    int fd;
    struct sockaddr_un sa;
    socklen_t salen = sizeof(sa);
    if ((fd = anetGenericAccept(err,s,(struct sockaddr*)&sa,&salen)) ==
ANET_ERR)
        return ANET_ERR;

    return fd;
}

int anetFdToString(int fd, char *ip, size_t ip_len, int *port, int
fd_to_str_type) {
    struct sockaddr_storage sa;
    socklen_t salen = sizeof(sa);

    if (fd_to_str_type == FD_TO_PEER_NAME) {
        if (getpeername(fd, (struct sockaddr *)&sa, &salen) == -1) goto error;
    } else {
        if (getsockname(fd, (struct sockaddr *)&sa, &salen) == -1) goto error;
    }

    if (sa.ss_family == AF_INET) {
        struct sockaddr_in *s = (struct sockaddr_in *)&sa;
        if (ip) {
            if (inet_ntop(AF_INET,(void*)&(s->sin_addr),ip,ip_len) == NULL)
                goto error;
        }
        if (port) *port = ntohs(s->sin_port);
    } else if (sa.ss_family == AF_INET6) {
        struct sockaddr_in6 *s = (struct sockaddr_in6 *)&sa;
        if (ip) {
            if (inet_ntop(AF_INET6,(void*)&(s->sin6_addr),ip,ip_len) == NULL)
                goto error;
        }
        if (port) *port = ntohs(s->sin6_port);
    } else if (sa.ss_family == AF_UNIX) {
        if (ip) {
            int res = snprintf(ip, ip_len, "/unixsocket");
            if (res < 0 || (unsigned int) res >= ip_len) goto error;
        }
        if (port) *port = 0;
    } else {
        goto error;
    }
    return 0;
}

```

```

error:
    if (ip) {
        if (ip_len >= 2) {
            ip[0] = '?';
            ip[1] = '\0';
        } else if (ip_len == 1) {
            ip[0] = '\0';
        }
    }
    if (port) *port = 0;
    return -1;
}

/* Format an IP,port pair into something easy to parse. If IP is IPv6
 * (matches for ":"), the ip is surrounded by []. IP and port are just
 * separated by colons. This the standard to display addresses within Redis. */
int anetFormatAddr(char *buf, size_t buf_len, char *ip, int port) {
    return snprintf(buf, buf_len, strchr(ip, ':') ?
        "[%s]:%d" : "%s:%d", ip, port);
}

/* Like anetFormatAddr() but extract ip and port from the socket's
peer/sockname. */
int anetFormatFdAddr(int fd, char *buf, size_t buf_len, int fd_to_str_type) {
    char ip[INET6_ADDRSTRLEN];
    int port;

    anetFdToString(fd, ip, sizeof(ip), &port, fd_to_str_type);
    return anetFormatAddr(buf, buf_len, ip, port);
}

/* Create a pipe buffer with given flags for read end and write end.
 * Note that it supports the file flags defined by pipe2() and fcntl(F_SETFL),
 * and one of the use cases is O_CLOEXEC|O_NONBLOCK. */
int anetPipe(int fds[2], int read_flags, int write_flags) {
    int pipe_flags = 0;
#ifdef __linux__ || defined(__FreeBSD__)
    /* When possible, try to leverage pipe2() to apply flags that are common to
both ends.
    * There is no harm to set O_CLOEXEC to prevent fd leaks. */
    pipe_flags = O_CLOEXEC | (read_flags & write_flags);
    if (pipe2(fds, pipe_flags)) {
        /* Fail on real failures, and fallback to simple pipe if pipe2 is
unsupported. */
        if (errno != ENOSYS && errno != EINVAL)
            return -1;
        pipe_flags = 0;
    } else {
        /* If the flags on both ends are identical, no need to do anything
else. */
        if ((O_CLOEXEC | read_flags) == (O_CLOEXEC | write_flags))

```

```

        return 0;
        /* Clear the flags which have already been set using pipe2. */
        read_flags &= ~pipe_flags;
        write_flags &= ~pipe_flags;
    }
#endif

    /* When we reach here with pipe_flags of 0, it means pipe2 failed (or was
    not attempted),
    * so we try to use pipe. Otherwise, we skip and proceed to set specific
    flags below. */
    if (pipe_flags == 0 && pipe(fds))
        return -1;

    /* File descriptor flags.
    * Currently, only one such flag is defined: FD_CLOEXEC, the close-on-exec
    flag. */
    if (read_flags & O_CLOEXEC)
        if (fcntl(fds[0], F_SETFD, FD_CLOEXEC))
            goto error;
    if (write_flags & O_CLOEXEC)
        if (fcntl(fds[1], F_SETFD, FD_CLOEXEC))
            goto error;

    /* File status flags after clearing the file descriptor flag O_CLOEXEC. */
    read_flags &= ~O_CLOEXEC;
    if (read_flags)
        if (fcntl(fds[0], F_SETFL, read_flags))
            goto error;
    write_flags &= ~O_CLOEXEC;
    if (write_flags)
        if (fcntl(fds[1], F_SETFL, write_flags))
            goto error;

    return 0;

error:
    close(fds[0]);
    close(fds[1]);
    return -1;
}

int anetSetSockMarkId(char *err, int fd, uint32_t id) {
#ifdef HAVE_SOCKOPTMARKID
    if (setsockopt(fd, SOL_SOCKET, SOCKOPTMARKID, (void *)&id, sizeof(id)) ==
    -1) {
        anetSetError(err, "setsockopt: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
#else

```

```
UNUSED(fd);
UNUSED(id);
anetSetError(err,"anetSetSockMarkid unsupported on this platform");
return ANET_OK;
#endif
}
```

/anet.h

[to top](#)

```

/* anet.c -- Basic TCP socket stuff made a bit less boring
 *
 * Copyright (c) 2006-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifdef ANET_H
#define ANET_H

#include <sys/types.h>

#define ANET_OK 0
#define ANET_ERR -1
#define ANET_ERR_LEN 256

/* Flags used with certain functions. */
#define ANET_NONE 0
#define ANET_IP_ONLY (1<<0)

#ifdef __sun || defined(_AIX)
#define AF_LOCAL AF_UNIX
#endif

#ifdef _AIX
#undef ip_len
#endif

```



```

/* FD to address string conversion types */
#define FD_TO_PEER_NAME 0
#define FD_TO_SOCKET_NAME 1

int anetTcpNonBlockConnect(char *err, const char *addr, int port);
int anetTcpNonBlockBestEffortBindConnect(char *err, const char *addr, int port,
const char *source_addr);
int anetResolve(char *err, char *host, char *ipbuf, size_t ipbuf_len, int
flags);
int anetTcpServer(char *err, int port, char *bindaddr, int backlog);
int anetTcp6Server(char *err, int port, char *bindaddr, int backlog);
int anetUnixServer(char *err, char *path, mode_t perm, int backlog);
int anetTcpAccept(char *err, int serversock, char *ip, size_t ip_len, int
*port);
int anetUnixAccept(char *err, int serversock);
int anetNonBlock(char *err, int fd);
int anetBlock(char *err, int fd);
int anetCloexec(int fd);
int anetEnableTcpNoDelay(char *err, int fd);
int anetDisableTcpNoDelay(char *err, int fd);
int anetSendTimeout(char *err, int fd, long long ms);
int anetRecvTimeout(char *err, int fd, long long ms);
int anetFdToString(int fd, char *ip, size_t ip_len, int *port, int
fd_to_str_type);
int anetKeepAlive(char *err, int fd, int interval);
int anetFormatAddr(char *fmt, size_t fmt_len, char *ip, int port);
int anetFormatFdAddr(int fd, char *buf, size_t buf_len, int fd_to_str_type);
int anetPipe(int fds[2], int read_flags, int write_flags);
int anetSetSockMarkId(char *err, int fd, uint32_t id);

#endif

```

/aof.c

[to top](#)

```

/*
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "server.h"
#include "bio.h"
#include "rio.h"
#include "functions.h"

#include <signal.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/param.h>

void freeClientArgv(client *c);
off_t getAppendOnlyFileSize(sds filename, int *status);
off_t getBaseAndIncrAppendOnlyFilesSize(aofManifest *am, int *status);
int getBaseAndIncrAppendOnlyFilesNum(aofManifest *am);
int aofFileExist(char *filename);
int rewriteAppendOnlyFile(char *filename);
aofManifest *aofLoadManifestFromFile(sds am_filepath);

```

```

void aofManifestFreeAndUpdate(aofManifest *am);

/* -----
 * AOF Manifest file implementation.
 *
 * The following code implements the read/write logic of AOF manifest file,
which
 * is used to track and manage all AOF files.
 *
 * Append-only files consist of three types:
 *
 * BASE: Represents a Redis snapshot from the time of last AOF rewrite. The
manifest
 * file contains at most a single BASE file, which will always be the first
file in the
 * list.
 *
 * INCR: Represents all write commands executed by Redis following the last
successful
 * AOF rewrite. In some cases it is possible to have several ordered INCR
files. For
 * example:
 *   - During an on-going AOF rewrite
 *   - After an AOF rewrite was aborted/failed, and before the next one
succeeded.
 *
 * HISTORY: After a successful rewrite, the previous BASE and INCR become
HISTORY files.
 * They will be automatically removed unless garbage collection is disabled.
 *
 * The following is a possible AOF manifest file content:
 *
 * file appendonly.aof.2.base.rdb seq 2 type b
 * file appendonly.aof.1.incr.aof seq 1 type h
 * file appendonly.aof.2.incr.aof seq 2 type h
 * file appendonly.aof.3.incr.aof seq 3 type h
 * file appendonly.aof.4.incr.aof seq 4 type i
 * file appendonly.aof.5.incr.aof seq 5 type i
 * ----- */

/* Naming rules. */
#define BASE_FILE_SUFFIX      ".base"
#define INCR_FILE_SUFFIX     ".incr"
#define RDB_FORMAT_SUFFIX    ".rdb"
#define AOF_FORMAT_SUFFIX    ".aof"
#define MANIFEST_NAME_SUFFIX ".manifest"
#define TEMP_FILE_NAME_PREFIX "temp-"

/* AOF manifest key. */
#define AOF_MANIFEST_KEY_FILE_NAME "file"
#define AOF_MANIFEST_KEY_FILE_SEQ "seq"

```

```

#define AOF_MANIFEST_KEY_FILE_TYPE    "type"

/* Create an empty aofInfo. */
aofInfo *aofInfoCreate(void) {
    return zcalloc(sizeof(aofInfo));
}

/* Free the aofInfo structure (pointed to by ai) and its embedded file_name. */
void aofInfoFree(aofInfo *ai) {
    serverAssert(ai != NULL);
    if (ai->file_name) sdsfree(ai->file_name);
    zfree(ai);
}

/* Deep copy an aofInfo. */
aofInfo *aofInfoDup(aofInfo *orig) {
    serverAssert(orig != NULL);
    aofInfo *ai = aofInfoCreate();
    ai->file_name = sdsdup(orig->file_name);
    ai->file_seq = orig->file_seq;
    ai->file_type = orig->file_type;
    return ai;
}

/* Format aofInfo as a string and it will be a line in the manifest. */
sds aofInfoFormat(sds buf, aofInfo *ai) {
    sds filename_repr = NULL;

    if (sdsneedsrepr(ai->file_name))
        filename_repr = sdscatrepr(sdsempy(), ai->file_name, sdslen(ai->file_name));

    sds ret = sdscatprintf(buf, "%s %s %s %lld %s %c\n",
        AOF_MANIFEST_KEY_FILE_NAME, filename_repr ? filename_repr : ai->file_name,
        AOF_MANIFEST_KEY_FILE_SEQ, ai->file_seq,
        AOF_MANIFEST_KEY_FILE_TYPE, ai->file_type);
    sdsfree(filename_repr);

    return ret;
}

/* Method to free AOF list elements. */
void aofListFree(void *item) {
    aofInfo *ai = (aofInfo *)item;
    aofInfoFree(ai);
}

/* Method to duplicate AOF list elements. */
void *aofListDup(void *item) {
    return aofInfoDup(item);
}

```

```

}

/* Create an empty aofManifest, which will be called in
`aofLoadManifestFromDisk`. */
aofManifest *aofManifestCreate(void) {
    aofManifest *am = zcalloc(sizeof(aofManifest));
    am->incr_aof_list = listCreate();
    am->history_aof_list = listCreate();
    listSetFreeMethod(am->incr_aof_list, aofListFree);
    listSetDupMethod(am->incr_aof_list, aofListDup);
    listSetFreeMethod(am->history_aof_list, aofListFree);
    listSetDupMethod(am->history_aof_list, aofListDup);
    return am;
}

/* Free the aofManifest structure (pointed to by am) and its embedded members.
*/
void aofManifestFree(aofManifest *am) {
    if (am->base_aof_info) aofInfoFree(am->base_aof_info);
    if (am->incr_aof_list) listRelease(am->incr_aof_list);
    if (am->history_aof_list) listRelease(am->history_aof_list);
    zfree(am);
}

sds getAofManifestFileName() {
    return sdscatprintf(sdsempty(), "%s%s", server.aof_filename,
        MANIFEST_NAME_SUFFIX);
}

sds getTempAofManifestFileName() {
    return sdscatprintf(sdsempty(), "%s%s%s", TEMP_FILE_NAME_PREFIX,
        server.aof_filename, MANIFEST_NAME_SUFFIX);
}

/* Returns the string representation of aofManifest pointed to by am.
 *
 * The string is multiple lines separated by '\n', and each line represents
 * an AOF file.
 *
 * Each line is space delimited and contains 6 fields, as follows:
 * "file" [filename] "seq" [sequence] "type" [type]
 *
 * Where "file", "seq" and "type" are keywords that describe the next value,
 * [filename] and [sequence] describe file name and order, and [type] is one
 * of 'b' (base), 'h' (history) or 'i' (incr).
 *
 * The base file, if exists, will always be first, followed by history files,
 * and incremental files.
 */
sds getAofManifestAsString(aofManifest *am) {
    serverAssert(am != NULL);

```

```

sds buf = sdsempty();
listNode *ln;
listIter li;

/* 1. Add BASE File information, it is always at the beginning
 * of the manifest file. */
if (am->base_aof_info) {
    buf = aofInfoFormat(buf, am->base_aof_info);
}

/* 2. Add HISTORY type AOF information. */
listRewind(am->history_aof_list, &li);
while ((ln = listNext(&li)) != NULL) {
    aofInfo *ai = (aofInfo*)ln->value;
    buf = aofInfoFormat(buf, ai);
}

/* 3. Add INCR type AOF information. */
listRewind(am->incr_aof_list, &li);
while ((ln = listNext(&li)) != NULL) {
    aofInfo *ai = (aofInfo*)ln->value;
    buf = aofInfoFormat(buf, ai);
}

return buf;
}

/* Load the manifest information from the disk to `server.aof_manifest`
 * when the Redis server start.
 *
 * During loading, this function does strict error checking and will abort
 * the entire Redis server process on error (I/O error, invalid format, etc.)
 *
 * If the AOF directory or manifest file do not exist, this will be ignored
 * in order to support seamless upgrades from previous versions which did not
 * use them.
 */
void aofLoadManifestFromDisk(void) {
    server.aof_manifest = aofManifestCreate();
    if (!dirExists(server.aof_dirname)) {
        serverLog(LL_NOTICE, "The AOF directory %s doesn't exist",
server.aof_dirname);
        return;
    }

    sds am_name = getAofManifestFileName();
    sds am_filepath = makePath(server.aof_dirname, am_name);
    if (!fileExist(am_filepath)) {
        serverLog(LL_NOTICE, "The AOF manifest file %s doesn't exist",
am_name);
    }
}

```

```

        sdsfree(am_name);
        sdsfree(am_filepath);
        return;
    }

    aofManifest *am = aofLoadManifestFromFile(am_filepath);
    if (am) aofManifestFreeAndUpdate(am);
    sdsfree(am_name);
    sdsfree(am_filepath);
}

/* Generic manifest loading function, used in `aofLoadManifestFromDisk` and
redis-check-aof tool. */
#define MANIFEST_MAX_LINE 1024
aofManifest *aofLoadManifestFromFile(sds am_filepath) {
    const char *err = NULL;
    long long maxseq = 0;

    aofManifest *am = aofManifestCreate();
    FILE *fp = fopen(am_filepath, "r");
    if (fp == NULL) {
        serverLog(LL_WARNING, "Fatal error: can't open the AOF manifest "
            "file %s for reading: %s", am_filepath, strerror(errno));
        exit(1);
    }

    char buf[MANIFEST_MAX_LINE+1];
    sds *argv = NULL;
    int argc;
    aofInfo *ai = NULL;

    sds line = NULL;
    int linenum = 0;

    while (1) {
        if (fgets(buf, MANIFEST_MAX_LINE+1, fp) == NULL) {
            if (feof(fp)) {
                if (linenum == 0) {
                    err = "Found an empty AOF manifest";
                    goto loaderr;
                } else {
                    break;
                }
            } else {
                err = "Read AOF manifest failed";
                goto loaderr;
            }
        }

        linenum++;
    }

```

```

/* Skip comments lines */
if (buf[0] == '#') continue;

if (strchr(buf, '\n') == NULL) {
    err = "The AOF manifest file contains too long line";
    goto loaderr;
}

line = sdstrim(sdsnew(buf), " \t\r\n");
if (!sdslen(line)) {
    err = "Invalid AOF manifest file format";
    goto loaderr;
}

argv = sdssplitargs(line, &argc);
/* 'argc < 6' was done for forward compatibility. */
if (argv == NULL || argc < 6 || (argc % 2)) {
    err = "Invalid AOF manifest file format";
    goto loaderr;
}

ai = aofInfoCreate();
for (int i = 0; i < argc; i += 2) {
    if (!strcasecmp(argv[i], AOF_MANIFEST_KEY_FILE_NAME)) {
        ai->file_name = sdsnew(argv[i+1]);
        if (!pathIsBaseName(ai->file_name)) {
            err = "File can't be a path, just a filename";
            goto loaderr;
        }
    } else if (!strcasecmp(argv[i], AOF_MANIFEST_KEY_FILE_SEQ)) {
        ai->file_seq = atoll(argv[i+1]);
    } else if (!strcasecmp(argv[i], AOF_MANIFEST_KEY_FILE_TYPE)) {
        ai->file_type = (argv[i+1])[0];
    }
    /* else if (!strcasecmp(argv[i], AOF_MANIFEST_KEY_OTHER)) {} */
}

/* We have to make sure we load all the information. */
if (!ai->file_name || !ai->file_seq || !ai->file_type) {
    err = "Invalid AOF manifest file format";
    goto loaderr;
}

sdsfreesplitres(argv, argc);
argv = NULL;

if (ai->file_type == AOF_FILE_TYPE_BASE) {
    if (am->base_aof_info) {
        err = "Found duplicate base file information";
        goto loaderr;
    }
}

```



```

        am->base_aof_info = ai;
        am->curr_base_file_seq = ai->file_seq;
    } else if (ai->file_type == AOF_FILE_TYPE_HIST) {
        listAddNodeTail(am->history_aof_list, ai);
    } else if (ai->file_type == AOF_FILE_TYPE_INCR) {
        if (ai->file_seq <= maxseq) {
            err = "Found a non-monotonic sequence number";
            goto loaderr;
        }
        listAddNodeTail(am->incr_aof_list, ai);
        am->curr_incr_file_seq = ai->file_seq;
        maxseq = ai->file_seq;
    } else {
        err = "Unknown AOF file type";
        goto loaderr;
    }
}

sdsfree(line);
line = NULL;
ai = NULL;
}

fclose(fp);
return am;

```

loaderr:

```

/* Sanitizer suppression: may report a false positive if we goto loaderr
 * and exit(1) without freeing these allocations. */
if (argv) sdsfreesplitres(argv, argc);
if (ai) aofInfoFree(ai);

```

```

serverLog(LL_WARNING, "\n*** FATAL AOF MANIFEST FILE ERROR ***\n");
if (line) {
    serverLog(LL_WARNING, "Reading the manifest file, at line %d\n",
linenum);
    serverLog(LL_WARNING, ">>> '%s'\n", line);
}
serverLog(LL_WARNING, "%s\n", err);
exit(1);
}

```

/* Deep copy an aofManifest from orig.

```

 *
 * In `backgroundRewriteDoneHandler` and `openNewIncrAofForAppend`, we will
 * first deep copy a temporary AOF manifest from the `server.aof_manifest` and
 * try to modify it. Once everything is modified, we will atomically make the
 * `server.aof_manifest` point to this temporary aof_manifest.
 */

```

```

aofManifest *aofManifestDup(aofManifest *orig) {
    serverAssert(orig != NULL);
    aofManifest *am = zcalloc(sizeof(aofManifest));

```

```

am->curr_base_file_seq = orig->curr_base_file_seq;
am->curr_incr_file_seq = orig->curr_incr_file_seq;
am->dirty = orig->dirty;

if (orig->base_aof_info) {
    am->base_aof_info = aofInfoDup(orig->base_aof_info);
}

am->incr_aof_list = listDup(orig->incr_aof_list);
am->history_aof_list = listDup(orig->history_aof_list);
serverAssert(am->incr_aof_list != NULL);
serverAssert(am->history_aof_list != NULL);
return am;
}

/* Change the `server.aof_manifest` pointer to 'am' and free the previous
 * one if we have. */
void aofManifestFreeAndUpdate(aofManifest *am) {
    serverAssert(am != NULL);
    if (server.aof_manifest) aofManifestFree(server.aof_manifest);
    server.aof_manifest = am;
}

/* Called in `backgroundRewriteDoneHandler` to get a new BASE file
 * name, and mark the previous (if we have) BASE file as HISTORY type.
 *
 * BASE file naming rules: `server.aof_filename`.seq.base.format
 *
 * for example:
 * appendonly.aof.1.base.aof (server.aof_use_rdb_preamble is no)
 * appendonly.aof.1.base.rdb (server.aof_use_rdb_preamble is yes)
 */
sds getNewBaseFileNameAndMarkPreAsHistory(aofManifest *am) {
    serverAssert(am != NULL);
    if (am->base_aof_info) {
        serverAssert(am->base_aof_info->file_type == AOF_FILE_TYPE_BASE);
        am->base_aof_info->file_type = AOF_FILE_TYPE_HIST;
        listAddNodeHead(am->history_aof_list, am->base_aof_info);
    }

    char *format_suffix = server.aof_use_rdb_preamble ?
        RDB_FORMAT_SUFFIX:AOF_FORMAT_SUFFIX;

    aofInfo *ai = aofInfoCreate();
    ai->file_name = sdscatprintf(sdsempty(), "%s.%lld%s%s",
server.aof_filename,
                                ++am->curr_base_file_seq, BASE_FILE_SUFFIX,
format_suffix);
    ai->file_seq = am->curr_base_file_seq;
    ai->file_type = AOF_FILE_TYPE_BASE;

```

```

    am->base_aof_info = ai;
    am->dirty = 1;
    return am->base_aof_info->file_name;
}

/* Get a new INCR type AOF name.
 *
 * INCR AOF naming rules: `server.aof_filename`.seq.incr.aof
 *
 * for example:
 *  appendonly.aof.1.incr.aof
 */
sds getNewIncrAofName(aofManifest *am) {
    aofInfo *ai = aofInfoCreate();
    ai->file_type = AOF_FILE_TYPE_INCR;
    ai->file_name = sdscatprintf(sdsempy(), "%s.%lld%s",
server.aof_filename,
                                ++am->curr_incr_file_seq, INCR_FILE_SUFFIX,
AOF_FORMAT_SUFFIX);
    ai->file_seq = am->curr_incr_file_seq;
    listAddNodeTail(am->incr_aof_list, ai);
    am->dirty = 1;
    return ai->file_name;
}

/* Get temp INCR type AOF name. */
sds getTempIncrAofName() {
    return sdscatprintf(sdsempy(), "%s%s", TEMP_FILE_NAME_PREFIX,
server.aof_filename,
                                INCR_FILE_SUFFIX);
}

/* Get the last INCR AOF name or create a new one. */
sds getLastIncrAofName(aofManifest *am) {
    serverAssert(am != NULL);

    /* If 'incr_aof_list' is empty, just create a new one. */
    if (!listLength(am->incr_aof_list)) {
        return getNewIncrAofName(am);
    }

    /* Or return the last one. */
    listNode *lastnode = listIndex(am->incr_aof_list, -1);
    aofInfo *ai = listNodeValue(lastnode);
    return ai->file_name;
}

/* Called in `backgroundRewriteDoneHandler`. when AOFRW success, This
 * function will change the AOF file type in 'incr_aof_list' from
 * AOF_FILE_TYPE_INCR to AOF_FILE_TYPE_HIST, and move them to the
 * 'history_aof_list'.

```

```

*/
void markRewrittenIncrAofAsHistory(aofManifest *am) {
    serverAssert(am != NULL);
    if (!listLength(am->incr_aof_list)) {
        return;
    }

    listNode *ln;
    listIter li;

    listRewindTail(am->incr_aof_list, &li);

    /* "server.aof_fd != -1" means AOF enabled, then we must skip the
     * last AOF, because this file is our currently writing. */
    if (server.aof_fd != -1) {
        ln = listNext(&li);
        serverAssert(ln != NULL);
    }

    /* Move aofInfo from 'incr_aof_list' to 'history_aof_list'. */
    while ((ln = listNext(&li)) != NULL) {
        aofInfo *ai = (aofInfo*)ln->value;
        serverAssert(ai->file_type == AOF_FILE_TYPE_INCR);

        aofInfo *hai = aofInfoDup(ai);
        hai->file_type = AOF_FILE_TYPE_HIST;
        listAddNodeHead(am->history_aof_list, hai);
        listDelNode(am->incr_aof_list, ln);
    }

    am->dirty = 1;
}

/* Write the formatted manifest string to disk. */
int writeAofManifestFile(sds buf) {
    int ret = C_OK;
    ssize_t nwritten;
    int len;

    sds am_name = getAofManifestFileName();
    sds am_filepath = makePath(server.aof_dirname, am_name);
    sds tmp_am_name = getTempAofManifestFileName();
    sds tmp_am_filepath = makePath(server.aof_dirname, tmp_am_name);

    int fd = open(tmp_am_filepath, O_WRONLY|O_TRUNC|O_CREAT, 0644);
    if (fd == -1) {
        serverLog(LL_WARNING, "Can't open the AOF manifest file %s: %s",
            tmp_am_name, strerror(errno));

        ret = C_ERR;
        goto cleanup;
    }

```

```

}

len = sdslen(buf);
while(len) {
    nwritten = write(fd, buf, len);

    if (nwritten < 0) {
        if (errno == EINTR) continue;

        serverLog(LL_WARNING, "Error trying to write the temporary AOF
manifest file %s: %s",
            tmp_am_name, strerror(errno));

        ret = C_ERR;
        goto cleanup;
    }

    len -= nwritten;
    buf += nwritten;
}

if (redis_fsync(fd) == -1) {
    serverLog(LL_WARNING, "Fail to fsync the temp AOF file %s: %s.",
        tmp_am_name, strerror(errno));

    ret = C_ERR;
    goto cleanup;
}

if (rename(tmp_am_filepath, am_filepath) != 0) {
    serverLog(LL_WARNING,
        "Error trying to rename the temporary AOF manifest file %s into %s:
%s",
        tmp_am_name, am_name, strerror(errno));

    ret = C_ERR;
}

cleanup:
    if (fd != -1) close(fd);
    sdsfree(am_name);
    sdsfree(am_filepath);
    sdsfree(tmp_am_name);
    sdsfree(tmp_am_filepath);
    return ret;
}

/* Persist the aofManifest information pointed to by am to disk. */
int persistAofManifest(aofManifest *am) {
    if (am->dirty == 0) {
        return C_OK;
    }
}

```

```

}

sds amstr = getAofManifestAsString(am);
int ret = writeAofManifestFile(amstr);
sdsfree(amstr);
if (ret == C_OK) am->dirty = 0;
return ret;
}

/* Called in `loadAppendOnlyFiles` when we upgrade from a old version redis.
 *
 * 1) Create AOF directory use 'server.aof_dirname' as the name.
 * 2) Use 'server.aof_filename' to construct a BASE type aofInfo and add it to
 *    aofManifest, then persist the manifest file to AOF directory.
 * 3) Move the old AOF file (server.aof_filename) to AOF directory.
 *
 * If any of the above steps fails or crash occurs, this will not cause any
 * problems, and redis will retry the upgrade process when it restarts.
 */
void aofUpgradePrepare(aofManifest *am) {
    serverAssert(!aofFileExist(server.aof_filename));

    /* Create AOF directory use 'server.aof_dirname' as the name. */
    if (dirCreateIfMissing(server.aof_dirname) == -1) {
        serverLog(LL_WARNING, "Can't open or create append-only dir %s: %s",
            server.aof_dirname, strerror(errno));
        exit(1);
    }

    /* Manually construct a BASE type aofInfo and add it to aofManifest. */
    if (am->base_aof_info) aofInfoFree(am->base_aof_info);
    aofInfo *ai = aofInfoCreate();
    ai->file_name = sdsnew(server.aof_filename);
    ai->file_seq = 1;
    ai->file_type = AOF_FILE_TYPE_BASE;
    am->base_aof_info = ai;
    am->curr_base_file_seq = 1;
    am->dirty = 1;

    /* Persist the manifest file to AOF directory. */
    if (persistAofManifest(am) != C_OK) {
        exit(1);
    }

    /* Move the old AOF file to AOF directory. */
    sds aof_filepath = makePath(server.aof_dirname, server.aof_filename);
    if (rename(server.aof_filename, aof_filepath) == -1) {
        serverLog(LL_WARNING,
            "Error trying to move the old AOF file %s into dir %s: %s",
            server.aof_filename,
            server.aof_dirname,

```

```

        strerror(errno));
        sdsfree(aof_filepath);
        exit(1);
    }
    sdsfree(aof_filepath);

    serverLog(LL_NOTICE, "Successfully migrated an old-style AOF file (%s) into
the AOF directory (%s).",
        server.aof_filename, server.aof_dirname);
}

/* When AOFRW success, the previous BASE and INCR AOFs will
 * become HISTORY type and be moved into 'history_aof_list'.
 *
 * The function will traverse the 'history_aof_list' and submit
 * the delete task to the bio thread.
 */
int aofDelHistoryFiles(void) {
    if (server.aof_manifest == NULL ||
        server.aof_disable_auto_gc == 1 ||
        !listLength(server.aof_manifest->history_aof_list))
    {
        return C_OK;
    }

    listNode *ln;
    listIter li;

    listRewind(server.aof_manifest->history_aof_list, &li);
    while ((ln = listNext(&li)) != NULL) {
        aofInfo *ai = (aofInfo*)ln->value;
        serverAssert(ai->file_type == AOF_FILE_TYPE_HIST);
        serverLog(LL_NOTICE, "Removing the history file %s in the background",
ai->file_name);
        sds aof_filepath = makePath(server.aof_dirname, ai->file_name);
        bg_unlink(aof_filepath);
        sdsfree(aof_filepath);
        listDelNode(server.aof_manifest->history_aof_list, ln);
    }

    server.aof_manifest->dirty = 1;
    return persistAofManifest(server.aof_manifest);
}

/* Used to clean up temp INCR AOF when AOFRW fails. */
void aofDelTempIncrAofFile() {
    sds aof_filename = getTempIncrAofName();
    sds aof_filepath = makePath(server.aof_dirname, aof_filename);
    serverLog(LL_NOTICE, "Removing the temp incr aof file %s in the
background", aof_filename);
    bg_unlink(aof_filepath);
}

```

```

    sdsfree(aof_filepath);
    sdsfree(aof_filename);
    return;
}

/* Called after `loadDataFromDisk` when redis start. If `server.aof_state` is
 * 'AOF_ON', It will do three things:
 * 1. Force create a BASE file when redis starts with an empty dataset
 * 2. Open the last opened INCR type AOF for writing, If not, create a new one
 * 3. Synchronously update the manifest file to the disk
 *
 * If any of the above steps fails, the redis process will exit.
 */
void aofOpenIfNeededOnServerStart(void) {
    if (server.aof_state != AOF_ON) {
        return;
    }

    serverAssert(server.aof_manifest != NULL);
    serverAssert(server.aof_fd == -1);

    if (dirCreateIfMissing(server.aof_dirname) == -1) {
        serverLog(LL_WARNING, "Can't open or create append-only dir %s: %s",
            server.aof_dirname, strerror(errno));
        exit(1);
    }

    /* If we start with an empty dataset, we will force create a BASE file. */
    if (!server.aof_manifest->base_aof_info &&
        !listLength(server.aof_manifest->incr_aof_list))
    {
        sds base_name =
getNewBaseFileNameAndMarkPreAsHistory(server.aof_manifest);
        sds base_filepath = makePath(server.aof_dirname, base_name);
        if (rewriteAppendOnlyFile(base_filepath) != C_OK) {
            exit(1);
        }
        sdsfree(base_filepath);
    }

    /* Because we will 'exit(1)' if open AOF or persistent manifest fails, so
     * we don't need atomic modification here. */
    sds aof_name = getLastIncrAofName(server.aof_manifest);

    /* Here we should use 'O_APPEND' flag. */
    sds aof_filepath = makePath(server.aof_dirname, aof_name);
    server.aof_fd = open(aof_filepath, O_WRONLY|O_APPEND|O_CREAT, 0644);
    sdsfree(aof_filepath);
    if (server.aof_fd == -1) {
        serverLog(LL_WARNING, "Can't open the append-only file %s: %s",
            aof_name, strerror(errno));
    }
}

```



```

        exit(1);
    }

    /* Persist our changes. */
    int ret = persistAofManifest(server.aof_manifest);
    if (ret != C_OK) {
        exit(1);
    }

    server.aof_last_incr_size = getAppendOnlyFileSize(aof_name, NULL);
}

int aofFileExist(char *filename) {
    sds file_path = makePath(server.aof_dirname, filename);
    int ret = fileExist(file_path);
    sdsfree(file_path);
    return ret;
}

/* Called in `rewriteAppendOnlyFileBackground`. If `server.aof_state`
 * is 'AOF_ON', It will do two things:
 * 1. Open a new INCR type AOF for writing
 * 2. Synchronously update the manifest file to the disk
 *
 * The above two steps of modification are atomic, that is, if
 * any step fails, the entire operation will rollback and returns
 * C_ERR, and if all succeeds, it returns C_OK.
 *
 * If `server.aof_state` is 'AOF_WAIT_REWRITE', It will open a temporary INCR
AOF
 * file to accumulate data during AOF_WAIT_REWRITE, and it will eventually be
 * renamed in the `backgroundRewriteDoneHandler` and written to the manifest
file.
 */
int openNewIncrAofForAppend(void) {
    serverAssert(server.aof_manifest != NULL);
    int newfd = -1;
    aofManifest *temp_am = NULL;
    sds new_aof_name = NULL;

    /* Only open new INCR AOF when AOF enabled. */
    if (server.aof_state == AOF_OFF) return C_OK;

    /* Open new AOF. */
    if (server.aof_state == AOF_WAIT_REWRITE) {
        /* Use a temporary INCR AOF file to accumulate data during
AOF_WAIT_REWRITE. */
        new_aof_name = getTempIncrAofName();
    } else {
        /* Dup a temp aof_manifest to modify. */
        temp_am = aofManifestDup(server.aof_manifest);
    }
}

```

```

        new_aof_name = sdsdup(getNewIncrAofName(temp_am));
    }
    sds new_aof_filepath = makePath(server.aof_dirname, new_aof_name);
    newfd = open(new_aof_filepath, O_WRONLY|O_TRUNC|O_CREAT, 0644);
    sdsfree(new_aof_filepath);
    if (newfd == -1) {
        serverLog(LL_WARNING, "Can't open the append-only file %s: %s",
            new_aof_name, strerror(errno));
        goto cleanup;
    }

    if (temp_am) {
        /* Persist AOF Manifest. */
        if (persistAofManifest(temp_am) == C_ERR) {
            goto cleanup;
        }
    }
    sdsfree(new_aof_name);

    /* If reaches here, we can safely modify the `server.aof_manifest`
     * and `server.aof_fd`. */

    /* Close old aof_fd if needed. */
    if (server.aof_fd != -1) bioCreateCloseJob(server.aof_fd);
    server.aof_fd = newfd;

    /* Reset the aof_last_incr_size. */
    server.aof_last_incr_size = 0;
    /* Update `server.aof_manifest`. */
    if (temp_am) aofManifestFreeAndUpdate(temp_am);
    return C_OK;

cleanup:
    if (new_aof_name) sdsfree(new_aof_name);
    if (newfd != -1) close(newfd);
    if (temp_am) aofManifestFree(temp_am);
    return C_ERR;
}

/* Whether to limit the execution of Background AOF rewrite.
 *
 * * At present, if AOFWR fails, redis will automatically retry. If it continues
 * * to fail, we may get a lot of very small INCR files. so we need an AOFWR
 * * limiting measure.
 *
 * * We can't directly use `server.aof_current_size` and
 * * `server.aof_last_incr_size`,
 * * because there may be no new writes after AOFWR fails.
 *
 * * So, we use time delay to achieve our goal. When AOFWR fails, we delay the
 * execution

```

```

* of the next AOFRW by 1 minute. If the next AOFRW also fails, it will be
delayed by 2
* minutes. The next is 4, 8, 16, the maximum delay is 60 minutes (1 hour).
*
* During the limit period, we can still use the 'bgrewriteaof' command to
execute AOFRW
* immediately.
*
* Return 1 means that AOFRW is limited and cannot be executed. 0 means that we
can execute
* AOFRW, which may be that we have reached the 'next_rewrite_time' or the
number of INCR
* AOFs has not reached the limit threshold.
* */
#define AOF_REWRITE_LIMITE_THRESHOLD    3
#define AOF_REWRITE_LIMITE_MAX_MINUTES 60 /* 1 hour */
int aofRewriteLimited(void) {
    static int next_delay_minutes = 0;
    static time_t next_rewrite_time = 0;

    if (server.stat_aofrw_consecutive_failures < AOF_REWRITE_LIMITE_THRESHOLD)
    {
        /* We may be recovering from limited state, so reset all states. */
        next_delay_minutes = 0;
        next_rewrite_time = 0;
        return 0;
    }

    /* if it is in the limiting state, then check if the next_rewrite_time is
reached */
    if (next_rewrite_time != 0) {
        if (server.unixtime < next_rewrite_time) {
            return 1;
        } else {
            next_rewrite_time = 0;
            return 0;
        }
    }

    next_delay_minutes = (next_delay_minutes == 0) ? 1 : (next_delay_minutes *
2);
    if (next_delay_minutes > AOF_REWRITE_LIMITE_MAX_MINUTES) {
        next_delay_minutes = AOF_REWRITE_LIMITE_MAX_MINUTES;
    }

    next_rewrite_time = server.unixtime + next_delay_minutes * 60;
    serverLog(LL_WARNING,
        "Background AOF rewrite has repeatedly failed and triggered the limit,
will retry in %d minutes", next_delay_minutes);
    return 1;
}

```

```

/* -----
 * AOF file implementation
 * ----- */

/* Return true if an Aof fsync is currently already in progress in a
 * BIO thread. */
int aofFsyncInProgress(void) {
    return bioPendingJobsOfType(BIO_AOF_FSYNC) != 0;
}

/* Starts a background task that performs fsync() against the specified
 * file descriptor (the one of the AOF file) in another thread. */
void aof_background_fsync(int fd) {
    bioCreateFsyncJob(fd);
}

/* Kills an AOFRW child process if exists */
void killAppendOnlyChild(void) {
    int statloc;
    /* No AOFRW child? return. */
    if (server.child_type != CHILD_TYPE_AOF) return;
    /* Kill AOFRW child, wait for child exit. */
    serverLog(LL_NOTICE,"Killing running AOF rewrite child: %ld",
        (long) server.child_pid);
    if (kill(server.child_pid,SIGUSR1) != -1) {
        while(waitpid(-1, &statloc, 0) != server.child_pid);
    }
    aofRemoveTempFile(server.child_pid);
    resetChildState();
    server.aof_rewrite_time_start = -1;
}

/* Called when the user switches from "appendonly yes" to "appendonly no"
 * at runtime using the CONFIG command. */
void stopAppendOnly(void) {
    serverAssert(server.aof_state != AOF_OFF);
    flushAppendOnlyFile(1);
    if (redis_fsync(server.aof_fd) == -1) {
        serverLog(LL_WARNING,"Fail to fsync the AOF file: %s",strerror(errno));
    } else {
        server.aof_fsync_offset = server.aof_current_size;
        server.aof_last_fsync = server.unixtime;
    }
    close(server.aof_fd);

    server.aof_fd = -1;
    server.aof_selected_db = -1;
    server.aof_state = AOF_OFF;
    server.aof_rewrite_scheduled = 0;
    server.aof_last_incr_size = 0;
}

```

```

    killAppendOnlyChild();
    sdsfree(server.aof_buf);
    server.aof_buf = sdsempty();
}

/* Called when the user switches from "appendonly no" to "appendonly yes"
 * at runtime using the CONFIG command. */
int startAppendOnly(void) {
    serverAssert(server.aof_state == AOF_OFF);

    server.aof_state = AOF_WAIT_REWRITE;
    if (hasActiveChildProcess() && server.child_type != CHILD_TYPE_AOF) {
        server.aof_rewrite_scheduled = 1;
        serverLog(LL_WARNING,"AOF was enabled but there is already another
background operation. An AOF background was scheduled to start when
possible.");
    } else if (server.in_exec){
        server.aof_rewrite_scheduled = 1;
        serverLog(LL_WARNING,"AOF was enabled during a transaction. An AOF
background was scheduled to start when possible.");
    } else {
        /* If there is a pending AOF rewrite, we need to switch it off and
         * start a new one: the old one cannot be reused because it is not
         * accumulating the AOF buffer. */
        if (server.child_type == CHILD_TYPE_AOF) {
            serverLog(LL_WARNING,"AOF was enabled but there is already an AOF
rewriting in background. Stopping background AOF and starting a rewrite now.");
            killAppendOnlyChild();
        }

        if (rewriteAppendOnlyFileBackground() == C_ERR) {
            server.aof_state = AOF_OFF;
            serverLog(LL_WARNING,"Redis needs to enable the AOF but can't
trigger a background AOF rewrite operation. Check the above logs for more info
about the error.");
            return C_ERR;
        }
    }
    server.aof_last_fsync = server.unixtime;
    /* If AOF fsync error in bio job, we just ignore it and log the event. */
    int aof_bio_fsync_status;
    atomicGet(server.aof_bio_fsync_status, aof_bio_fsync_status);
    if (aof_bio_fsync_status == C_ERR) {
        serverLog(LL_WARNING,
            "AOF reopen, just ignore the AOF fsync error in bio job");
        atomicSet(server.aof_bio_fsync_status,C_OK);
    }

    /* If AOF was in error state, we just ignore it and log the event. */
    if (server.aof_last_write_status == C_ERR) {
        serverLog(LL_WARNING,"AOF reopen, just ignore the last error.");
    }
}

```

```

        server.aof_last_write_status = C_OK;
    }
    return C_OK;
}

/* This is a wrapper to the write syscall in order to retry on short writes
 * or if the syscall gets interrupted. It could look strange that we retry
 * on short writes given that we are writing to a block device: normally if
 * the first call is short, there is a end-of-space condition, so the next
 * is likely to fail. However apparently in modern systems this is no longer
 * true, and in general it looks just more resilient to retry the write. If
 * there is an actual error condition we'll get it at the next try. */
ssize_t aofWrite(int fd, const char *buf, size_t len) {
    ssize_t nwritten = 0, totwritten = 0;

    while(len) {
        nwritten = write(fd, buf, len);

        if (nwritten < 0) {
            if (errno == EINTR) continue;
            return totwritten ? totwritten : -1;
        }

        len -= nwritten;
        buf += nwritten;
        totwritten += nwritten;
    }

    return totwritten;
}

/* Write the append only file buffer on disk.
 *
 * Since we are required to write the AOF before replying to the client,
 * and the only way the client socket can get a write is entering when
 * the event loop, we accumulate all the AOF writes in a memory
 * buffer and write it on disk using this function just before entering
 * the event loop again.
 *
 * About the 'force' argument:
 *
 * When the fsync policy is set to 'everysec' we may delay the flush if there
 * is still an fsync() going on in the background thread, since for instance
 * on Linux write(2) will be blocked by the background fsync anyway.
 * When this happens we remember that there is some aof buffer to be
 * flushed ASAP, and will try to do that in the serverCron() function.
 *
 * However if force is set to 1 we'll write regardless of the background
 * fsync. */
#define AOF_WRITE_LOG_ERROR_RATE 30 /* Seconds between errors logging. */
void flushAppendOnlyFile(int force) {

```

```

ssize_t nwritten;
int sync_in_progress = 0;
mstime_t latency;

if (sdslen(server.aof_buf) == 0) {
    /* Check if we need to do fsync even the aof buffer is empty,
     * because previously in AOF_FSYNC_EVERYSEC mode, fsync is
     * called only when aof buffer is not empty, so if users
     * stop write commands before fsync called in one second,
     * the data in page cache cannot be flushed in time. */
    if (server.aof_fsync == AOF_FSYNC_EVERYSEC &&
        server.aof_fsync_offset != server.aof_current_size &&
        server.unixtime > server.aof_last_fsync &&
        !(sync_in_progress = aofFsyncInProgress())) {
        goto try_fsync;
    } else {
        return;
    }
}

if (server.aof_fsync == AOF_FSYNC_EVERYSEC)
    sync_in_progress = aofFsyncInProgress();

if (server.aof_fsync == AOF_FSYNC_EVERYSEC && !force) {
    /* With this append fsync policy we do background fsyncing.
     * If the fsync is still in progress we can try to delay
     * the write for a couple of seconds. */
    if (sync_in_progress) {
        if (server.aof_flush_postponed_start == 0) {
            /* No previous write postponing, remember that we are
             * postponing the flush and return. */
            server.aof_flush_postponed_start = server.unixtime;
            return;
        } else if (server.unixtime - server.aof_flush_postponed_start < 2)
        {
            /* We were already waiting for fsync to finish, but for less
             * than two seconds this is still ok. Postpone again. */
            return;
        }
        /* Otherwise fall through, and go write since we can't wait
         * over two seconds. */
        server.aof_delayed_fsync++;
        serverLog(LL_NOTICE,"Asynchronous AOF fsync is taking too long
(disk is busy?). Writing the AOF buffer without waiting for fsync to complete,
this may slow down Redis.");
    }
}

/* We want to perform a single write. This should be guaranteed atomic
 * at least if the filesystem we are writing is a real physical one.
 * While this will save us against the server being killed I don't think
 * there is much to do about the whole server stopping for power problems

```

```

    * or alike */

if (server.aof_flush_sleep && sdslen(server.aof_buf)) {
    usleep(server.aof_flush_sleep);
}

latencyStartMonitor(latency);
nwritten = aofWrite(server.aof_fd,server.aof_buf,sdslen(server.aof_buf));
latencyEndMonitor(latency);
/* We want to capture different events for delayed writes:
 * when the delay happens with a pending fsync, or with a saving child
 * active, and when the above two conditions are missing.
 * We also use an additional event name to save all samples which is
 * useful for graphing / monitoring purposes. */
if (sync_in_progress) {
    latencyAddSampleIfNeeded("aof-write-pending-fsync",latency);
} else if (hasActiveChildProcess()) {
    latencyAddSampleIfNeeded("aof-write-active-child",latency);
} else {
    latencyAddSampleIfNeeded("aof-write-alone",latency);
}
latencyAddSampleIfNeeded("aof-write",latency);

/* We performed the write so reset the postponed flush sentinel to zero. */
server.aof_flush_postponed_start = 0;

if (nwritten != (ssize_t)sdslen(server.aof_buf)) {
    static time_t last_write_error_log = 0;
    int can_log = 0;

    /* Limit logging rate to 1 line per AOF_WRITE_LOG_ERROR_RATE seconds.
    */
    if ((server.unixtime - last_write_error_log) >
AOF_WRITE_LOG_ERROR_RATE) {
        can_log = 1;
        last_write_error_log = server.unixtime;
    }

    /* Log the AOF write error and record the error code. */
    if (nwritten == -1) {
        if (can_log) {
            serverLog(LL_WARNING,"Error writing to the AOF file: %s",
                strerror(errno));
            server.aof_last_write_errno = errno;
        }
    } else {
        if (can_log) {
            serverLog(LL_WARNING,"Short write while writing to "
                "the AOF file: (nwritten=%lld, "
                "expected=%lld)",
                (long long)nwritten,

```



```

        (long long)sdslen(server.aof_buf));
    }

    if (ftruncate(server.aof_fd, server.aof_last_incr_size) == -1) {
        if (can_log) {
            serverLog(LL_WARNING, "Could not remove short write "
                "from the append-only file. Redis may refuse "
                "to load the AOF the next time it starts. "
                "ftruncate: %s", strerror(errno));
        }
    } else {
        /* If the ftruncate() succeeded we can set nwritten to
         * -1 since there is no longer partial data into the AOF. */
        nwritten = -1;
    }
    server.aof_last_write_errno = ENOSPC;
}

/* Handle the AOF write error. */
if (server.aof_fsync == AOF_FSYNC_ALWAYS) {
    /* We can't recover when the fsync policy is ALWAYS since the reply
     * for the client is already in the output buffers (both writes and
     * reads), and the changes to the db can't be rolled back. Since we
     * have a contract with the user that on acknowledged or observed
     * writes are is synced on disk, we must exit. */
    serverLog(LL_WARNING, "Can't recover from AOF write error when the
AOF fsync policy is 'always'. Exiting...");
    exit(1);
} else {
    /* Recover from failed write leaving data into the buffer. However
     * set an error to stop accepting writes as long as the error
     * condition is not cleared. */
    server.aof_last_write_status = C_ERR;

    /* Trim the sds buffer if there was a partial write, and there
     * was no way to undo it with ftruncate(2). */
    if (nwritten > 0) {
        server.aof_current_size += nwritten;
        server.aof_last_incr_size += nwritten;
        sdsrange(server.aof_buf, nwritten, -1);
    }
    return; /* We'll try again on the next call... */
}
} else {
    /* Successful write(2). If AOF was in error state, restore the
     * OK state and log the event. */
    if (server.aof_last_write_status == C_ERR) {
        serverLog(LL_WARNING,
            "AOF write error looks solved, Redis can write again.");
        server.aof_last_write_status = C_OK;
    }
}

```

```

}
server.aof_current_size += nwritten;
server.aof_last_incr_size += nwritten;

/* Re-use AOF buffer when it is small enough. The maximum comes from the
 * arena size of 4k minus some overhead (but is otherwise arbitrary). */
if ((sdslen(server.aof_buf)+sdsavail(server.aof_buf)) < 4000) {
    sdsclear(server.aof_buf);
} else {
    sdsfree(server.aof_buf);
    server.aof_buf = sdsempty();
}

try_fsync:
/* Don't fsync if no-appendfsync-on-rewrite is set to yes and there are
 * children doing I/O in the background. */
if (server.aof_no_fsync_on_rewrite && hasActiveChildProcess())
    return;

/* Perform the fsync if needed. */
if (server.aof_fsync == AOF_FSYNC_ALWAYS) {
    /* redis_fsync is defined as fdatsync() for Linux in order to avoid
     * flushing metadata. */
    latencyStartMonitor(latency);
    /* Let's try to get this data on the disk. To guarantee data safe when
     * the AOF fsync policy is 'always', we should exit if failed to fsync
     * AOF (see comment next to the exit(1) after write error above). */
    if (redis_fsync(server.aof_fd) == -1) {
        serverLog(LL_WARNING,"Can't persist AOF for fsync error when the "
            "AOF fsync policy is 'always': %s. Exiting...", strerror(errno));
        exit(1);
    }
    latencyEndMonitor(latency);
    latencyAddSampleIfNeeded("aof-fsync-always",latency);
    server.aof_fsync_offset = server.aof_current_size;
    server.aof_last_fsync = server.unixtime;
} else if ((server.aof_fsync == AOF_FSYNC_EVERYSEC &&
    server.unixtime > server.aof_last_fsync)) {
    if (!sync_in_progress) {
        aof_background_fsync(server.aof_fd);
        server.aof_fsync_offset = server.aof_current_size;
    }
    server.aof_last_fsync = server.unixtime;
}
}

sds catAppendOnlyGenericCommand(sds dst, int argc, robj **argv) {
    char buf[32];
    int len, j;
    robj *o;

```

```

    buf[0] = '*';
    len = 1+ll2string(buf+1,sizeof(buf)-1,argc);
    buf[len++] = '\r';
    buf[len++] = '\n';
    dst = sdscatlen(dst,buf,len);

    for (j = 0; j < argc; j++) {
        o = getDecodedObject(argv[j]);
        buf[0] = '{{content}}#x27;;
        len = 1+ll2string(buf+1,sizeof(buf)-1,sdslen(o->ptr));
        buf[len++] = '\r';
        buf[len++] = '\n';
        dst = sdscatlen(dst,buf,len);
        dst = sdscatlen(dst,o->ptr,sdslen(o->ptr));
        dst = sdscatlen(dst,"\r\n",2);
        decrRefCount(o);
    }
    return dst;
}

/* Generate a piece of timestamp annotation for AOF if current record timestamp
 * in AOF is not equal server unix time. If we specify 'force' argument to 1,
 * we would generate one without check, currently, it is useful in AOF
rewriting
 * child process which always needs to record one timestamp at the beginning of
 * rewriting AOF.
 *
 * Timestamp annotation format is "#TS:${timestamp}\r\n". "TS" is short of
 * timestamp and this method could save extra bytes in AOF. */
sds genAofTimestampAnnotationIfNeeded(int force) {
    sds ts = NULL;

    if (force || server.aof_cur_timestamp < server.unixtime) {
        server.aof_cur_timestamp = force ? time(NULL) : server.unixtime;
        ts = sdscatfmt(sdsempty(), "#TS:%I\r\n", server.aof_cur_timestamp);
        serverAssert(sdslen(ts) <= AOF_ANNOTATION_LINE_MAX_LEN);
    }
    return ts;
}

void feedAppendOnlyFile(int dictid, robj **argv, int argc) {
    sds buf = sdsempty();

    serverAssert(dictid >= 0 && dictid < server.dbnum);

    /* Feed timestamp if needed */
    if (server.aof_timestamp_enabled) {
        sds ts = genAofTimestampAnnotationIfNeeded(0);
        if (ts != NULL) {
            buf = sdscatsds(buf, ts);
            sdsfree(ts);
        }
    }
}

```

```

    }
}

/* The DB this command was targeting is not the same as the last command
 * we appended. To issue a SELECT command is needed. */
if (dictid != server.aof_selected_db) {
    char seldb[64];

    snprintf(seldb, sizeof(seldb), "%d", dictid);
    buf = sdscatprintf(buf, "*2\r\n$6\r\nSELECT\r\n$%lu\r\n%s\r\n",
        (unsigned long)strlen(seldb), seldb);
    server.aof_selected_db = dictid;
}

/* All commands should be propagated the same way in AOF as in replication.
 * No need for AOF-specific translation. */
buf = catAppendOnlyGenericCommand(buf, argc, argv);

/* Append to the AOF buffer. This will be flushed on disk just before
 * of re-entering the event loop, so before the client will get a
 * positive reply about the operation performed. */
if (server.aof_state == AOF_ON ||
    (server.aof_state == AOF_WAIT_REWRITE && server.child_type ==
CHILD_TYPE_AOF))
{
    server.aof_buf = sdscatlen(server.aof_buf, buf, sdslen(buf));
}

sdsfree(buf);
}

/* -----
 * AOF loading
 * ----- */

/* In Redis commands are always executed in the context of a client, so in
 * order to load the append only file we need to create a fake client. */
struct client *createAOFClient(void) {
    struct client *c = createClient(NULL);

    c->id = CLIENT_ID_AOF; /* So modules can identify it's the AOF client. */

    /*
     * The AOF client should never be blocked (unlike master
     * replication connection).
     * This is because blocking the AOF client might cause
     * deadlock (because potentially no one will unblock it).
     * Also, if the AOF client will be blocked just for
     * background processing there is a chance that the
     * command execution order will be violated.
     */
}

```

```

c->flags = CLIENT_DENY_BLOCKING;

/* We set the fake client as a slave waiting for the synchronization
 * so that Redis will not try to send replies to this client. */
c->replstate = SLAVE_STATE_WAIT_BGSAVE_START;
return c;
}

/* Replay an append log file. On success AOF_OK or AOF_TRUNCATED is returned,
 * otherwise, one of the following is returned:
 * AOF_OPEN_ERR: Failed to open the AOF file.
 * AOF_NOT_EXIST: AOF file doesn't exist.
 * AOF_EMPTY: The AOF file is empty (nothing to load).
 * AOF_FAILED: Failed to load the AOF file. */
int loadSingleAppendOnlyFile(char *filename) {
    struct client *fakeClient;
    struct redis_stat sb;
    int old_aof_state = server.aof_state;
    long loops = 0;
    off_t valid_up_to = 0; /* Offset of latest well-formed command loaded. */
    off_t valid_before_multi = 0; /* Offset before MULTI command loaded. */
    off_t last_progress_report_size = 0;
    int ret = C_OK;

    sds aof_filepath = makePath(server.aof_dirname, filename);
    FILE *fp = fopen(aof_filepath, "r");
    if (fp == NULL) {
        int en = errno;
        if (redis_stat(aof_filepath, &sb) == 0 || errno != ENOENT) {
            serverLog(LL_WARNING, "Fatal error: can't open the append log file
%s for reading: %s", filename, strerror(en));
            sdsfree(aof_filepath);
            return AOF_OPEN_ERR;
        } else {
            serverLog(LL_WARNING, "The append log file %s doesn't exist: %s",
filename, strerror(errno));
            sdsfree(aof_filepath);
            return AOF_NOT_EXIST;
        }
    }

    if (fp && redis_fstat(fileno(fp), &sb) != -1 && sb.st_size == 0) {
        fclose(fp);
        sdsfree(aof_filepath);
        return AOF_EMPTY;
    }

    /* Temporarily disable AOF, to prevent EXEC from feeding a MULTI
     * to the same file we're about to read. */
    server.aof_state = AOF_OFF;

```

```

client *old_client = server.current_client;
fakeClient = server.current_client = createAOFClient();

/* Check if the AOF file is in RDB format (it may be RDB encoded base AOF
 * or old style RDB-preamble AOF). In that case we need to load the RDB
file
 * and later continue loading the AOF tail if it is an old style RDB-
preamble AOF. */
char sig[5]; /* "REDIS" */
if (fread(sig,1,5,fp) != 5 || memcmp(sig,"REDIS",5) != 0) {
    /* Not in RDB format, seek back at 0 offset. */
    if (fseek(fp,0,SEEK_SET) == -1) goto readerr;
} else {
    /* RDB format. Pass loading the RDB functions. */
    rio rdb;
    int old_style = !strcmp(filename, server.aof_filename);
    if (old_style)
        serverLog(LL_NOTICE, "Reading RDB preamble from AOF file...");
    else
        serverLog(LL_NOTICE, "Reading RDB base file on AOF loading...");

    if (fseek(fp,0,SEEK_SET) == -1) goto readerr;
    rioInitWithFile(&rdb,fp);
    if (rdbLoadRio(&rdb,RDBFLAGS_AOF_PREAMBLE,NULL) != C_OK) {
        if (old_style)
            serverLog(LL_WARNING, "Error reading the RDB preamble of the
AOF file %s, AOF loading aborted", filename);
        else
            serverLog(LL_WARNING, "Error reading the RDB base file %s, AOF
loading aborted", filename);

        goto readerr;
    } else {
        loadingAbsProgress(ftello(fp));
        last_progress_report_size = ftello(fp);
        if (old_style) serverLog(LL_NOTICE, "Reading the remaining AOF
tail...");
    }
}

/* Read the actual AOF file, in REPL format, command by command. */
while(1) {
    int argc, j;
    unsigned long len;
    robj **argv;
    char buf[AOF_ANNOTATION_LINE_MAX_LEN];
    sds argsds;
    struct redisCommand *cmd;

    /* Serve the clients from time to time */
    if (!(loops++ % 1024)) {

```

```

        off_t progress_delta = ftello(fp) - last_progress_report_size;
        loadingIncrProgress(progress_delta);
        last_progress_report_size += progress_delta;
        processEventsWhileBlocked();
        processModuleLoadingProgressEvent(1);
    }
    if (fgets(buf, sizeof(buf), fp) == NULL) {
        if (feof(fp)) {
            break;
        } else {
            goto readerr;
        }
    }
    if (buf[0] == '#') continue; /* Skip annotations */
    if (buf[0] != '*') goto fmterr;
    if (buf[1] == '\\0') goto readerr;
    argc = atoi(buf+1);
    if (argc < 1) goto fmterr;
    if ((size_t)argc > SIZE_MAX / sizeof(robj*)) goto fmterr;

    /* Load the next command in the AOF as our fake client
     * argv. */
    argv = zmalloc(sizeof(robj*)*argc);
    fakeClient->argc = argc;
    fakeClient->argv = argv;
    fakeClient->argv_len = argc;

    for (j = 0; j < argc; j++) {
        /* Parse the argument len. */
        char *readres = fgets(buf, sizeof(buf), fp);
        if (readres == NULL || buf[0] != '{{content}}#x27;) {
            fakeClient->argc = j; /* Free up to j-1. */
            freeClientArgv(fakeClient);
            if (readres == NULL)
                goto readerr;
            else
                goto fmterr;
        }
        len = strtol(buf+1, NULL, 10);

        /* Read it into a string object. */
        argsds = sdsnewlen(SDS_NOINIT, len);
        if (len && fread(argsds, len, 1, fp) == 0) {
            sdsfree(argsds);
            fakeClient->argc = j; /* Free up to j-1. */
            freeClientArgv(fakeClient);
            goto readerr;
        }
        argv[j] = createObject(OBJ_STRING, argsds);

        /* Discard CRLF. */
    }

```

```

        if (fread(buf,2,1,fp) == 0) {
            fakeClient->argc = j+1; /* Free up to j. */
            freeClientArgv(fakeClient);
            goto readerr;
        }
    }

    /* Command lookup */
    cmd = lookupCommand(argv,argc);
    if (!cmd) {
        serverLog(LL_WARNING,
            "Unknown command '%s' reading the append only file %s",
            (char*)argv[0]->ptr, filename);
        freeClientArgv(fakeClient);
        ret = AOF_FAILED;
        goto cleanup;
    }

    if (cmd->proc == multiCommand) valid_before_multi = valid_up_to;

    /* Run the command in the context of a fake client */
    fakeClient->cmd = fakeClient->lastcmd = cmd;
    if (fakeClient->flags & CLIENT_MULTI &&
        fakeClient->cmd->proc != execCommand)
    {
        queueMultiCommand(fakeClient);
    } else {
        cmd->proc(fakeClient);
    }

    /* The fake client should not have a reply */
    serverAssert(fakeClient->bufpos == 0 &&
        listLength(fakeClient->reply) == 0);

    /* The fake client should never get blocked */
    serverAssert((fakeClient->flags & CLIENT_BLOCKED) == 0);

    /* Clean up. Command code may have changed argv/argc so we use the
     * argv/argc of the client instead of the local variables. */
    freeClientArgv(fakeClient);
    if (server.aof_load_truncated) valid_up_to = ftello(fp);
    if (server.key_load_delay)
        debugDelay(server.key_load_delay);
}

/* This point can only be reached when EOF is reached without errors.
 * If the client is in the middle of a MULTI/EXEC, handle it as it was
 * a short read, even if technically the protocol is correct: we want
 * to remove the unprocessed tail and continue. */
if (fakeClient->flags & CLIENT_MULTI) {
    serverLog(LL_WARNING,

```



```

        "Revert incomplete MULTI/EXEC transaction in AOF file %s",
filename);
    valid_up_to = valid_before_multi;
    goto uxeof;
}

loaded_ok: /* DB loaded, cleanup and return C_OK to the caller. */
    loadingIncrProgress(ftello(fp) - last_progress_report_size);
    server.aof_state = old_aof_state;
    goto cleanup;

readerr: /* Read error. If feof(fp) is true, fall through to unexpected EOF. */
    if (!feof(fp)) {
        serverLog(LL_WARNING,"Unrecoverable error reading the append only file
%s: %s", filename, strerror(errno));
        ret = AOF_FAILED;
        goto cleanup;
    }

uxeof: /* Unexpected AOF end of file. */
    if (server.aof_load_truncated) {
        serverLog(LL_WARNING,"!!! Warning: short read while loading the AOF
file %s!!!", filename);
        serverLog(LL_WARNING,"!!! Truncating the AOF %s at offset %llu !!!",
filename, (unsigned long long) valid_up_to);
        if (valid_up_to == -1 || truncate(aof_filepath,valid_up_to) == -1) {
            if (valid_up_to == -1) {
                serverLog(LL_WARNING,"Last valid command offset is invalid");
            } else {
                serverLog(LL_WARNING,"Error truncating the AOF file %s: %s",
filename, strerror(errno));
            }
        } else {
            /* Make sure the AOF file descriptor points to the end of the
            * file after the truncate call. */
            if (server.aof_fd != -1 && lseek(server.aof_fd,0,SEEK_END) == -1) {
                serverLog(LL_WARNING,"Can't seek the end of the AOF file %s:
%s",
filename, strerror(errno));
            } else {
                serverLog(LL_WARNING,
                "AOF %s loaded anyway because aof-load-truncated is
enabled", filename);
                ret = AOF_TRUNCATED;
                goto loaded_ok;
            }
        }
    }
    serverLog(LL_WARNING, "Unexpected end of file reading the append only file
%s. You can: "
    "1) Make a backup of your AOF file, then use ./redis-check-aof --fix

```

```

<filename.manifest>. "
    "2) Alternatively you can set the 'aof-load-truncated' configuration
option to yes and restart the server.", filename);
    ret = AOF_FAILED;
    goto cleanup;

fmtterr: /* Format error. */
    serverLog(LL_WARNING, "Bad file format reading the append only file %s: "
        "make a backup of your AOF file, then use ./redis-check-aof --fix
<filename.manifest>", filename);
    ret = AOF_FAILED;
    /* fall through to cleanup. */

cleanup:
    if (fakeClient) freeClient(fakeClient);
    server.current_client = old_client;
    fclose(fp);
    sdsfree(aof_filepath);
    return ret;
}

/* Load the AOF files according the aofManifest pointed by am. */
int loadAppendOnlyFiles(aofManifest *am) {
    serverAssert(am != NULL);
    int status, ret = C_OK;
    long long start;
    off_t total_size = 0, base_size = 0;
    sds aof_name;
    int total_num, aof_num = 0, last_file;

    /* If the 'server.aof_filename' file exists in dir, we may be starting
    * from an old redis version. We will use enter upgrade mode in three
    situations.
    *
    * 1. If the 'server.aof_dirname' directory not exist
    * 2. If the 'server.aof_dirname' directory exists but the manifest file is
    missing
    * 3. If the 'server.aof_dirname' directory exists and the manifest file it
    contains
    *    has only one base AOF record, and the file name of this base AOF is
    'server.aof_filename',
    *    and the 'server.aof_filename' file not exist in 'server.aof_dirname'
    directory
    * */
    if (fileExist(server.aof_filename)) {
        if (!dirExists(server.aof_dirname) ||
            (am->base_aof_info == NULL && listLength(am->incr_aof_list) == 0)
        ||
            (am->base_aof_info != NULL && listLength(am->incr_aof_list) == 0 &&
                !strcmp(am->base_aof_info->file_name, server.aof_filename) &&
                !aofFileExist(server.aof_filename)))

```

```

    {
        aofUpgradePrepare(am);
    }
}

if (am->base_aof_info == NULL && listLength(am->incr_aof_list) == 0) {
    return AOF_NOT_EXIST;
}

total_num = getBaseAndIncrAppendOnlyFilesNum(am);
serverAssert(total_num > 0);

/* Here we calculate the total size of all BASE and INCR files in
 * advance, it will be set to `server.loading_total_bytes`. */
total_size = getBaseAndIncrAppendOnlyFileSize(am, &status);
if (status != AOF_OK) {
    /* If an AOF exists in the manifest but not on the disk, we consider
this to be a fatal error. */
    if (status == AOF_NOT_EXIST) status = AOF_FAILED;

    return status;
} else if (total_size == 0) {
    return AOF_EMPTY;
}

startLoading(total_size, RDBFLAGS_AOF_PREAMBLE, 0);

/* Load BASE AOF if needed. */
if (am->base_aof_info) {
    serverAssert(am->base_aof_info->file_type == AOF_FILE_TYPE_BASE);
    aof_name = (char*)am->base_aof_info->file_name;
    updateLoadingFileName(aof_name);
    base_size = getAppendOnlyFileSize(aof_name, NULL);
    last_file = ++aof_num == total_num;
    start = ustime();
    ret = loadSingleAppendOnlyFile(aof_name);
    if (ret == AOF_OK || (ret == AOF_TRUNCATED && last_file)) {
        serverLog(LL_NOTICE, "DB loaded from base file %s: %.3f seconds",
            aof_name, (float)(ustime()-start)/1000000);
    }

    /* If the truncated file is not the last file, we consider this to be a
fatal error. */
    if (ret == AOF_TRUNCATED && !last_file) {
        ret = AOF_FAILED;
    }

    if (ret == AOF_OPEN_ERR || ret == AOF_FAILED) {
        goto cleanup;
    }
}
}

```

```

/* Load INCR AOFs if needed. */
if (listLength(am->incr_aof_list)) {
    listNode *ln;
    listIter li;

    listRewind(am->incr_aof_list, &li);
    while ((ln = listNext(&li)) != NULL) {
        aofInfo *ai = (aofInfo*)ln->value;
        serverAssert(ai->file_type == AOF_FILE_TYPE_INCR);
        aof_name = (char*)ai->file_name;
        updateLoadingFileName(aof_name);
        last_file = ++aof_num == total_num;
        start = ustime();
        ret = loadSingleAppendOnlyFile(aof_name);
        if (ret == AOF_OK || (ret == AOF_TRUNCATED && last_file)) {
            serverLog(LL_NOTICE, "DB loaded from incr file %s: %.3f
seconds",
                    aof_name, (float)(ustime()-start)/1000000);
        }

        /* We know that (at least) one of the AOF files has data
(total_size > 0),
        * so empty incr AOF file doesn't count as a AOF_EMPTY result */
        if (ret == AOF_EMPTY) ret = AOF_OK;

        if (ret == AOF_TRUNCATED && !last_file) {
            ret = AOF_FAILED;
        }

        if (ret == AOF_OPEN_ERR || ret == AOF_FAILED) {
            goto cleanup;
        }
    }
}

server.aof_current_size = total_size;
/* Ideally, the aof_rewrite_base_size variable should hold the size of the
* AOF when the last rewrite ended, this should include the size of the
* incremental file that was created during the rewrite since otherwise we
* risk the next automatic rewrite to happen too soon (or immediately if
* auto-aof-rewrite-percentage is low). However, since we do not persist
* aof_rewrite_base_size information anywhere, we initialize it on restart
* to the size of BASE AOF file. This might cause the first AOFRW to be
* executed early, but that shouldn't be a problem since everything will be
* fine after the first AOFRW. */
server.aof_rewrite_base_size = base_size;
server.aof_fsync_offset = server.aof_current_size;

cleanup:
stopLoading(ret == AOF_OK || ret == AOF_TRUNCATED);

```

```

    return ret;
}

/* -----
 * AOF rewrite
 * ----- */

/* Delegate writing an object to writing a bulk string or bulk long long.
 * This is not placed in rio.c since that adds the server.h dependency. */
int rioWriteBulkObject(rio *r, robj *obj) {
    /* Avoid using getDecodedObject to help copy-on-write (we are often
     * in a child process when this function is called). */
    if (obj->encoding == OBJ_ENCODING_INT) {
        return rioWriteBulkLongLong(r, (long)obj->ptr);
    } else if (sdsEncodedObject(obj)) {
        return rioWriteBulkString(r, obj->ptr, sdslen(obj->ptr));
    } else {
        serverPanic("Unknown string encoding");
    }
}

/* Emit the commands needed to rebuild a list object.
 * The function returns 0 on error, 1 on success. */
int rewriteListObject(rio *r, robj *key, robj *o) {
    long long count = 0, items = listTypeLength(o);

    if (o->encoding == OBJ_ENCODING_QUICKLIST) {
        quicklist *list = o->ptr;
        quicklistIter *li = quicklistGetIterator(list, AL_START_HEAD);
        quicklistEntry entry;

        while (quicklistNext(li, &entry)) {
            if (count == 0) {
                int cmd_items = (items > AOF_REWRITE_ITEMS_PER_CMD) ?
                    AOF_REWRITE_ITEMS_PER_CMD : items;
                if (!rioWriteBulkCount(r, '*', 2+cmd_items) ||
                    !rioWriteBulkString(r, "RPUSH", 5) ||
                    !rioWriteBulkObject(r, key))
                {
                    quicklistReleaseIterator(li);
                    return 0;
                }
            }

            if (entry.value) {
                if (!rioWriteBulkString(r, (char*)entry.value, entry.sz)) {
                    quicklistReleaseIterator(li);
                    return 0;
                }
            } else {
                if (!rioWriteBulkLongLong(r, entry.longval)) {

```



```

        !rioWriteBulkObject(r,key))
    {
        dictReleaseIterator(di);
        return 0;
    }
}
if (!rioWriteBulkString(r,ele,sdslen(ele))) {
    dictReleaseIterator(di);
    return 0;
}
if (++count == AOF_REWRITE_ITEMS_PER_CMD) count = 0;
items--;
}
dictReleaseIterator(di);
} else {
    serverPanic("Unknown set encoding");
}
return 1;
}

/* Emit the commands needed to rebuild a sorted set object.
 * The function returns 0 on error, 1 on success. */
int rewriteSortedSetObject(rio *r, robj *key, robj *o) {
    long long count = 0, items = zsetLength(o);

    if (o->encoding == OBJ_ENCODING_LISTPACK) {
        unsigned char *zl = o->ptr;
        unsigned char *eptr, *sptr;
        unsigned char *vstr;
        unsigned int vlen;
        long long vll;
        double score;

        eptr = lpSeek(zl,0);
        serverAssert(eptr != NULL);
        sptr = lpNext(zl,eptr);
        serverAssert(sptr != NULL);

        while (eptr != NULL) {
            vstr = lpGetValue(eptr,&vlen,&vll);
            score = zzlGetScore(sptr);

            if (count == 0) {
                int cmd_items = (items > AOF_REWRITE_ITEMS_PER_CMD) ?
                    AOF_REWRITE_ITEMS_PER_CMD : items;

                if (!rioWriteBulkCount(r,'*',2+cmd_items*2) ||
                    !rioWriteBulkString(r,"ZADD",4) ||
                    !rioWriteBulkObject(r,key))
                {
                    return 0;
                }
            }
        }
    }
}

```

```

    }
}
if (!rioWriteBulkDouble(r,score)) return 0;
if (vstr != NULL) {
    if (!rioWriteBulkString(r,(char*)vstr,vlen)) return 0;
} else {
    if (!rioWriteBulkLongLong(r,vll)) return 0;
}
zzlNext(zl,&eptr,&sptr);
if (++count == AOF_REWRITE_ITEMS_PER_CMD) count = 0;
items--;
}
} else if (o->encoding == OBJ_ENCODING_SKIPLIST) {
    zset *zs = o->ptr;
    dictIterator *di = dictGetIterator(zs->dict);
    dictEntry *de;

    while((de = dictNext(di)) != NULL) {
        sds ele = dictGetKey(de);
        double *score = dictGetVal(de);

        if (count == 0) {
            int cmd_items = (items > AOF_REWRITE_ITEMS_PER_CMD) ?
                AOF_REWRITE_ITEMS_PER_CMD : items;

            if (!rioWriteBulkCount(r,'*',2+cmd_items*2) ||
                !rioWriteBulkString(r,"ZADD",4) ||
                !rioWriteBulkObject(r,key))
            {
                dictReleaseIterator(di);
                return 0;
            }
        }
        if (!rioWriteBulkDouble(r,*score) ||
            !rioWriteBulkString(r,ele,sdslen(ele)))
        {
            dictReleaseIterator(di);
            return 0;
        }
        if (++count == AOF_REWRITE_ITEMS_PER_CMD) count = 0;
        items--;
    }
    dictReleaseIterator(di);
} else {
    serverPanic("Unknown sorted zset encoding");
}
return 1;
}

```

/* Write either the key or the value of the currently selected item of a hash.
 * The 'hi' argument passes a valid Redis hash iterator.


```

* The 'what' field specifies if to write a key or a value and can be
* either OBJ_HASH_KEY or OBJ_HASH_VALUE.
*
* The function returns 0 on error, non-zero on success. */
static int rioWriteHashIteratorCursor(rio *r, hashTypeIterator *hi, int what) {
    if (hi->encoding == OBJ_ENCODING_LISTPACK) {
        unsigned char *vstr = NULL;
        unsigned int vlen = UINT_MAX;
        long long vll = LLONG_MAX;

        hashTypeCurrentFromListpack(hi, what, &vstr, &vlen, &vll);
        if (vstr)
            return rioWriteBulkString(r, (char*)vstr, vlen);
        else
            return rioWriteBulkLongLong(r, vll);
    } else if (hi->encoding == OBJ_ENCODING_HT) {
        sds value = hashTypeCurrentFromHashTable(hi, what);
        return rioWriteBulkString(r, value, sdslen(value));
    }

    serverPanic("Unknown hash encoding");
    return 0;
}

/* Emit the commands needed to rebuild a hash object.
* The function returns 0 on error, 1 on success. */
int rewriteHashObject(rio *r, robj *key, robj *o) {
    hashTypeIterator *hi;
    long long count = 0, items = hashTypeLength(o);

    hi = hashTypeInitIterator(o);
    while (hashTypeNext(hi) != C_ERR) {
        if (count == 0) {
            int cmd_items = (items > AOF_REWRITE_ITEMS_PER_CMD) ?
                AOF_REWRITE_ITEMS_PER_CMD : items;

            if (!rioWriteBulkCount(r, '*', 2+cmd_items*2) ||
                !rioWriteBulkString(r, "HMSET", 5) ||
                !rioWriteBulkObject(r, key))
            {
                hashTypeReleaseIterator(hi);
                return 0;
            }
        }

        if (!rioWriteHashIteratorCursor(r, hi, OBJ_HASH_KEY) ||
            !rioWriteHashIteratorCursor(r, hi, OBJ_HASH_VALUE))
        {
            hashTypeReleaseIterator(hi);
            return 0;
        }
    }
}

```

```

        if (++count == AOF_REWRITE_ITEMS_PER_CMD) count = 0;
        items--;
    }

    hashTypeReleaseIterator(hi);

    return 1;
}

/* Helper for rewriteStreamObject() that generates a bulk string into the
 * AOF representing the ID 'id'. */
int rioWriteBulkStreamID(rio *r, streamID *id) {
    int retval;

    sds replyid = sdscatfmt(sdsempty(), "%U-%U", id->ms, id->seq);
    retval = rioWriteBulkString(r, replyid, sdslen(replyid));
    sdsfree(replyid);
    return retval;
}

/* Helper for rewriteStreamObject(): emit the XCLAIM needed in order to
 * add the message described by 'nack' having the id 'rawid', into the pending
 * list of the specified consumer. All this in the context of the specified
 * key and group. */
int rioWriteStreamPendingEntry(rio *r, robj *key, const char *groupname, size_t
groupname_len, streamConsumer *consumer, unsigned char *rawid, streamNACK
*nack) {
    /* XCLAIM <key> <group> <consumer> 0 <id> TIME <milliseconds-unix-time>
        RETRYCOUNT <count> JUSTID FORCE. */
    streamID id;
    streamDecodeID(rawid, &id);
    if (rioWriteBulkCount(r, '*', 12) == 0) return 0;
    if (rioWriteBulkString(r, "XCLAIM", 6) == 0) return 0;
    if (rioWriteBulkObject(r, key) == 0) return 0;
    if (rioWriteBulkString(r, groupname, groupname_len) == 0) return 0;
    if (rioWriteBulkString(r, consumer->name, sdslen(consumer->name)) == 0)
return 0;
    if (rioWriteBulkString(r, "0", 1) == 0) return 0;
    if (rioWriteBulkStreamID(r, &id) == 0) return 0;
    if (rioWriteBulkString(r, "TIME", 4) == 0) return 0;
    if (rioWriteBulkLongLong(r, nack->delivery_time) == 0) return 0;
    if (rioWriteBulkString(r, "RETRYCOUNT", 10) == 0) return 0;
    if (rioWriteBulkLongLong(r, nack->delivery_count) == 0) return 0;
    if (rioWriteBulkString(r, "JUSTID", 6) == 0) return 0;
    if (rioWriteBulkString(r, "FORCE", 5) == 0) return 0;
    return 1;
}

/* Helper for rewriteStreamObject(): emit the XGROUP CREATECONSUMER is
 * needed in order to create consumers that do not have any pending entries.
 * All this in the context of the specified key and group. */

```

```

int rioWriteStreamEmptyConsumer(rio *r, robj *key, const char *groupname,
size_t groupname_len, streamConsumer *consumer) {
    /* XGROUP CREATECONSUMER <key> <group> <consumer> */
    if (rioWriteBulkCount(r, '*',5) == 0) return 0;
    if (rioWriteBulkString(r,"XGROUP",6) == 0) return 0;
    if (rioWriteBulkString(r,"CREATECONSUMER",14) == 0) return 0;
    if (rioWriteBulkObject(r,key) == 0) return 0;
    if (rioWriteBulkString(r,groupname,groupname_len) == 0) return 0;
    if (rioWriteBulkString(r,consumer->name,sdslen(consumer->name)) == 0)
return 0;
    return 1;
}

/* Emit the commands needed to rebuild a stream object.
 * The function returns 0 on error, 1 on success. */
int rewriteStreamObject(rio *r, robj *key, robj *o) {
    stream *s = o->ptr;
    streamIterator si;
    streamIteratorStart(&si,s,NULL,NULL,0);
    streamID id;
    int64_t numfields;

    if (s->length) {
        /* Reconstruct the stream data using XADD commands. */
        while(streamIteratorGetID(&si,&id,&numfields)) {
            /* Emit a two elements array for each item. The first is
             * the ID, the second is an array of field-value pairs. */

            /* Emit the XADD <key> <id> ...fields... command. */
            if (!rioWriteBulkCount(r, '*',3+numfields*2) ||
                !rioWriteBulkString(r,"XADD",4) ||
                !rioWriteBulkObject(r,key) ||
                !rioWriteBulkStreamID(r,&id))
            {
                streamIteratorStop(&si);
                return 0;
            }
            while(numfields--) {
                unsigned char *field, *value;
                int64_t field_len, value_len;

                streamIteratorGetField(&si,&field,&value,&field_len,&value_len);
                if (!rioWriteBulkString(r,(char*)field,field_len) ||
                    !rioWriteBulkString(r,(char*)value,value_len))
                {
                    streamIteratorStop(&si);
                    return 0;
                }
            }
        }
    }
    } else {

```

```

/* Use the XADD MAXLEN 0 trick to generate an empty stream if
 * the key we are serializing is an empty string, which is possible
 * for the Stream type. */
id.ms = 0; id.seq = 1;
if (!rioWriteBulkCount(r, '*', 7) ||
    !rioWriteBulkString(r, "XADD", 4) ||
    !rioWriteBulkObject(r, key) ||
    !rioWriteBulkString(r, "MAXLEN", 6) ||
    !rioWriteBulkString(r, "0", 1) ||
    !rioWriteBulkStreamID(r, &id) ||
    !rioWriteBulkString(r, "x", 1) ||
    !rioWriteBulkString(r, "y", 1))
{
    streamIteratorStop(&si);
    return 0;
}

/* Append XSETID after XADD, make sure lastid is correct,
 * in case of XDEL lastid. */
if (!rioWriteBulkCount(r, '*', 7) ||
    !rioWriteBulkString(r, "XSETID", 6) ||
    !rioWriteBulkObject(r, key) ||
    !rioWriteBulkStreamID(r, &s->last_id) ||
    !rioWriteBulkString(r, "ENTRIESADDED", 12) ||
    !rioWriteBulkLongLong(r, s->entries_added) ||
    !rioWriteBulkString(r, "MAXDELETEDID", 12) ||
    !rioWriteBulkStreamID(r, &s->max_deleted_entry_id))
{
    streamIteratorStop(&si);
    return 0;
}

/* Create all the stream consumer groups. */
if (s->cgroups) {
    raxIterator ri;
    raxStart(&ri, s->cgroups);
    raxSeek(&ri, "^", NULL, 0);
    while(raxNext(&ri)) {
        streamCG *group = ri.data;
        /* Emit the XGROUP CREATE in order to create the group. */
        if (!rioWriteBulkCount(r, '*', 7) ||
            !rioWriteBulkString(r, "XGROUP", 6) ||
            !rioWriteBulkString(r, "CREATE", 6) ||
            !rioWriteBulkObject(r, key) ||
            !rioWriteBulkString(r, (char*)ri.key, ri.key_len) ||
            !rioWriteBulkStreamID(r, &group->last_id) ||
            !rioWriteBulkString(r, "ENTRIESREAD", 11) ||
            !rioWriteBulkLongLong(r, group->entries_read))
        {

```

```

        raxStop(&ri);
        streamIteratorStop(&si);
        return 0;
    }

    /* Generate XCLAIMs for each consumer that happens to
     * have pending entries. Empty consumers would be generated with
     * XGROUP CREATECONSUMER. */
    raxIterator ri_cons;
    raxStart(&ri_cons, group->consumers);
    raxSeek(&ri_cons, "^", NULL, 0);
    while(raxNext(&ri_cons)) {
        streamConsumer *consumer = ri_cons.data;
        /* If there are no pending entries, just emit XGROUP
CREATECONSUMER */
        if (raxSize(consumer->pel) == 0) {
            if (rioWriteStreamEmptyConsumer(r, key, (char*)ri.key,
                                           ri.key_len, consumer) == 0)
            {
                raxStop(&ri_cons);
                raxStop(&ri);
                streamIteratorStop(&si);
                return 0;
            }
            continue;
        }
        /* For the current consumer, iterate all the PEL entries
         * to emit the XCLAIM protocol. */
        raxIterator ri_pel;
        raxStart(&ri_pel, consumer->pel);
        raxSeek(&ri_pel, "^", NULL, 0);
        while(raxNext(&ri_pel)) {
            streamNACK *nack = ri_pel.data;
            if (rioWriteStreamPendingEntry(r, key, (char*)ri.key,
                                           ri.key_len, consumer,
                                           ri_pel.key, nack) == 0)
            {
                raxStop(&ri_pel);
                raxStop(&ri_cons);
                raxStop(&ri);
                streamIteratorStop(&si);
                return 0;
            }
        }
        raxStop(&ri_pel);
    }
    raxStop(&ri_cons);
}
raxStop(&ri);
}

```

```

    streamIteratorStop(&si);
    return 1;
}

/* Call the module type callback in order to rewrite a data type
 * that is exported by a module and is not handled by Redis itself.
 * The function returns 0 on error, 1 on success. */
int rewriteModuleObject(rio *r, robj *key, robj *o, int dbid) {
    RedisModuleIO io;
    moduleValue *mv = o->ptr;
    moduleType *mt = mv->type;
    moduleInitIOContext(io,mt,r,key,dbid);
    mt->aof_rewrite(&io,key,mv->value);
    if (io.ctx) {
        moduleFreeContext(io.ctx);
        zfree(io.ctx);
    }
    return io.error ? 0 : 1;
}

static int rewriteFunctions(rio *aof) {
    dict *functions = functionsLibGet();
    dictIterator *iter = dictGetIterator(functions);
    dictEntry *entry = NULL;
    while ((entry = dictNext(iter))) {
        functionLibInfo *li = dictGetVal(entry);
        if (rioWrite(aof, "*3\r\n", 4) == 0) goto werr;
        char function_load[] = "$8\r\nFUNCTION\r\n$4\r\nLOAD\r\n";
        if (rioWrite(aof, function_load, sizeof(function_load) - 1) == 0) goto
werr;
        if (rioWriteBulkString(aof, li->code, sdslen(li->code)) == 0) goto
werr;
    }
    dictReleaseIterator(iter);
    return 1;

werr:
    dictReleaseIterator(iter);
    return 0;
}

int rewriteAppendOnlyFileRio(rio *aof) {
    dictIterator *di = NULL;
    dictEntry *de;
    int j;
    long key_count = 0;
    long long updated_time = 0;

    /* Record timestamp at the beginning of rewriting AOF. */
    if (server.aof_timestamp_enabled) {
        sds ts = genAofTimestampAnnotationIfNeeded(1);

```

```

        if (rioWrite(aof,ts,sdslen(ts)) == 0) { sdsfree(ts); goto werr; }
        sdsfree(ts);
    }

    if (rewriteFunctions(aof) == 0) goto werr;

    for (j = 0; j < server.dbnum; j++) {
        char selectcmd[] = "*2\r\n$6\r\nSELECT\r\n";
        redisDb *db = server.db+j;
        dict *d = db->dict;
        if (dictSize(d) == 0) continue;
        di = dictGetSafeIterator(d);

        /* SELECT the new DB */
        if (rioWrite(aof,selectcmd,sizeof(selectcmd)-1) == 0) goto werr;
        if (rioWriteBulkLongLong(aof,j) == 0) goto werr;

        /* Iterate this DB writing every entry */
        while((de = dictNext(di)) != NULL) {
            sds keystr;
            robj key, *o;
            long long expiretime;
            size_t aof_bytes_before_key = aof->processed_bytes;

            keystr = dictGetKey(de);
            o = dictGetVal(de);
            initStaticStringObject(key,keystr);

            expiretime = getExpire(db,&key);

            /* Save the key and associated value */
            if (o->type == OBJ_STRING) {
                /* Emit a SET command */
                char cmd[] = "*3\r\n$3\r\nSET\r\n";
                if (rioWrite(aof,cmd,sizeof(cmd)-1) == 0) goto werr;
                /* Key and value */
                if (rioWriteBulkObject(aof,&key) == 0) goto werr;
                if (rioWriteBulkObject(aof,o) == 0) goto werr;
            } else if (o->type == OBJ_LIST) {
                if (rewriteListObject(aof,&key,o) == 0) goto werr;
            } else if (o->type == OBJ_SET) {
                if (rewriteSetObject(aof,&key,o) == 0) goto werr;
            } else if (o->type == OBJ_ZSET) {
                if (rewriteSortedSetObject(aof,&key,o) == 0) goto werr;
            } else if (o->type == OBJ_HASH) {
                if (rewriteHashObject(aof,&key,o) == 0) goto werr;
            } else if (o->type == OBJ_STREAM) {
                if (rewriteStreamObject(aof,&key,o) == 0) goto werr;
            } else if (o->type == OBJ_MODULE) {
                if (rewriteModuleObject(aof,&key,o,j) == 0) goto werr;
            } else {

```

```

        serverPanic("Unknown object type");
    }

    /* In fork child process, we can try to release memory back to the
     * OS and possibly avoid or decrease COW. We guve the dismiss
     * mechanism a hint about an estimated size of the object we
stored. */
    size_t dump_size = aof->processed_bytes - aof_bytes_before_key;
    if (server.in_fork_child) dismissObject(o, dump_size);

    /* Save the expire time */
    if (expiretime != -1) {
        char cmd[] = "*3\r\n$9\r\nPEXPIREAT\r\n";
        if (rioWrite(aof, cmd, sizeof(cmd)-1) == 0) goto werr;
        if (rioWriteBulkObject(aof, &key) == 0) goto werr;
        if (rioWriteBulkLongLong(aof, expiretime) == 0) goto werr;
    }

    /* Update info every 1 second (approximately).
     * in order to avoid calling mstime() on each iteration, we will
     * check the diff every 1024 keys */
    if ((key_count++ & 1023) == 0) {
        long long now = mstime();
        if (now - updated_time >= 1000) {
            sendChildInfo(CHILD_INFO_TYPE_CURRENT_INFO, key_count, "AOF
rewrite");
            updated_time = now;
        }
    }

    /* Delay before next key if required (for testing) */
    if (server.rdb_key_save_delay)
        debugDelay(server.rdb_key_save_delay);
}
dictReleaseIterator(di);
di = NULL;
}
return C_OK;

werr:
    if (di) dictReleaseIterator(di);
    return C_ERR;
}

/* Write a sequence of commands able to fully rebuild the dataset into
 * "filename". Used both by REWRITEAOF and BGREWRITEAOF.
 *
 * In order to minimize the number of commands needed in the rewritten
 * log Redis uses variadic commands when possible, such as Rpush, SADD
 * and ZADD. However at max AOF_REWRITE_ITEMS_PER_CMD items per time
 * are inserted using a single command. */

```



```

int rewriteAppendOnlyFile(char *filename) {
    rio aof;
    FILE *fp = NULL;
    char tmpfile[256];

    /* Note that we have to use a different temp name here compared to the
     * one used by rewriteAppendOnlyFileBackground() function. */
    snprintf(tmpfile,256,"temp-rewriteaof-%d.aof", (int) getpid());
    fp = fopen(tmpfile,"w");
    if (!fp) {
        serverLog(LL_WARNING, "Opening the temp file for AOF rewrite in
rewriteAppendOnlyFile(): %s", strerror(errno));
        return C_ERR;
    }

    rioInitWithFile(&aof,fp);

    if (server.aof_rewrite_incremental_fsync)
        rioSetAutoSync(&aof,REDIS_AUTOSYNC_BYTES);

    startSaving(RDBFLAGS_AOF_PREAMBLE);

    if (server.aof_use_rdb_preamble) {
        int error;
        if (rdbSaveRio(SLAVE_REQ_NONE,&aof,&error,RDBFLAGS_AOF_PREAMBLE,NULL)
== C_ERR) {
            errno = error;
            goto werr;
        }
    } else {
        if (rewriteAppendOnlyFileRio(&aof) == C_ERR) goto werr;
    }

    /* Make sure data will not remain on the OS's output buffers */
    if (fflush(fp)) goto werr;
    if (fsync(fileno(fp))) goto werr;
    if (fclose(fp)) { fp = NULL; goto werr; }
    fp = NULL;

    /* Use RENAME to make sure the DB file is changed atomically only
     * if the generate DB file is ok. */
    if (rename(tmpfile,filename) == -1) {
        serverLog(LL_WARNING,"Error moving temp append only file on the final
destination: %s", strerror(errno));
        unlink(tmpfile);
        stopSaving(0);
        return C_ERR;
    }
    serverLog(LL_NOTICE,"SYNC append only file rewrite performed");
    stopSaving(1);
}

```

```

    return C_OK;

werr:
    serverLog(LL_WARNING,"Write error writing append only file on disk: %s",
    strerror(errno));
    if (fp) fclose(fp);
    unlink(tmpfile);
    stopSaving(0);
    return C_ERR;
}

/* -----
 * AOF background rewrite
 * ----- */

/* This is how rewriting of the append only file in background works:
 *
 * 1) The user calls BGREWRITEAOF
 * 2) Redis calls this function, that forks():
 *    2a) the child rewrite the append only file in a temp file.
 *    2b) the parent open a new INCR AOF file to continue writing.
 * 3) When the child finished '2a' exists.
 * 4) The parent will trap the exit code, if it's OK, it will:
 *    4a) get a new BASE file name and mark the previous (if we have) as the
HISTORY type
 *    4b) rename(2) the temp file in new BASE file name
 *    4c) mark the rewritten INCR AOFs as history type
 *    4d) persist AOF manifest file
 *    4e) Delete the history files use bio
 */
int rewriteAppendOnlyFileBackground(void) {
    pid_t childpid;

    if (hasActiveChildProcess()) return C_ERR;

    if (dirCreateIfMissing(server.aof_dirname) == -1) {
        serverLog(LL_WARNING, "Can't open or create append-only dir %s: %s",
            server.aof_dirname, strerror(errno));
        return C_ERR;
    }

    /* We set aof_selected_db to -1 in order to force the next call to the
     * feedAppendOnlyFile() to issue a SELECT command. */
    server.aof_selected_db = -1;
    flushAppendOnlyFile(1);
    if (openNewIncrAofForAppend() != C_OK) return C_ERR;
    server.stat_aof_rewrites++;
    if ((childpid = redisFork(CHILD_TYPE_AOF)) == 0) {
        char tmpfile[256];

        /* Child */
        redisSetProcTitle("redis-aof-rewrite");

```

```

        redisSetCpuAffinity(server.aof_rewrite_cpulist);
        snprintf(tmpfile,256,"temp-rewriteaof-bg-%d.aof", (int) getpid());
        if (rewriteAppendOnlyFile(tmpfile) == C_OK) {
            sendChildCowInfo(CHILD_INFO_TYPE_AOF_COW_SIZE, "AOF rewrite");
            exitFromChild(0);
        } else {
            exitFromChild(1);
        }
    } else {
        /* Parent */
        if (childpid == -1) {
            server.aof_lastbgrewrite_status = C_ERR;
            serverLog(LL_WARNING,
                "Can't rewrite append only file in background: fork: %s",
                strerror(errno));
            return C_ERR;
        }
        serverLog(LL_NOTICE,
            "Background append only file rewriting started by pid %ld", (long)
childpid);
        server.aof_rewrite_scheduled = 0;
        server.aof_rewrite_time_start = time(NULL);
        return C_OK;
    }
    return C_OK; /* unreachable */
}

void bgrewriteaofCommand(client *c) {
    if (server.child_type == CHILD_TYPE_AOF) {
        addReplyError(c,"Background append only file rewriting already in
progress");
    } else if (hasActiveChildProcess() || server.in_exec) {
        server.aof_rewrite_scheduled = 1;
        /* When manually triggering AOFRW we reset the count
        * so that it can be executed immediately. */
        server.stat_aofrw_consecutive_failures = 0;
        addReplyStatus(c,"Background append only file rewriting scheduled");
    } else if (rewriteAppendOnlyFileBackground() == C_OK) {
        addReplyStatus(c,"Background append only file rewriting started");
    } else {
        addReplyError(c,"Can't execute an AOF background rewriting. "
            "Please check the server logs for more information.");
    }
}

void aofRemoveTempFile(pid_t childpid) {
    char tmpfile[256];

    snprintf(tmpfile,256,"temp-rewriteaof-bg-%d.aof", (int) childpid);
    bg_unlink(tmpfile);
}

```

```

    snprintf(tmpfile,256,"temp-rewriteaof-%d.aof", (int) childpid);
    bg_unlink(tmpfile);
}

/* Get size of an AOF file.
 * The status argument is an optional output argument to be filled with
 * one of the AOF_ status values. */
off_t getAppendOnlyFileSize(sds filename, int *status) {
    struct redis_stat sb;
    off_t size;
    mstime_t latency;

    sds aof_filepath = makePath(server.aof_dirname, filename);
    latencyStartMonitor(latency);
    if (redis_stat(aof_filepath, &sb) == -1) {
        if (status) *status = errno == ENOENT ? AOF_NOT_EXIST : AOF_OPEN_ERR;
        serverLog(LL_WARNING, "Unable to obtain the AOF file %s length. stat:
%s",
                filename, strerror(errno));
        size = 0;
    } else {
        if (status) *status = AOF_OK;
        size = sb.st_size;
    }
    latencyEndMonitor(latency);
    latencyAddSampleIfNeeded("aof-fstat", latency);
    sdsfree(aof_filepath);
    return size;
}

/* Get size of all AOF files referred by the manifest (excluding history).
 * The status argument is an output argument to be filled with
 * one of the AOF_ status values. */
off_t getBaseAndIncrAppendOnlyFilesSize(aofManifest *am, int *status) {
    off_t size = 0;
    listNode *ln;
    listIter li;

    if (am->base_aof_info) {
        serverAssert(am->base_aof_info->file_type == AOF_FILE_TYPE_BASE);

        size += getAppendOnlyFileSize(am->base_aof_info->file_name, status);
        if (*status != AOF_OK) return 0;
    }

    listRewind(am->incr_aof_list, &li);
    while ((ln = listNext(&li)) != NULL) {
        aofInfo *ai = (aofInfo*)ln->value;
        serverAssert(ai->file_type == AOF_FILE_TYPE_INCR);
        size += getAppendOnlyFileSize(ai->file_name, status);
        if (*status != AOF_OK) return 0;
    }
}

```

```

    }

    return size;
}

int getBaseAndIncrAppendOnlyFilesNum(aofManifest *am) {
    int num = 0;
    if (am->base_aof_info) num++;
    if (am->incr_aof_list) num += listLength(am->incr_aof_list);
    return num;
}

/* A background append only file rewriting (BGREWRITEAOF) terminated its work.
 * Handle this. */
void backgroundRewriteDoneHandler(int exitcode, int bysignal) {
    if (!bysignal && exitcode == 0) {
        char tmpfile[256];
        long long now = ustime();
        sds new_base_filepath = NULL;
        sds new_incr_filepath = NULL;
        aofManifest *temp_am;
        mstime_t latency;

        serverLog(LL_NOTICE,
            "Background AOF rewrite terminated with success");

        snprintf(tmpfile, 256, "temp-rewriteaof-bg-%d.aof",
            (int)server.child_pid);

        serverAssert(server.aof_manifest != NULL);

        /* Dup a temporary aof_manifest for subsequent modifications. */
        temp_am = aofManifestDup(server.aof_manifest);

        /* Get a new BASE file name and mark the previous (if we have)
         * as the HISTORY type. */
        sds new_base_filename = getNewBaseFileNameAndMarkPreAsHistory(temp_am);
        serverAssert(new_base_filename != NULL);
        new_base_filepath = makePath(server.aof_dirname, new_base_filename);

        /* Rename the temporary aof file to 'new_base_filename'. */
        latencyStartMonitor(latency);
        if (rename(tmpfile, new_base_filepath) == -1) {
            serverLog(LL_WARNING,
                "Error trying to rename the temporary AOF file %s into %s: %s",
                tmpfile,
                new_base_filepath,
                strerror(errno));
            aofManifestFree(temp_am);
            sdsfree(new_base_filepath);
            goto cleanup;
        }
    }
}

```

```

    }
    latencyEndMonitor(latency);
    latencyAddSampleIfNeeded("aof-rename", latency);

    /* Rename the temporary incr aof file to 'new_incr_filename'. */
    if (server.aof_state == AOF_WAIT_REWRITE) {
        /* Get temporary incr aof name. */
        sds temp_incr_aof_name = getTempIncrAofName();
        sds temp_incr_filepath = makePath(server.aof_dirname,
temp_incr_aof_name);
        sdsfree(temp_incr_aof_name);
        /* Get next new incr aof name. */
        sds new_incr_filename = getNewIncrAofName(temp_am);
        new_incr_filepath = makePath(server.aof_dirname,
new_incr_filename);
        latencyStartMonitor(latency);
        if (rename(temp_incr_filepath, new_incr_filepath) == -1) {
            serverLog(LL_WARNING,
                "Error trying to rename the temporary incr AOF file %s into
%s: %s",
                temp_incr_filepath,
                new_incr_filepath,
                strerror(errno));
            bg_unlink(new_base_filepath);
            sdsfree(new_base_filepath);
            aofManifestFree(temp_am);
            sdsfree(temp_incr_filepath);
            sdsfree(new_incr_filepath);
            goto cleanup;
        }
        latencyEndMonitor(latency);
        latencyAddSampleIfNeeded("aof-rename", latency);
        sdsfree(temp_incr_filepath);
    }

    /* Change the AOF file type in 'incr_aof_list' from AOF_FILE_TYPE_INCR
    * to AOF_FILE_TYPE_HIST, and move them to the 'history_aof_list'. */
    markRewrittenIncrAofAsHistory(temp_am);

    /* Persist our modifications. */
    if (persistAofManifest(temp_am) == C_ERR) {
        bg_unlink(new_base_filepath);
        aofManifestFree(temp_am);
        sdsfree(new_base_filepath);
        if (new_incr_filepath) {
            bg_unlink(new_incr_filepath);
            sdsfree(new_incr_filepath);
        }
        goto cleanup;
    }
    sdsfree(new_base_filepath);

```

```

        if (new_incr_filepath) sdsfree(new_incr_filepath);

        /* We can safely let `server.aof_manifest` point to 'temp_am' and free
the previous one. */
        aofManifestFreeAndUpdate(temp_am);

        if (server.aof_fd != -1) {
            /* AOF enabled. */
            server.aof_selected_db = -1; /* Make sure SELECT is re-issued */
            server.aof_current_size = getAppendOnlyFileSize(new_base_filename,
NULL) + server.aof_last_incr_size;
            server.aof_rewrite_base_size = server.aof_current_size;
            server.aof_fsync_offset = server.aof_current_size;
            server.aof_last_fsync = server.unixtime;
        }

        /* We don't care about the return value of `aofDelHistoryFiles`,
because the history
        * deletion failure will not cause any problems. */
        aofDelHistoryFiles();

        server.aof_lastbgrewrite_status = C_OK;
        server.stat_aofrw_consecutive_failures = 0;

        serverLog(LL_NOTICE, "Background AOF rewrite finished successfully");
        /* Change state from WAIT_REWRITE to ON if needed */
        if (server.aof_state == AOF_WAIT_REWRITE)
            server.aof_state = AOF_ON;

        serverLog(LL_VERBOSE,
            "Background AOF rewrite signal handler took %lldus", ustime()-now);
    } else if (!bysignal && exitcode != 0) {
        server.aof_lastbgrewrite_status = C_ERR;
        server.stat_aofrw_consecutive_failures++;

        serverLog(LL_WARNING,
            "Background AOF rewrite terminated with error");
    } else {
        /* SIGUSR1 is whitelisted, so we have a way to kill a child without
        * triggering an error condition. */
        if (bysignal != SIGUSR1) {
            server.aof_lastbgrewrite_status = C_ERR;
            server.stat_aofrw_consecutive_failures++;
        }

        serverLog(LL_WARNING,
            "Background AOF rewrite terminated by signal %d", bysignal);
    }
}

cleanup:
    aofRemoveTempFile(server.child_pid);

```

```
/* Clear AOF buffer and delete temp incr aof for next rewrite. */
if (server.aof_state == AOF_WAIT_REWRITE) {
    sdsfree(server.aof_buf);
    server.aof_buf = sdsempty();
    aofDelTempIncrAofFile();
}
server.aof_rewrite_time_last = time(NULL)-server.aof_rewrite_time_start;
server.aof_rewrite_time_start = -1;
/* Schedule a new rewrite if we are waiting for it to switch the AOF ON. */
if (server.aof_state == AOF_WAIT_REWRITE)
    server.aof_rewrite_scheduled = 1;
}
```

/asciilogo.h

[to top](#)


```

/* This file implements atomic counters using c11 _Atomic, __atomic or __sync
 * macros if available, otherwise we will throw an error when compile.
 *
 * The exported interface is composed of three macros:
 *
 * atomicIncr(var,count) -- Increment the atomic counter
 * atomicGetIncr(var,oldvalue_var,count) -- Get and increment the atomic
counter
 * atomicDecr(var,count) -- Decrement the atomic counter
 * atomicGet(var,dstvar) -- Fetch the atomic counter value
 * atomicSet(var,value) -- Set the atomic counter value
 * atomicGetWithSync(var,value) -- 'atomicGet' with inter-thread
synchronization
 * atomicSetWithSync(var,value) -- 'atomicSet' with inter-thread
synchronization
 *
 * Never use return value from the macros, instead use the AtomicGetIncr()
 * if you need to get the current value and increment it atomically, like
 * in the following example:
 *
 * long oldvalue;
 * atomicGetIncr(myvar,oldvalue,1);
 * doSomethingWith(oldvalue);
 *
 * -----
 *
 * Copyright (c) 2015, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)

```

```

* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/

#include <pthread.h>
#include "config.h"

#ifndef __ATOMIC_VAR_H
#define __ATOMIC_VAR_H

/* Define redisAtomic for atomic variable. */
#define redisAtomic

/* To test Redis with Helgrind (a Valgrind tool) it is useful to define
 * the following macro, so that __sync macros are used: those can be detected
 * by Helgrind (even if they are less efficient) so that no false positive
 * is reported. */
// #define __ATOMIC_VAR_FORCE_SYNC_MACROS

/* There will be many false positives if we test Redis with Helgrind, since
 * Helgrind can't understand we have imposed ordering on the program, so
 * we use macros in helgrind.h to tell Helgrind inter-thread happens-before
 * relationship explicitly for avoiding false positives.
 *
 * For more details, please see: valgrind/helgrind.h and
 * https://www.valgrind.org/docs/manual/hg-manual.html#hg-manual.effective-use
 *
 * These macros take effect only when 'make helgrind', and you must first
 * install Valgrind in the default path configuration. */
#ifdef __ATOMIC_VAR_FORCE_SYNC_MACROS
#include <valgrind/helgrind.h>
#else
#define ANNOTATE_HAPPENS_BEFORE(v) ((void) v)
#define ANNOTATE_HAPPENS_AFTER(v) ((void) v)
#endif

#if !defined(__ATOMIC_VAR_FORCE_SYNC_MACROS) && defined(__STDC_VERSION__) && \
    (__STDC_VERSION__ >= 201112L) && !defined(__STDC_NO_ATOMICS__)
/* Use '_Atomic' keyword if the compiler supports. */
#undef redisAtomic
#define redisAtomic _Atomic
/* Implementation using _Atomic in C11. */

#include <stdatomic.h>
#define atomicIncr(var,count) atomic_fetch_add_explicit(&var,
    (count),memory_order_relaxed)
#define atomicGetIncr(var,oldvalue_var,count) do { \
    oldvalue_var = atomic_fetch_add_explicit(&var,
    (count),memory_order_relaxed); \
} while(0)
#define atomicDecr(var,count) atomic_fetch_sub_explicit(&var,

```

```

(count),memory_order_relaxed)
#define atomicGet(var,dstvar) do { \
    dstvar = atomic_load_explicit(&var,memory_order_relaxed); \
} while(0)
#define atomicSet(var,value)
atomic_store_explicit(&var,value,memory_order_relaxed)
#define atomicGetWithSync(var,dstvar) do { \
    dstvar = atomic_load_explicit(&var,memory_order_seq_cst); \
} while(0)
#define atomicSetWithSync(var,value) \
    atomic_store_explicit(&var,value,memory_order_seq_cst)
#define REDIS_ATOMIC_API "c11-builtin"

#elif !defined(__ATOMIC_VAR_FORCE_SYNC_MACROS) && \
    (!defined(__clang__) || !defined(__APPLE__) || __apple_build_version__ >
4210057) && \
    defined(__ATOMIC_RELAXED) && defined(__ATOMIC_SEQ_CST)
/* Implementation using __atomic macros. */

#define atomicIncr(var,count) __atomic_add_fetch(&var,(count),__ATOMIC_RELAXED)
#define atomicGetIncr(var,oldvalue_var,count) do { \
    oldvalue_var = __atomic_fetch_add(&var,(count),__ATOMIC_RELAXED); \
} while(0)
#define atomicDecr(var,count) __atomic_sub_fetch(&var,(count),__ATOMIC_RELAXED)
#define atomicGet(var,dstvar) do { \
    dstvar = __atomic_load_n(&var,__ATOMIC_RELAXED); \
} while(0)
#define atomicSet(var,value) __atomic_store_n(&var,value,__ATOMIC_RELAXED)
#define atomicGetWithSync(var,dstvar) do { \
    dstvar = __atomic_load_n(&var,__ATOMIC_SEQ_CST); \
} while(0)
#define atomicSetWithSync(var,value) \
    __atomic_store_n(&var,value,__ATOMIC_SEQ_CST)
#define REDIS_ATOMIC_API "atomic-builtin"

#elif defined(HAVE_ATOMIC)
/* Implementation using __sync macros. */

#define atomicIncr(var,count) __sync_add_and_fetch(&var,(count))
#define atomicGetIncr(var,oldvalue_var,count) do { \
    oldvalue_var = __sync_fetch_and_add(&var,(count)); \
} while(0)
#define atomicDecr(var,count) __sync_sub_and_fetch(&var,(count))
#define atomicGet(var,dstvar) do { \
    dstvar = __sync_sub_and_fetch(&var,0); \
} while(0)
#define atomicSet(var,value) do { \
    while(!__sync_bool_compare_and_swap(&var,var,value)); \
} while(0)
/* Actually the builtin issues a full memory barrier by default. */
#define atomicGetWithSync(var,dstvar) { \

```

```
        dstvar = __sync_sub_and_fetch(&var,0,__sync_synchronize); \
        ANNOTATE_HAPPENS_AFTER(&var); \
    } while(0)
#define atomicSetWithSync(var,value) do { \
    ANNOTATE_HAPPENS_BEFORE(&var); \
    while(!__sync_bool_compare_and_swap(&var,var,value,__sync_synchronize)); \
} while(0)
#define REDIS_ATOMIC_API "sync-builtin"

#else
#error "Unable to determine atomic operations for your platform"

#endif
#endif /* __ATOMIC_VAR_H */
```

/bio.c

[to top](#)

```

/* Background I/O service for Redis.
 *
 * This file implements operations that we need to perform in the background.
 * Currently there is only a single operation, that is a background close(2)
 * system call. This is needed as when the process is the last owner of a
 * reference to a file closing it means unlinking it, and the deletion of the
 * file is slow, blocking the server.
 *
 * In the future we'll either continue implementing new things we need or
 * we'll switch to libeio. However there are probably long term uses for this
 * file as we may want to put here Redis specific background tasks (for
instance
 * it is not impossible that we'll need a non blocking FLUSHDB/FLUSHALL
 * implementation).
 *
 * DESIGN
 * -----
 *
 * The design is trivial, we have a structure representing a job to perform
 * and a different thread and job queue for every job type.
 * Every thread waits for new jobs in its queue, and process every job
 * sequentially.
 *
 * Jobs of the same type are guaranteed to be processed from the least
 * recently inserted to the most recently inserted (older jobs processed
 * first).
 *
 * Currently there is no way for the creator of the job to be notified about
 * the completion of the operation, this will only be added when/if needed.
 *
 * -----
 *
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

```

```
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/
```

```
#include "server.h"
#include "bio.h"
```

```
static pthread_t bio_threads[BIO_NUM_OPS];
static pthread_mutex_t bio_mutex[BIO_NUM_OPS];
static pthread_cond_t bio_newjob_cond[BIO_NUM_OPS];
static pthread_cond_t bio_step_cond[BIO_NUM_OPS];
static list *bio_jobs[BIO_NUM_OPS];
/* The following array is used to hold the number of pending jobs for every
 * OP type. This allows us to export the bioPendingJobsOfType() API that is
 * useful when the main thread wants to perform some operation that may involve
 * objects shared with the background thread. The main thread will just wait
 * that there are no longer jobs of this type to be executed before performing
 * the sensible operation. This data is also useful for reporting. */
static unsigned long long bio_pending[BIO_NUM_OPS];

/* This structure represents a background Job. It is only used locally to this
 * file as the API does not expose the internals at all. */
struct bio_job {
    /* Job specific arguments.*/
    int fd; /* Fd for file based background jobs */
    lazy_free_fn *free_fn; /* Function that will free the provided arguments */
    void *free_args[]; /* List of arguments to be passed to the free function
*/
};

void *bioProcessBackgroundJobs(void *arg);

/* Make sure we have enough stack to perform all the things we do in the
 * main thread. */
#define REDIS_THREAD_STACK_SIZE (1024*1024*4)

/* Initialize the background system, spawning the thread. */
void bioInit(void) {
    pthread_attr_t attr;
    pthread_t thread;
    size_t stacksize;
    int j;

    /* Initialization of state vars and objects */
```



```

for (j = 0; j < BIO_NUM_OPS; j++) {
    pthread_mutex_init(&bio_mutex[j],NULL);
    pthread_cond_init(&bio_newjob_cond[j],NULL);
    pthread_cond_init(&bio_step_cond[j],NULL);
    bio_jobs[j] = listCreate();
    bio_pending[j] = 0;
}

/* Set the stack size as by default it may be small in some system */
pthread_attr_init(&attr);
pthread_attr_getstacksize(&attr,&stacksize);
if (!stacksize) stacksize = 1; /* The world is full of Solaris Fixes */
while (stacksize < REDIS_THREAD_STACK_SIZE) stacksize *= 2;
pthread_attr_setstacksize(&attr, stacksize);

/* Ready to spawn our threads. We use the single argument the thread
 * function accepts in order to pass the job ID the thread is
 * responsible of. */
for (j = 0; j < BIO_NUM_OPS; j++) {
    void *arg = (void*)(unsigned long) j;
    if (pthread_create(&thread,&attr,bioProcessBackgroundJobs,arg) != 0) {
        serverLog(LL_WARNING,"Fatal: Can't initialize Background Jobs.");
        exit(1);
    }
    bio_threads[j] = thread;
}
}

void bioSubmitJob(int type, struct bio_job *job) {
    pthread_mutex_lock(&bio_mutex[type]);
    listAddNodeTail(bio_jobs[type],job);
    bio_pending[type]++;
    pthread_cond_signal(&bio_newjob_cond[type]);
    pthread_mutex_unlock(&bio_mutex[type]);
}

void bioCreateLazyFreeJob(lazy_free_fn free_fn, int arg_count, ...) {
    va_list valist;
    /* Allocate memory for the job structure and all required
     * arguments */
    struct bio_job *job = zmalloc(sizeof(*job) + sizeof(void *) * (arg_count));
    job->free_fn = free_fn;

    va_start(valist, arg_count);
    for (int i = 0; i < arg_count; i++) {
        job->free_args[i] = va_arg(valist, void *);
    }
    va_end(valist);
    bioSubmitJob(BIO_LAZY_FREE, job);
}

```

```

void bioCreateCloseJob(int fd) {
    struct bio_job *job = zmalloc(sizeof(*job));
    job->fd = fd;

    bioSubmitJob(BIO_CLOSE_FILE, job);
}

void bioCreateFsyncJob(int fd) {
    struct bio_job *job = zmalloc(sizeof(*job));
    job->fd = fd;

    bioSubmitJob(BIO_AOF_FSYNC, job);
}

void *bioProcessBackgroundJobs(void *arg) {
    struct bio_job *job;
    unsigned long type = (unsigned long) arg;
    sigset_t sigset;

    /* Check that the type is within the right interval. */
    if (type >= BIO_NUM_OPS) {
        serverLog(LL_WARNING,
            "Warning: bio thread started with wrong type %lu", type);
        return NULL;
    }

    switch (type) {
    case BIO_CLOSE_FILE:
        redis_set_thread_title("bio_close_file");
        break;
    case BIO_AOF_FSYNC:
        redis_set_thread_title("bio_aof_fsync");
        break;
    case BIO_LAZY_FREE:
        redis_set_thread_title("bio_lazy_free");
        break;
    }

    redisSetCpuAffinity(server.bio_cpulist);

    makeThreadKillable();

    pthread_mutex_lock(&bio_mutex[type]);
    /* Block SIGALRM so we are sure that only the main thread will
     * receive the watchdog signal. */
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGALRM);
    if (pthread_sigmask(SIG_BLOCK, &sigset, NULL))
        serverLog(LL_WARNING,
            "Warning: can't mask SIGALRM in bio.c thread: %s",
            strerror(errno));
}

```

```

while(1) {
    listNode *ln;

    /* The loop always starts with the lock hold. */
    if (listLength(bio_jobs[type]) == 0) {
        pthread_cond_wait(&bio_newjob_cond[type], &bio_mutex[type]);
        continue;
    }
    /* Pop the job from the queue. */
    ln = listFirst(bio_jobs[type]);
    job = ln->value;
    /* It is now possible to unlock the background system as we know have
     * a stand alone job structure to process.*/
    pthread_mutex_unlock(&bio_mutex[type]);

    /* Process the job accordingly to its type. */
    if (type == BIO_CLOSE_FILE) {
        close(job->fd);
    } else if (type == BIO_AOF_FSYNC) {
        /* The fd may be closed by main thread and reused for another
         * socket, pipe, or file. We just ignore these errno because
         * aof fsync did not really fail. */
        if (redis_fsync(job->fd) == -1 &&
            errno != EBADF && errno != EINVAL)
        {
            int last_status;
            atomicGet(server.aof_bio_fsync_status, last_status);
            atomicSet(server.aof_bio_fsync_status, C_ERR);
            atomicSet(server.aof_bio_fsync_errno, errno);
            if (last_status == C_OK) {
                serverLog(LL_WARNING,
                    "Fail to fsync the AOF file: %s", strerror(errno));
            }
        } else {
            atomicSet(server.aof_bio_fsync_status, C_OK);
        }
    } else if (type == BIO_LAZY_FREE) {
        job->free_fn(job->free_args);
    } else {
        serverPanic("Wrong job type in bioProcessBackgroundJobs().");
    }
    zfree(job);

    /* Lock again before reiterating the loop, if there are no longer
     * jobs to process we'll block again in pthread_cond_wait(). */
    pthread_mutex_lock(&bio_mutex[type]);
    listDelNode(bio_jobs[type], ln);
    bio_pending[type]--;

    /* Unblock threads blocked on bioWaitStepOfType() if any. */

```

```

        pthread_cond_broadcast(&bio_step_cond[type]);
    }
}

/* Return the number of pending jobs of the specified type. */
unsigned long long bioPendingJobsOfType(int type) {
    unsigned long long val;
    pthread_mutex_lock(&bio_mutex[type]);
    val = bio_pending[type];
    pthread_mutex_unlock(&bio_mutex[type]);
    return val;
}

/* If there are pending jobs for the specified type, the function blocks
 * and waits that the next job was processed. Otherwise the function
 * does not block and returns ASAP.
 *
 * The function returns the number of jobs still to process of the
 * requested type.
 *
 * This function is useful when from another thread, we want to wait
 * a bio.c thread to do more work in a blocking way.
 */
unsigned long long bioWaitStepOfType(int type) {
    unsigned long long val;
    pthread_mutex_lock(&bio_mutex[type]);
    val = bio_pending[type];
    if (val != 0) {
        pthread_cond_wait(&bio_step_cond[type], &bio_mutex[type]);
        val = bio_pending[type];
    }
    pthread_mutex_unlock(&bio_mutex[type]);
    return val;
}

/* Kill the running bio threads in an unclean way. This function should be
 * used only when it's critical to stop the threads for some reason.
 * Currently Redis does this only on crash (for instance on SIGSEGV) in order
 * to perform a fast memory check without other threads messing with memory. */
void bioKillThreads(void) {
    int err, j;

    for (j = 0; j < BIO_NUM_OPS; j++) {
        if (bio_threads[j] == pthread_self()) continue;
        if (bio_threads[j] && pthread_cancel(bio_threads[j]) == 0) {
            if ((err = pthread_join(bio_threads[j], NULL)) != 0) {
                serverLog(LL_WARNING,
                    "Bio thread for job type #%d can not be joined: %s",
                    j, strerror(err));
            } else {
                serverLog(LL_WARNING,

```

```
        "Bio thread for job type #%"d terminated",j));  
    }  
}  
}
```

/bio.h

[to top](#)

```

/*
 * Copyright (c) 2009–2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef __BIO_H
#define __BIO_H

typedef void lazy_free_fn(void *args[]);

/* Exported API */
void bioInit(void);
unsigned long long bioPendingJobsOfType(int type);
unsigned long long bioWaitStepOfType(int type);
void bioKillThreads(void);
void bioCreateCloseJob(int fd);
void bioCreateFsyncJob(int fd);
void bioCreateLazyFreeJob(lazy_free_fn free_fn, int arg_count, ...);

/* Background job opcodes */
#define BIO_CLOSE_FILE      0 /* Deferred close(2) syscall. */
#define BIO_AOF_FSYNC       1 /* Deferred AOF fsync. */
#define BIO_LAZY_FREE       2 /* Deferred objects freeing. */
#define BIO_NUM_OPS         3

```

```
#endif
```

/bitops.c

[to top](#)

```

/* Bit operations.
 *
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "server.h"

/* -----
 *
 * Helpers and low level bit functions.
 * -----
 */

/* Count number of bits set in the binary array pointed by 's' and long
 * 'count' bytes. The implementation of this function is required to
 * work with an input string length up to 512 MB or more
(server.proto_max_bulk_len) */
long long redisPopcount(void *s, long count) {
    long long bits = 0;
    unsigned char *p = s;
    uint32_t *p4;
    static const unsigned char bitsinbyte[256] =
{0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,1,2,2,3,2,3,3,4,2
    /* Count initial bytes not aligned to 32 bit. */

```



```

while((unsigned long)p & 3 && count) {
    bits += bitsinbyte[*p++];
    count--;
}

/* Count bits 28 bytes at a time */
p4 = (uint32_t*)p;
while(count >= 28) {
    uint32_t aux1, aux2, aux3, aux4, aux5, aux6, aux7;

    aux1 = *p4++;
    aux2 = *p4++;
    aux3 = *p4++;
    aux4 = *p4++;
    aux5 = *p4++;
    aux6 = *p4++;
    aux7 = *p4++;
    count -= 28;

    aux1 = aux1 - ((aux1 >> 1) & 0x55555555);
    aux1 = (aux1 & 0x33333333) + ((aux1 >> 2) & 0x33333333);
    aux2 = aux2 - ((aux2 >> 1) & 0x55555555);
    aux2 = (aux2 & 0x33333333) + ((aux2 >> 2) & 0x33333333);
    aux3 = aux3 - ((aux3 >> 1) & 0x55555555);
    aux3 = (aux3 & 0x33333333) + ((aux3 >> 2) & 0x33333333);
    aux4 = aux4 - ((aux4 >> 1) & 0x55555555);
    aux4 = (aux4 & 0x33333333) + ((aux4 >> 2) & 0x33333333);
    aux5 = aux5 - ((aux5 >> 1) & 0x55555555);
    aux5 = (aux5 & 0x33333333) + ((aux5 >> 2) & 0x33333333);
    aux6 = aux6 - ((aux6 >> 1) & 0x55555555);
    aux6 = (aux6 & 0x33333333) + ((aux6 >> 2) & 0x33333333);
    aux7 = aux7 - ((aux7 >> 1) & 0x55555555);
    aux7 = (aux7 & 0x33333333) + ((aux7 >> 2) & 0x33333333);
    bits += (((aux1 + (aux1 >> 4)) & 0x0F0F0F0F) +
             ((aux2 + (aux2 >> 4)) & 0x0F0F0F0F) +
             ((aux3 + (aux3 >> 4)) & 0x0F0F0F0F) +
             ((aux4 + (aux4 >> 4)) & 0x0F0F0F0F) +
             ((aux5 + (aux5 >> 4)) & 0x0F0F0F0F) +
             ((aux6 + (aux6 >> 4)) & 0x0F0F0F0F) +
             ((aux7 + (aux7 >> 4)) & 0x0F0F0F0F)) * 0x01010101) >> 24;
}

/* Count the remaining bytes. */
p = (unsigned char*)p4;
while(count-->0) bits += bitsinbyte[*p++];
return bits;
}

/* Return the position of the first bit set to one (if 'bit' is 1) or
 * zero (if 'bit' is 0) in the bitmap starting at 's' and long 'count' bytes.
 *
 * The function is guaranteed to return a value >= 0 if 'bit' is 0 since if

```

```

* no zero bit is found, it returns count*8 assuming the string is zero
* padded on the right. However if 'bit' is 1 it is possible that there is
* not a single set bit in the bitmap. In this special case -1 is returned. */
long long redisBitpos(void *s, unsigned long count, int bit) {
    unsigned long *l;
    unsigned char *c;
    unsigned long skipval, word = 0, one;
    long long pos = 0; /* Position of bit, to return to the caller. */
    unsigned long j;
    int found;

    /* Process whole words first, seeking for first word that is not
    * all ones or all zeros respectively if we are looking for zeros
    * or ones. This is much faster with large strings having contiguous
    * blocks of 1 or 0 bits compared to the vanilla bit per bit processing.
    *
    * Note that if we start from an address that is not aligned
    * to sizeof(unsigned long) we consume it byte by byte until it is
    * aligned. */

    /* Skip initial bits not aligned to sizeof(unsigned long) byte by byte. */
    skipval = bit ? 0 : UCHAR_MAX;
    c = (unsigned char*) s;
    found = 0;
    while(((unsigned long)c & (sizeof(*l)-1) && count) {
        if (*c != skipval) {
            found = 1;
            break;
        }
        c++;
        count--;
        pos += 8;
    }

    /* Skip bits with full word step. */
    l = (unsigned long*) c;
    if (!found) {
        skipval = bit ? 0 : ULONG_MAX;
        while (count >= sizeof(*l)) {
            if (*l != skipval) break;
            l++;
            count -= sizeof(*l);
            pos += sizeof(*l)*8;
        }
    }

    /* Load bytes into "word" considering the first byte as the most
    significant
    * (we basically consider it as written in big endian, since we consider
    the
    * string as a set of bits from left to right, with the first bit at

```

```

position
    * zero.
    *
    * Note that the loading is designed to work even when the bytes left
    * (count) are less than a full word. We pad it with zero on the right. */
c = (unsigned char*)l;
for (j = 0; j < sizeof(*l); j++) {
    word <= 8;
    if (count) {
        word |= *c;
        c++;
        count--;
    }
}

/* Special case:
 * If bits in the string are all zero and we are looking for one,
 * return -1 to signal that there is not a single "1" in the whole
 * string. This can't happen when we are looking for "0" as we assume
 * that the right of the string is zero padded. */
if (bit == 1 && word == 0) return -1;

/* Last word left, scan bit by bit. The first thing we need is to
 * have a single "1" set in the most significant position in an
 * unsigned long. We don't know the size of the long so we use a
 * simple trick. */
one = ULONG_MAX; /* All bits set to 1.*/
one >>= 1;        /* All bits set to 1 but the MSB. */
one = ~one;       /* All bits set to 0 but the MSB. */

while(one) {
    if (((one & word) != 0) == bit) return pos;
    pos++;
    one >>= 1;
}

/* If we reached this point, there is a bug in the algorithm, since
 * the case of no match is handled as a special case before. */
serverPanic("End of redisBitpos() reached.");
return 0; /* Just to avoid warnings. */
}

/* The following set.*Bitfield and get.*Bitfield functions implement setting
 * and getting arbitrary size (up to 64 bits) signed and unsigned integers
 * at arbitrary positions into a bitmap.
 *
 * The representation considers the bitmap as having the bit number 0 to be
 * the most significant bit of the first byte, and so forth, so for example
 * setting a 5 bits unsigned integer to value 23 at offset 7 into a bitmap
 * previously set to all zeroes, will produce the following representation:
 *

```

```

* +-----+-----+
* |00000001|01110000|
* +-----+-----+
*
* When offsets and integer sizes are aligned to bytes boundaries, this is the
* same as big endian, however when such alignment does not exist, its
important
* to also understand how the bits inside a byte are ordered.
*
* Note that this format follows the same convention as SETBIT and related
* commands.
*/

void setUnsignedBitfield(unsigned char *p, uint64_t offset, uint64_t bits,
uint64_t value) {
    uint64_t byte, bit, byteval, bitval, j;

    for (j = 0; j < bits; j++) {
        bitval = (value & ((uint64_t)1<<(bits-1-j))) != 0;
        byte = offset >> 3;
        bit = 7 - (offset & 0x7);
        byteval = p[byte];
        byteval &= ~(1 << bit);
        byteval |= bitval << bit;
        p[byte] = byteval & 0xff;
        offset++;
    }
}

void setSignedBitfield(unsigned char *p, uint64_t offset, uint64_t bits,
int64_t value) {
    uint64_t uv = value; /* Casting will add UINT64_MAX + 1 if v is negative.
*/
    setUnsignedBitfield(p, offset, bits, uv);
}

uint64_t getUnsignedBitfield(unsigned char *p, uint64_t offset, uint64_t bits)
{
    uint64_t byte, bit, byteval, bitval, j, value = 0;

    for (j = 0; j < bits; j++) {
        byte = offset >> 3;
        bit = 7 - (offset & 0x7);
        byteval = p[byte];
        bitval = (byteval >> bit) & 1;
        value = (value<<1) | bitval;
        offset++;
    }
    return value;
}

```

```

int64_t getSignedBitfield(unsigned char *p, uint64_t offset, uint64_t bits) {
    int64_t value;
    union {uint64_t u; int64_t i;} conv;

    /* Converting from unsigned to signed is undefined when the value does
     * not fit, however here we assume two's complement and the original value
     * was obtained from signed -> unsigned conversion, so we'll find the
     * most significant bit set if the original value was negative.
     *
     * Note that two's complement is mandatory for exact-width types
     * according to the C99 standard. */
    conv.u = getUnsignedBitfield(p, offset, bits);
    value = conv.i;

    /* If the top significant bit is 1, propagate it to all the
     * higher bits for two's complement representation of signed
     * integers. */
    if (bits < 64 && (value & ((uint64_t)1 << (bits-1))))
        value |= ((uint64_t)-1) << bits;
    return value;
}

/* The following two functions detect overflow of a value in the context
 * of storing it as an unsigned or signed integer with the specified
 * number of bits. The functions both take the value and a possible increment.
 * If no overflow could happen and the value+increment fit inside the limits,
 * then zero is returned, otherwise in case of overflow, 1 is returned,
 * otherwise in case of underflow, -1 is returned.
 *
 * When non-zero is returned (overflow or underflow), if not NULL, *limit is
 * set to the value the operation should result when an overflow happens,
 * depending on the specified overflow semantics:
 *
 * For BFOVERFLOW_SAT if 1 is returned, *limit it is set maximum value that
 * you can store in that integer. when -1 is returned, *limit is set to the
 * minimum value that an integer of that size can represent.
 *
 * For BFOVERFLOW_WRAP *limit is set by performing the operation in order to
 * "wrap" around towards zero for unsigned integers, or towards the most
 * negative number that is possible to represent for signed integers. */

#define BFOVERFLOW_WRAP 0
#define BFOVERFLOW_SAT 1
#define BFOVERFLOW_FAIL 2 /* Used by the BITFIELD command implementation. */

int checkUnsignedBitfieldOverflow(uint64_t value, int64_t incr, uint64_t bits,
int otype, uint64_t *limit) {
    uint64_t max = (bits == 64) ? UINT64_MAX : (((uint64_t)1<<bits)-1);
    int64_t maxincr = max-value;
    int64_t minincr = -value;

```

```

if (value > max || (incr > 0 && incr > maxincr)) {
    if (limit) {
        if (owtype == BFOVERFLOW_WRAP) {
            goto handle_wrap;
        } else if (owtype == BFOVERFLOW_SAT) {
            *limit = max;
        }
    }
    return 1;
} else if (incr < 0 && incr < minincr) {
    if (limit) {
        if (owtype == BFOVERFLOW_WRAP) {
            goto handle_wrap;
        } else if (owtype == BFOVERFLOW_SAT) {
            *limit = 0;
        }
    }
    return -1;
}
return 0;

handle_wrap:
{
    uint64_t mask = ((uint64_t)-1) << bits;
    uint64_t res = value+incr;

    res &= ~mask;
    *limit = res;
}
return 1;
}

int checkSignedBitfieldOverflow(int64_t value, int64_t incr, uint64_t bits, int
owtype, int64_t *limit) {
    int64_t max = (bits == 64) ? INT64_MAX : (((int64_t)1<<(bits-1))-1);
    int64_t min = (-max)-1;

    /* Note that maxincr and minincr could overflow, but we use the values
     * only after checking 'value' range, so when we use it no overflow
     * happens. 'uint64_t' cast is there just to prevent undefined behavior on
     * overflow */
    int64_t maxincr = (uint64_t)max-value;
    int64_t minincr = min-value;

    if (value > max || (bits != 64 && incr > maxincr) || (value >= 0 && incr >
0 && incr > maxincr))
    {
        if (limit) {
            if (owtype == BFOVERFLOW_WRAP) {
                goto handle_wrap;
            } else if (owtype == BFOVERFLOW_SAT) {

```

```

        *limit = max;
    }
}
return 1;
} else if (value < min || (bits != 64 && incr < minincr) || (value < 0 &&
incr < 0 && incr < minincr)) {
    if (limit) {
        if (owtype == BFOVERFLOW_WRAP) {
            goto handle_wrap;
        } else if (owtype == BFOVERFLOW_SAT) {
            *limit = min;
        }
    }
    return -1;
}
return 0;

```

handle_wrap:

```

{
    uint64_t msb = (uint64_t)1 << (bits-1);
    uint64_t a = value, b = incr, c;
    c = a+b; /* Perform addition as unsigned so that's defined. */

    /* If the sign bit is set, propagate to all the higher order
     * bits, to cap the negative value. If it's clear, mask to
     * the positive integer limit. */
    if (bits < 64) {
        uint64_t mask = ((uint64_t)-1) << bits;
        if (c & msb) {
            c |= mask;
        } else {
            c &= ~mask;
        }
    }
    *limit = c;
}
return 1;
}

```

/* Debugging function. Just show bits in the specified bitmap. Not used
* but here for not having to rewrite it when debugging is needed. */

```

void printBits(unsigned char *p, unsigned long count) {
    unsigned long j, i, byte;

    for (j = 0; j < count; j++) {
        byte = p[j];
        for (i = 0x80; i > 0; i /= 2)
            printf("%c", (byte & i) ? '1' : '0');
        printf("|");
    }
    printf("\n");
}

```

```

}

/* -----
-
* Bits related string commands: GETBIT, SETBIT, BITCOUNT, BITOP.
* -----
*/

#define BITOP_AND    0
#define BITOP_OR     1
#define BITOP_XOR    2
#define BITOP_NOT    3

#define BITFIELDOP_GET 0
#define BITFIELDOP_SET 1
#define BITFIELDOP_INCRBY 2

/* This helper function used by GETBIT / SETBIT parses the bit offset argument
 * making sure an error is returned if it is negative or if it overflows
 * Redis 512 MB limit for the string value or more (server.proto_max_bulk_len).
 *
 * If the 'hash' argument is true, and 'bits' is positive, then the command
 * will also parse bit offsets prefixed by "#". In such a case the offset
 * is multiplied by 'bits'. This is useful for the BITFIELD command. */
int getBitOffsetFromArgument(client *c, robj *o, uint64_t *offset, int hash,
int bits) {
    long long loffset;
    char *err = "bit offset is not an integer or out of range";
    char *p = o->ptr;
    size_t plen = sdslen(p);
    int usehash = 0;

    /* Handle #<offset> form. */
    if (p[0] == '#' && hash && bits > 0) usehash = 1;

    if (string2ll(p+usehash,plen-usehash,&loffset) == 0) {
        addReplyError(c,err);
        return C_ERR;
    }

    /* Adjust the offset by 'bits' for #<offset> form. */
    if (usehash) loffset *= bits;

    /* Limit offset to server.proto_max_bulk_len (512MB in bytes by default) */
    if (loffset < 0 || (!mustObeyClient(c) && (loffset >> 3) >=
server.proto_max_bulk_len))
    {
        addReplyError(c,err);
        return C_ERR;
    }
}

```



```

    *offset = loffset;
    return C_OK;
}

/* This helper function for BITFIELD parses a bitfield type in the form
 * <sign><bits> where sign is 'u' or 'i' for unsigned and signed, and
 * the bits is a value between 1 and 64. However 64 bits unsigned integers
 * are reported as an error because of current limitations of Redis protocol
 * to return unsigned integer values greater than INT64_MAX.
 *
 * On error C_ERR is returned and an error is sent to the client. */
int getBitfieldTypeFromArgument(client *c, robj *o, int *sign, int *bits) {
    char *p = o->ptr;
    char *err = "Invalid bitfield type. Use something like i16 u8. Note that
u64 is not supported but i64 is.";
    long long llbits;

    if (p[0] == 'i') {
        *sign = 1;
    } else if (p[0] == 'u') {
        *sign = 0;
    } else {
        addReplyError(c,err);
        return C_ERR;
    }

    if ((string2ll(p+1,strlen(p+1),&llbits)) == 0 ||
        llbits < 1 ||
        (*sign == 1 && llbits > 64) ||
        (*sign == 0 && llbits > 63))
    {
        addReplyError(c,err);
        return C_ERR;
    }
    *bits = llbits;
    return C_OK;
}

/* This is a helper function for commands implementations that need to write
 * bits to a string object. The command creates or pad with zeroes the string
 * so that the 'maxbit' bit can be addressed. The object is finally
 * returned. Otherwise if the key holds a wrong type NULL is returned and
 * an error is sent to the client. */
robj *lookupStringForBitCommand(client *c, uint64_t maxbit, int *dirty) {
    size_t byte = maxbit >> 3;
    robj *o = lookupKeyWrite(c->db,c->argv[1]);
    if (checkType(c,o,OBJ_STRING)) return NULL;
    if (dirty) *dirty = 0;

    if (o == NULL) {
        o = createObject(OBJ_STRING,sdsnewlen(NULL, byte+1));
    }
}

```

```

        dbAdd(c->db,c->argv[1],o);
        if (dirty) *dirty = 1;
    } else {
        o = dbUnshareStringValue(c->db,c->argv[1],o);
        size_t oldlen = sdslen(o->ptr);
        o->ptr = sdsgrowzero(o->ptr,byte+1);
        if (dirty && oldlen != sdslen(o->ptr)) *dirty = 1;
    }
    return o;
}

/* Return a pointer to the string object content, and stores its length
 * in 'len'. The user is required to pass (likely stack allocated) buffer
 * 'llbuf' of at least LONG_STR_SIZE bytes. Such a buffer is used in the case
 * the object is integer encoded in order to provide the representation
 * without using heap allocation.
 *
 * The function returns the pointer to the object array of bytes representing
 * the string it contains, that may be a pointer to 'llbuf' or to the
 * internal object representation. As a side effect 'len' is filled with
 * the length of such buffer.
 *
 * If the source object is NULL the function is guaranteed to return NULL
 * and set 'len' to 0. */
unsigned char *getObjectReadOnlyString(robj *o, long *len, char *llbuf) {
    serverAssert(!o || o->type == OBJ_STRING);
    unsigned char *p = NULL;

    /* Set the 'p' pointer to the string, that can be just a stack allocated
     * array if our string was integer encoded. */
    if (o && o->encoding == OBJ_ENCODING_INT) {
        p = (unsigned char*) llbuf;
        if (len) *len = ll2string(llbuf,LONG_STR_SIZE,(long)o->ptr);
    } else if (o) {
        p = (unsigned char*) o->ptr;
        if (len) *len = sdslen(o->ptr);
    } else {
        if (len) *len = 0;
    }
    return p;
}

/* SETBIT key offset bitvalue */
void setbitCommand(client *c) {
    robj *o;
    char *err = "bit is not an integer or out of range";
    uint64_t bitoffset;
    ssize_t byte, bit;
    int byteval, bitval;
    long on;

```

```

if (getBitOffsetFromArgument(c,c->argv[2],&bitoffset,0,0) != C_OK)
    return;

if (getLongFromObjectOrReply(c,c->argv[3],&on,err) != C_OK)
    return;

/* Bits can only be set or cleared... */
if (on & ~1) {
    addReplyError(c,err);
    return;
}

int dirty;
if ((o = lookupStringForBitCommand(c,bitoffset,&dirty)) == NULL) return;

/* Get current values */
byte = bitoffset >> 3;
byteval = ((uint8_t*)o->ptr)[byte];
bit = 7 - (bitoffset & 0x7);
bitval = byteval & (1 << bit);

/* Either it is newly created, changed length, or the bit changes before
and after.
* Note that the bitval here is actually a decimal number.
* So we need to use `!!` to convert it to 0 or 1 for comparison. */
if (dirty || (!!bitval != on)) {
    /* Update byte with new bit value. */
    byteval &= ~(1 << bit);
    byteval |= ((on & 0x1) << bit);
    ((uint8_t*)o->ptr)[byte] = byteval;
    signalModifiedKey(c,c->db,c->argv[1]);
    notifyKeyspaceEvent(NOTIFY_STRING,"setbit",c->argv[1],c->db->id);
    server.dirty++;
}

/* Return original value. */
addReply(c, bitval ? shared.cone : shared.czero);
}

/* GETBIT key offset */
void getbitCommand(client *c) {
    robj *o;
    char llbuf[32];
    uint64_t bitoffset;
    size_t byte, bit;
    size_t bitval = 0;

    if (getBitOffsetFromArgument(c,c->argv[2],&bitoffset,0,0) != C_OK)
        return;

    if ((o = lookupKeyReadOrReply(c,c->argv[1],shared.czero)) == NULL ||

```

```

        checkType(c,o,OBJ_STRING)) return;

    byte = bitoffset >> 3;
    bit = 7 - (bitoffset & 0x7);
    if (sdsEncodedObject(o)) {
        if (byte < sdslen(o->ptr))
            bitval = ((uint8_t*)o->ptr)[byte] & (1 << bit);
    } else {
        if (byte < (size_t)ll2string(llbuf,sizeof(llbuf),(long)o->ptr))
            bitval = llbuf[byte] & (1 << bit);
    }

    addReply(c, bitval ? shared.cone : shared.czero);
}

/* BITOP op_name target_key src_key1 src_key2 src_key3 ... src_keyN */
REDIS_NO_SANITIZE("alignment")
void bitopCommand(client *c) {
    char *opname = c->argv[1]->ptr;
    robj *o, *targetkey = c->argv[2];
    unsigned long op, j, numkeys;
    robj **objects; /* Array of source objects. */
    unsigned char **src; /* Array of source strings pointers. */
    unsigned long *len, maxlen = 0; /* Array of length of src strings,
                                     and max len. */
    unsigned long minlen = 0; /* Min len among the input keys. */
    unsigned char *res = NULL; /* Resulting string. */

    /* Parse the operation name. */
    if ((opname[0] == 'a' || opname[0] == 'A') && !strcasecmp(opname,"and"))
        op = BITOP_AND;
    else if((opname[0] == 'o' || opname[0] == 'O') && !strcasecmp(opname,"or"))
        op = BITOP_OR;
    else if((opname[0] == 'x' || opname[0] == 'X') &&
!strcasecmp(opname,"xor"))
        op = BITOP_XOR;
    else if((opname[0] == 'n' || opname[0] == 'N') &&
!strcasecmp(opname,"not"))
        op = BITOP_NOT;
    else {
        addReplyErrorObject(c,shared.syntaxerr);
        return;
    }

    /* Sanity check: NOT accepts only a single key argument. */
    if (op == BITOP_NOT && c->argc != 4) {
        addReplyError(c,"BITOP NOT must be called with a single source key.");
        return;
    }

    /* Lookup keys, and store pointers to the string objects into an array. */

```

```

numkeys = c->argc - 3;
src = zmalloc(sizeof(unsigned char*) * numkeys);
len = zmalloc(sizeof(long) * numkeys);
objects = zmalloc(sizeof(robj*) * numkeys);
for (j = 0; j < numkeys; j++) {
    o = lookupKeyRead(c->db,c->argv[j+3]);
    /* Handle non-existing keys as empty strings. */
    if (o == NULL) {
        objects[j] = NULL;
        src[j] = NULL;
        len[j] = 0;
        minlen = 0;
        continue;
    }
    /* Return an error if one of the keys is not a string. */
    if (checkType(c,o,OBJ_STRING)) {
        unsigned long i;
        for (i = 0; i < j; i++) {
            if (objects[i])
                decrRefCount(objects[i]);
        }
        zfree(src);
        zfree(len);
        zfree(objects);
        return;
    }
    objects[j] = getDecodedObject(o);
    src[j] = objects[j]->ptr;
    len[j] = sdslen(objects[j]->ptr);
    if (len[j] > maxlen) maxlen = len[j];
    if (j == 0 || len[j] < minlen) minlen = len[j];
}

/* Compute the bit operation, if at least one string is not empty. */
if (maxlen) {
    res = (unsigned char*) sdsnewlen(NULL,maxlen);
    unsigned char output, byte;
    unsigned long i;

    /* Fast path: as far as we have data for all the input bitmaps we
     * can take a fast path that performs much better than the
     * vanilla algorithm. On ARM we skip the fast path since it will
     * result in GCC compiling the code using multiple-words load/store
     * operations that are not supported even in ARM >= v6. */
    j = 0;
#ifdef USE_ALIGNED_ACCESS
    if (minlen >= sizeof(unsigned long)*4 && numkeys <= 16) {
        unsigned long *lp[16];
        unsigned long *lres = (unsigned long*) res;

        memcpy(lp,src,sizeof(unsigned long)*numkeys);

```

```

memcpy(res,src[0],minlen);

/* Different branches per different operations for speed (sorry).
*/
if (op == BITOP_AND) {
    while(minlen >= sizeof(unsigned long)*4) {
        for (i = 1; i < numkeys; i++) {
            lres[0] &= lp[i][0];
            lres[1] &= lp[i][1];
            lres[2] &= lp[i][2];
            lres[3] &= lp[i][3];
            lp[i]+=4;
        }
        lres+=4;
        j += sizeof(unsigned long)*4;
        minlen -= sizeof(unsigned long)*4;
    }
} else if (op == BITOP_OR) {
    while(minlen >= sizeof(unsigned long)*4) {
        for (i = 1; i < numkeys; i++) {
            lres[0] |= lp[i][0];
            lres[1] |= lp[i][1];
            lres[2] |= lp[i][2];
            lres[3] |= lp[i][3];
            lp[i]+=4;
        }
        lres+=4;
        j += sizeof(unsigned long)*4;
        minlen -= sizeof(unsigned long)*4;
    }
} else if (op == BITOP_XOR) {
    while(minlen >= sizeof(unsigned long)*4) {
        for (i = 1; i < numkeys; i++) {
            lres[0] ^= lp[i][0];
            lres[1] ^= lp[i][1];
            lres[2] ^= lp[i][2];
            lres[3] ^= lp[i][3];
            lp[i]+=4;
        }
        lres+=4;
        j += sizeof(unsigned long)*4;
        minlen -= sizeof(unsigned long)*4;
    }
} else if (op == BITOP_NOT) {
    while(minlen >= sizeof(unsigned long)*4) {
        lres[0] = ~lres[0];
        lres[1] = ~lres[1];
        lres[2] = ~lres[2];
        lres[3] = ~lres[3];
        lres+=4;
        j += sizeof(unsigned long)*4;
    }
}

```

```

        minlen -= sizeof(unsigned long)*4;
    }
}
#endif

/* j is set to the next byte to process by the previous loop. */
for (; j < maxlen; j++) {
    output = (len[0] <= j) ? 0 : src[0][j];
    if (op == BITOP_NOT) output = ~output;
    for (i = 1; i < numkeys; i++) {
        int skip = 0;
        byte = (len[i] <= j) ? 0 : src[i][j];
        switch(op) {
            case BITOP_AND:
                output &= byte;
                skip = (output == 0);
                break;
            case BITOP_OR:
                output |= byte;
                skip = (output == 0xff);
                break;
            case BITOP_XOR: output ^= byte; break;
        }

        if (skip) {
            break;
        }
    }
    res[j] = output;
}
}
for (j = 0; j < numkeys; j++) {
    if (objects[j])
        decrRefCount(objects[j]);
}
zfree(src);
zfree(len);
zfree(objects);

/* Store the computed value into the target key */
if (maxlen) {
    o = createObject(OBJ_STRING, res);
    setKey(c, c->db, targetkey, o, 0);
    notifyKeyspaceEvent(NOTIFY_STRING, "set", targetkey, c->db->id);
    decrRefCount(o);
    server.dirty++;
} else if (dbDelete(c->db, targetkey)) {
    signalModifiedKey(c, c->db, targetkey);
    notifyKeyspaceEvent(NOTIFY_GENERIC, "del", targetkey, c->db->id);
    server.dirty++;
}

```

```

    }
    addReplyLongLong(c,maxlen); /* Return the output string length in bytes. */
}

/* BITCOUNT key [start end [BIT|BYTE]] */
void bitcountCommand(client *c) {
    robj *o;
    long long start, end;
    long strlen;
    unsigned char *p;
    char llbuf[LONG_STR_SIZE];
    int isbit = 0;
    unsigned char first_byte_neg_mask = 0, last_byte_neg_mask = 0;

    /* Lookup, check for type, and return 0 for non existing keys. */
    if ((o = lookupKeyReadOrReply(c,c->argv[1],shared.czero)) == NULL ||
        checkType(c,o,OBJ_STRING)) return;
    p = getObjectReadOnlyString(o,&strlen,llbuf);

    /* Parse start/end range if any. */
    if (c->argc == 4 || c->argc == 5) {
        long long totlen = strlen;
        /* Make sure we will not overflow */
        serverAssert(totlen <= LLONG_MAX >> 3);
        if (getLongLongFromObjectOrReply(c,c->argv[2],&start,NULL) != C_OK)
            return;
        if (getLongLongFromObjectOrReply(c,c->argv[3],&end,NULL) != C_OK)
            return;
        /* Convert negative indexes */
        if (start < 0 && end < 0 && start > end) {
            addReply(c,shared.czero);
            return;
        }
        if (c->argc == 5) {
            if (!strcasecmp(c->argv[4]->ptr,"bit")) isbit = 1;
            else if (!strcasecmp(c->argv[4]->ptr,"byte")) isbit = 0;
            else {
                addReplyErrorObject(c,shared.syntaxerr);
                return;
            }
        }
        if (isbit) totlen <= 3;
        if (start < 0) start = totlen+start;
        if (end < 0) end = totlen+end;
        if (start < 0) start = 0;
        if (end < 0) end = 0;
        if (end >= totlen) end = totlen-1;
        if (isbit && start <= end) {
            /* Before converting bit offset to byte offset, create negative
            masks
            * for the edges. */

```



```

        first_byte_neg_mask = ~((1<<(8-(start&7)))-1) & 0xFF;
        last_byte_neg_mask = (1<<(7-(end&7)))-1;
        start >>= 3;
        end >>= 3;
    }
} else if (c->argc == 2) {
    /* The whole string. */
    start = 0;
    end = strlen-1;
} else {
    /* Syntax error. */
    addReplyErrorObject(c,shared.syntaxerr);
    return;
}

/* Precondition: end >= 0 && end < strlen, so the only condition where
 * zero can be returned is: start > end. */
if (start > end) {
    addReply(c,shared.czero);
} else {
    long bytes = (long)(end-start+1);
    long long count = redisPopcount(p+start,bytes);
    if (first_byte_neg_mask != 0 || last_byte_neg_mask != 0) {
        unsigned char firstlast[2] = {0, 0};
        /* We may count bits of first byte and last byte which are out of
         * range. So we need to subtract them. Here we use a trick. We set
         * bits in the range to zero. So these bit will not be excluded. */
        if (first_byte_neg_mask != 0) firstlast[0] = p[start] &
first_byte_neg_mask;
        if (last_byte_neg_mask != 0) firstlast[1] = p[end] &
last_byte_neg_mask;
        count -= redisPopcount(firstlast,2);
    }
    addReplyLongLong(c,count);
}
}

/* BITPOS key bit [start [end [BIT|BYTE]]] */
void bitposCommand(client *c) {
    robj *o;
    long long start, end;
    long bit, strlen;
    unsigned char *p;
    char llbuf[LONG_STR_SIZE];
    int isbit = 0, end_given = 0;
    unsigned char first_byte_neg_mask = 0, last_byte_neg_mask = 0;

    /* Parse the bit argument to understand what we are looking for, set
     * or clear bits. */
    if (getLongFromObjectOrReply(c,c->argv[2],&bit,NULL) != C_OK)
        return;

```

```

if (bit != 0 && bit != 1) {
    addReplyError(c, "The bit argument must be 1 or 0.");
    return;
}

/* If the key does not exist, from our point of view it is an infinite
 * array of 0 bits. If the user is looking for the first clear bit return
0,
 * If the user is looking for the first set bit, return -1. */
if ((o = lookupKeyRead(c->db,c->argv[1])) == NULL) {
    addReplyLongLong(c, bit ? -1 : 0);
    return;
}
if (checkType(c,o,OBJ_STRING)) return;
p = getObjectReadOnlyString(o,&strlen,llbuf);

/* Parse start/end range if any. */
if (c->argc == 4 || c->argc == 5 || c->argc == 6) {
    long long totlen = strlen;
    /* Make sure we will not overflow */
    serverAssert(totlen <= LLONG_MAX >> 3);
    if (getLongLongFromObjectOrReply(c,c->argv[3],&start,NULL) != C_OK)
        return;
    if (c->argc == 6) {
        if (!strcasecmp(c->argv[5]->ptr,"bit")) isbit = 1;
        else if (!strcasecmp(c->argv[5]->ptr,"byte")) isbit = 0;
        else {
            addReplyErrorObject(c,shared.syntaxerr);
            return;
        }
    }
    if (c->argc >= 5) {
        if (getLongLongFromObjectOrReply(c,c->argv[4],&end,NULL) != C_OK)
            return;
        end_given = 1;
    } else {
        if (isbit) end = (totlen<<3) + 7;
        else end = totlen-1;
    }
    if (isbit) totlen <= 3;
    /* Convert negative indexes */
    if (start < 0) start = totlen+start;
    if (end < 0) end = totlen+end;
    if (start < 0) start = 0;
    if (end < 0) end = 0;
    if (end >= totlen) end = totlen-1;
    if (isbit && start <= end) {
        /* Before converting bit offset to byte offset, create negative
masks
        * for the edges. */
        first_byte_neg_mask = ~((1<<(8-(start&7)))-1) & 0xFF;

```

```

        last_byte_neg_mask = (1<<(7-(end&7)))-1;
        start >= 3;
        end >= 3;
    }
} else if (c->argc == 3) {
    /* The whole string. */
    start = 0;
    end = strlen-1;
} else {
    /* Syntax error. */
    addReplyErrorObject(c,shared.syntaxerr);
    return;
}

/* For empty ranges (start > end) we return -1 as an empty range does
 * not contain a 0 nor a 1. */
if (start > end) {
    addReplyLongLong(c, -1);
} else {
    long bytes = end-start+1;
    long long pos;
    unsigned char tmpchar;
    if (first_byte_neg_mask) {
        if (bit) tmpchar = p[start] & ~first_byte_neg_mask;
        else tmpchar = p[start] | first_byte_neg_mask;
        /* Special case, there is only one byte */
        if (last_byte_neg_mask && bytes == 1) {
            if (bit) tmpchar = tmpchar & ~last_byte_neg_mask;
            else tmpchar = tmpchar | last_byte_neg_mask;
        }
        pos = redisBitpos(&tmpchar,1,bit);
        /* If there are no more bytes or we get valid pos, we can exit
early */
        if (bytes == 1 || (pos != -1 && pos != 8)) goto result;
        start++;
        bytes--;
    }
    /* If the last byte has not bits in the range, we should exclude it */
    long curbytes = bytes - (last_byte_neg_mask ? 1 : 0);
    if (curbytes > 0) {
        pos = redisBitpos(p+start,curbytes,bit);
        /* If there is no more bytes or we get valid pos, we can exit early
*/
        if (bytes == curbytes || (pos != -1 && pos != (long
long)curbytes<<3)) goto result;
        start += curbytes;
        bytes -= curbytes;
    }
    if (bit) tmpchar = p[end] & ~last_byte_neg_mask;
    else tmpchar = p[end] | last_byte_neg_mask;
    pos = redisBitpos(&tmpchar,1,bit);

```

```

result:
    /* If we are looking for clear bits, and the user specified an exact
     * range with start-end, we can't consider the right of the range as
     * zero padded (as we do when no explicit end is given).
     *
     * So if redisBitpos() returns the first bit outside the range,
     * we return -1 to the caller, to mean, in the specified range there
     * is not a single "0" bit. */
    if (end_given && bit == 0 && pos == (long long)bytes<<3) {
        addReplyLongLong(c,-1);
        return;
    }
    if (pos != -1) pos += (long long)start<<3; /* Adjust for the bytes we
skipped. */
    addReplyLongLong(c,pos);
}
}

/* BITFIELD key subcommand-1 arg ... subcommand-2 arg ... subcommand-N ...
 *
 * Supported subcommands:
 *
 * GET <type> <offset>
 * SET <type> <offset> <value>
 * INCRBY <type> <offset> <increment>
 * OVERFLOW [WRAP|SAT|FAIL]
 */

#define BITFIELD_FLAG_NONE      0
#define BITFIELD_FLAG_READONLY (1<<0)

struct bitfieldOp {
    uint64_t offset; /* Bitfield offset. */
    int64_t i64; /* Increment amount (INCRBY) or SET value */
    int opcode; /* Operation id. */
    int owttype; /* Overflow type to use. */
    int bits; /* Integer bitfield bits width. */
    int sign; /* True if signed, otherwise unsigned op. */
};

/* This implements both the BITFIELD command and the BITFIELD_R0 command
 * when flags is set to BITFIELD_FLAG_READONLY: in this case only the
 * GET subcommand is allowed, other subcommands will return an error. */
void bitfieldGeneric(client *c, int flags) {
    robj *o;
    uint64_t bitoffset;
    int j, numops = 0, changes = 0, dirty = 0;
    struct bitfieldOp *ops = NULL; /* Array of ops to execute at end. */
    int owttype = BFOVERFLOW_WRAP; /* Overflow type. */
    int readonly = 1;

```

```

uint64_t highest_write_offset = 0;

for (j = 2; j < c->argc; j++) {
    int remargs = c->argc-j-1; /* Remaining args other than current. */
    char *subcmd = c->argv[j]->ptr; /* Current command name. */
    int opcode; /* Current operation code. */
    long long i64 = 0; /* Signed SET value. */
    int sign = 0; /* Signed or unsigned type? */
    int bits = 0; /* Bitfield width in bits. */

    if (!strcasecmp(subcmd,"get") && remargs >= 2)
        opcode = BITFIELDOP_GET;
    else if (!strcasecmp(subcmd,"set") && remargs >= 3)
        opcode = BITFIELDOP_SET;
    else if (!strcasecmp(subcmd,"incrby") && remargs >= 3)
        opcode = BITFIELDOP_INCRBY;
    else if (!strcasecmp(subcmd,"overflow") && remargs >= 1) {
        char *owtypename = c->argv[j+1]->ptr;
        j++;
        if (!strcasecmp(owtypename,"wrap"))
            owtype = BFOVERFLOW_WRAP;
        else if (!strcasecmp(owtypename,"sat"))
            owtype = BFOVERFLOW_SAT;
        else if (!strcasecmp(owtypename,"fail"))
            owtype = BFOVERFLOW_FAIL;
        else {
            addReplyError(c,"Invalid OVERFLOW type specified");
            zfree(ops);
            return;
        }
        continue;
    } else {
        addReplyErrorObject(c,shared.syntaxerr);
        zfree(ops);
        return;
    }

    /* Get the type and offset arguments, common to all the ops. */
    if (getBitfieldTypeFromArgument(c,c->argv[j+1],&sign,&bits) != C_OK) {
        zfree(ops);
        return;
    }

    if (getBitOffsetFromArgument(c,c->argv[j+2],&bitoffset,1,bits) != C_OK)
{
        zfree(ops);
        return;
    }

    if (opcode != BITFIELDOP_GET) {
        readonly = 0;
    }
}

```

```

        if (highest_write_offset < bitoffset + bits - 1)
            highest_write_offset = bitoffset + bits - 1;
        /* INCRBY and SET require another argument. */
        if (getLongLongFromObjectOrReply(c,c->argv[j+3],&i64,NULL) != C_OK)
    {
        zfree(ops);
        return;
    }

    /* Populate the array of operations we'll process. */
    ops = zrealloc(ops,sizeof(*ops)*(numops+1));
    ops[numops].offset = bitoffset;
    ops[numops].i64 = i64;
    ops[numops].opcode = opcode;
    ops[numops].owtype = owtype;
    ops[numops].bits = bits;
    ops[numops].sign = sign;
    numops++;

    j += 3 - (opcode == BITFIELDOP_GET);
}

if (readonly) {
    /* Lookup for read is ok if key doesn't exit, but errors
     * if it's not a string. */
    o = lookupKeyRead(c->db,c->argv[1]);
    if (o != NULL && checkType(c,o,OBJ_STRING)) {
        zfree(ops);
        return;
    }
} else {
    if (flags & BITFIELD_FLAG_READONLY) {
        zfree(ops);
        addReplyError(c, "BITFIELD_R0 only supports the GET subcommand");
        return;
    }

    /* Lookup by making room up to the farthest bit reached by
     * this operation. */
    if ((o = lookupStringForBitCommand(c,
        highest_write_offset,&dirty)) == NULL) {
        zfree(ops);
        return;
    }
}

addReplyArrayLen(c,numops);

/* Actually process the operations. */
for (j = 0; j < numops; j++) {

```



```

        if (thisop->opcode == BITFIELDOP_INCRBY) {
            newval = oldval + thisop->i64;
            overflow = checkUnsignedBitfieldOverflow(oldval,
                thisop->i64, thisop->bits, thisop->owtype, &wrapped);
            if (overflow) newval = wrapped;
            retval = newval;
        } else {
            newval = thisop->i64;
            overflow = checkUnsignedBitfieldOverflow(newval,
                0, thisop->bits, thisop->owtype, &wrapped);
            if (overflow) newval = wrapped;
            retval = oldval;
        }
        /* On overflow of type is "FAIL", don't write and return
         * NULL to signal the condition. */
        if (!(overflow && thisop->owtype == BFOVERFLOW_FAIL)) {
            addReplyLongLong(c, retval);
            setUnsignedBitfield(o->ptr, thisop->offset,
                thisop->bits, newval);

            if (dirty || (oldval != newval))
                changes++;
        } else {
            addReplyNull(c);
        }
    }
} else {
    /* GET */
    unsigned char buf[9];
    long strlen = 0;
    unsigned char *src = NULL;
    char llbuf[LONG_STR_SIZE];

    if (o != NULL)
        src = getObjectReadOnlyString(o, &strlen, llbuf);

    /* For GET we use a trick: before executing the operation
     * copy up to 9 bytes to a local buffer, so that we can easily
     * execute up to 64 bit operations that are at actual string
     * object boundaries. */
    memset(buf, 0, 9);
    int i;
    uint64_t byte = thisop->offset >> 3;
    for (i = 0; i < 9; i++) {
        if (src == NULL || i+byte >= (uint64_t)strlen) break;
        buf[i] = src[i+byte];
    }

    /* Now operate on the copied buffer which is guaranteed
     * to be zero-padded. */

```



```

        if (thisop->sign) {
            int64_t val = getSignedBitfield(buf, thisop->offset-(byte*8),
                                           thisop->bits);
            addReplyLongLong(c, val);
        } else {
            uint64_t val = getUnsignedBitfield(buf, thisop->offset-(byte*8),
                                               thisop->bits);
            addReplyLongLong(c, val);
        }
    }
}

if (changes) {
    signalModifiedKey(c, c->db, c->argv[1]);
    notifyKeyspaceEvent(NOTIFY_STRING, "setbit", c->argv[1], c->db->id);
    server.dirty += changes;
}
zfree(ops);
}

void bitfieldCommand(client *c) {
    bitfieldGeneric(c, BITFIELD_FLAG_NONE);
}

void bitfieldroCommand(client *c) {
    bitfieldGeneric(c, BITFIELD_FLAG_READONLY);
}

```

/blocked.c

[to top](#)

```

/* blocked.c - generic support for blocking operations like BLPOP & WAIT.
 *
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * -----
 *
 * API:
 *
 * blockClient() set the CLIENT_BLOCKED flag in the client, and set the
 * specified block type 'btype' filed to one of BLOCKED_* macros.
 *
 * unblockClient() unblocks the client doing the following:
 * 1) It calls the btype-specific function to cleanup the state.
 * 2) It unblocks the client by unsetting the CLIENT_BLOCKED flag.
 * 3) It puts the client into a list of just unblocked clients that are
 *    processed ASAP in the beforeSleep() event loop callback, so that
 *    if there is some query buffer to process, we do it. This is also
 *    required because otherwise there is no 'readable' event fired, we
 *    already read the pending commands. We also set the CLIENT_UNBLOCKED
 *    flag to remember the client is in the unblocked_clients list.
 *
 * processUnblockedClients() is called inside the beforeSleep() function
 * to process the query buffer from unblocked clients and remove the clients
 * from the blocked_clients queue.
 *

```

```

* replyToBlockedClientTimedOut() is called by the cron function when
* a client blocked reaches the specified timeout (if the timeout is set
* to 0, no timeout is processed).
* It usually just needs to send a reply to the client.
*
* When implementing a new type of blocking operation, the implementation
* should modify unblockClient() and replyToBlockedClientTimedOut() in order
* to handle the btype-specific behavior of this two functions.
* If the blocking operation waits for certain keys to change state, the
* clusterRedirectBlockedClientIfNeeded() function should also be updated.
*/

```

```

#include "server.h"
#include "slowlog.h"
#include "latency.h"
#include "monotonic.h"

```

```

void serveClientBlockedOnList(client *receiver, robj *o, robj *key, robj
*dstkey, redisDb *db, int wherefrom, int whereto, int *deleted);
int getListPositionFromObjectOrReply(client *c, robj *arg, int *position);

```

```

/* This structure represents the blocked key information that we store
* in the client structure. Each client blocked on keys, has a
* client->bpop.keys hash table. The keys of the hash table are Redis
* keys pointers to 'robj' structures. The value is this structure.
* The structure has two goals: firstly we store the list node that this
* client uses to be listed in the database "blocked clients for this key"
* list, so we can later unblock in O(1) without a list scan.
* Secondly for certain blocking types, we have additional info. Right now
* the only use for additional info we have is when clients are blocked
* on streams, as we have to remember the ID it blocked for. */

```

```

typedef struct bkinfo {
    listNode *listnode;    /* List node for db->blocking_keys[key] list. */
    streamID stream_id;    /* Stream ID if we blocked in a stream. */
} bkinfo;

```

```

/* Block a client for the specific operation type. Once the CLIENT_BLOCKED
* flag is set client query buffer is not longer processed, but accumulated,
* and will be processed when the client is unblocked. */

```

```

void blockClient(client *c, int btype) {
    /* Master client should never be blocked unless pause or module */
    serverAssert(!(c->flags & CLIENT_MASTER &&
        btype != BLOCKED_MODULE &&
        btype != BLOCKED_POSTPONE));

    c->flags |= CLIENT_BLOCKED;
    c->btype = btype;
    server.blocked_clients++;
    server.blocked_clients_by_type[btype]++;
    addClientToTimeoutTable(c);
    if (btype == BLOCKED_POSTPONE) {

```

```

        listAddNodeTail(server.postponed_clients, c);
        c->postponed_list_node = listLast(server.postponed_clients);
        /* Mark this client to execute its command */
        c->flags |= CLIENT_PENDING_COMMAND;
    }
}

/* This function is called after a client has finished a blocking operation
 * in order to update the total command duration, log the command into
 * the Slow log if needed, and log the reply duration event if needed. */
void updateStatsOnUnblock(client *c, long blocked_us, long reply_us, int
had_errors){
    const ustime_t total_cmd_duration = c->duration + blocked_us + reply_us;
    c->lastcmd->microseconds += total_cmd_duration;
    if (had_errors)
        c->lastcmd->failed_calls++;
    if (server.latency_tracking_enabled)
        updateCommandLatencyHistogram(&(c->lastcmd->latency_histogram),
total_cmd_duration*1000);
    /* Log the command into the Slow log if needed. */
    slowlogPushCurrentCommand(c, c->lastcmd, total_cmd_duration);
    /* Log the reply duration event. */
    latencyAddSampleIfNeeded("command-unblocking", reply_us/1000);
}

/* This function is called in the beforeSleep() function of the event loop
 * in order to process the pending input buffer of clients that were
 * unblocked after a blocking operation. */
void processUnblockedClients(void) {
    listNode *ln;
    client *c;

    while (listLength(server.unblocked_clients)) {
        ln = listFirst(server.unblocked_clients);
        serverAssert(ln != NULL);
        c = ln->value;
        listDelNode(server.unblocked_clients, ln);
        c->flags &= ~CLIENT_UNBLOCKED;

        /* Process remaining data in the input buffer, unless the client
         * is blocked again. Actually processInputBuffer() checks that the
         * client is not blocked before to proceed, but things may change and
         * the code is conceptually more correct this way. */
        if (!(c->flags & CLIENT_BLOCKED)) {
            /* If we have a queued command, execute it now. */
            if (processPendingCommandAndInputBuffer(c) == C_ERR) {
                c = NULL;
            }
        }
        beforeNextClient(c);
    }
}

```

```

}

/* This function will schedule the client for reprocessing at a safe time.
 *
 * This is useful when a client was blocked for some reason (blocking
operation,
 * CLIENT PAUSE, or whatever), because it may end with some accumulated query
 * buffer that needs to be processed ASAP:
 *
 * 1. When a client is blocked, its readable handler is still active.
 * 2. However in this case it only gets data into the query buffer, but the
 * query is not parsed or executed once there is enough to proceed as
 * usually (because the client is blocked... so we can't execute commands).
 * 3. When the client is unblocked, without this function, the client would
 * have to write some query in order for the readable handler to finally
 * call processQueryBuffer*() on it.
 * 4. With this function instead we can put the client in a queue that will
 * process it for queries ready to be executed at a safe time.
 */
void queueClientForReprocessing(client *c) {
    /* The client may already be into the unblocked list because of a previous
 * blocking operation, don't add back it into the list multiple times. */
    if (!(c->flags & CLIENT_UNBLOCKED)) {
        c->flags |= CLIENT_UNBLOCKED;
        listAddNodeTail(server.unblocked_clients,c);
    }
}

/* Unblock a client calling the right function depending on the kind
 * of operation the client is blocking for. */
void unblockClient(client *c) {
    if (c->btype == BLOCKED_LIST ||
        c->btype == BLOCKED_ZSET ||
        c->btype == BLOCKED_STREAM) {
        unblockClientWaitingData(c);
    } else if (c->btype == BLOCKED_WAIT) {
        unblockClientWaitingReplicas(c);
    } else if (c->btype == BLOCKED_MODULE) {
        if (moduleClientIsBlockedOnKeys(c)) unblockClientWaitingData(c);
        unblockClientFromModule(c);
    } else if (c->btype == BLOCKED_POSTPONE) {
        listDelNode(server.postponed_clients,c->postponed_list_node);
        c->postponed_list_node = NULL;
    } else if (c->btype == BLOCKED_SHUTDOWN) {
        /* No special cleanup. */
    } else {
        serverPanic("Unknown btype in unblockClient().");
    }

    /* Reset the client for a new query since, for blocking commands
 * we do not do it immediately after the command returns (when the

```

```

    * client got blocked) in order to be still able to access the argument
    * vector from module callbacks and updateStatsOnUnblock. */
if (c->btype != BLOCKED_POSTPONE && c->btype != BLOCKED_SHUTDOWN) {
    freeClientOriginalArgv(c);
    resetClient(c);
}

/* Clear the flags, and put the client in the unblocked list so that
 * we'll process new commands in its query buffer ASAP. */
server.blocked_clients--;
server.blocked_clients_by_type[c->btype]--;
c->flags &= ~CLIENT_BLOCKED;
c->btype = BLOCKED_NONE;
removeClientFromTimeoutTable(c);
queueClientForReprocessing(c);
}

/* This function gets called when a blocked client timed out in order to
 * send it a reply of some kind. After this function is called,
 * unblockClient() will be called with the same client as argument. */
void replyToBlockedClientTimedOut(client *c) {
    if (c->btype == BLOCKED_LIST ||
        c->btype == BLOCKED_ZSET ||
        c->btype == BLOCKED_STREAM) {
        addReplyNullArray(c);
    } else if (c->btype == BLOCKED_WAIT) {
        addReplyLongLong(c, replicationCountAcksByOffset(c->bpop.reploffset));
    } else if (c->btype == BLOCKED_MODULE) {
        moduleBlockedClientTimedOut(c);
    } else {
        serverPanic("Unknown btype in replyToBlockedClientTimedOut().");
    }
}

/* If one or more clients are blocked on the SHUTDOWN command, this function
 * sends them an error reply and unblocks them. */
void replyToClientsBlockedOnShutdown(void) {
    if (server.blocked_clients_by_type[BLOCKED_SHUTDOWN] == 0) return;
    listNode *ln;
    listIter li;
    listRewind(server.clients, &li);
    while((ln = listNext(&li))) {
        client *c = listNodeValue(ln);
        if (c->flags & CLIENT_BLOCKED && c->btype == BLOCKED_SHUTDOWN) {
            addReplyError(c, "Errors trying to SHUTDOWN. Check logs.");
            unblockClient(c);
        }
    }
}

/* Mass-unblock clients because something changed in the instance that makes

```

```
* blocking no longer safe. For example clients blocked in list operations
* in an instance which turns from master to slave is unsafe, so this function
* is called when a master turns into a slave.
```

```
*
```

```
* The semantics is to send an -UNBLOCKED error to the client, disconnecting
* it at the same time. */
```

```
void disconnectAllBlockedClients(void) {
```

```
    listNode *ln;
```

```
    listIter li;
```

```
    listRewind(server.clients,&li);
```

```
    while((ln = listNext(&li))) {
```

```
        client *c = listNodeValue(ln);
```

```
        if (c->flags & CLIENT_BLOCKED) {
```

```
            /* POSTPONED clients are an exception, when they'll be unblocked,
```

```
the
```

```
            * command processing will start from scratch, and the command will
            * be either executed or rejected. (unlike LIST blocked clients for
            * which the command is already in progress in a way. */
```

```
            if (c->btype == BLOCKED_POSTPONE)
```

```
                continue;
```

```
            addReplyError(c,
```

```
                "-UNBLOCKED force unblock from blocking operation, "
```

```
                "instance state changed (master -> replica?)");
```

```
            unblockClient(c);
```

```
            c->flags |= CLIENT_CLOSE_AFTER_REPLY;
```

```
        }
```

```
    }
```

```
}
```

```
/* Helper function for handleClientsBlockedOnKeys(). This function is called
* when there may be clients blocked on a list key, and there may be new
* data to fetch (the key is ready). */
```

```
void serveClientsBlockedOnListKey(robj *o, readyList *rl) {
```

```
    /* Optimization: If no clients are in type BLOCKED_LIST,
```

```
    * we can skip this loop. */
```

```
    if (!server.blocked_clients_by_type[BLOCKED_LIST]) return;
```

```
    /* We serve clients in the same order they blocked for
```

```
    * this key, from the first blocked to the last. */
```

```
    dictEntry *de = dictFind(rl->db->blocking_keys,rl->key);
```

```
    if (de) {
```

```
        list *clients = dictGetVal(de);
```

```
        listNode *ln;
```

```
        listIter li;
```

```
        listRewind(clients,&li);
```

```
        while((ln = listNext(&li))) {
```

```
            client *receiver = listNodeValue(ln);
```

```

        if (receiver->btype != BLOCKED_LIST) continue;

        int deleted = 0;
        robj *dstkey = receiver->bpop.target;
        int wherefrom = receiver->bpop.blockpos.wherefrom;
        int whereto = receiver->bpop.blockpos.whereto;

        /* Protect receiver->bpop.target, that will be
         * freed by the next unblockClient()
         * call. */
        if (dstkey) incrRefCount(dstkey);

        long long prev_error_replies = server.stat_total_error_replies;
        client *old_client = server.current_client;
        server.current_client = receiver;
        monotime replyTimer;
        elapsedStart(&replyTimer);
        serveClientBlockedOnList(receiver, o,
                                rl->key, dstkey, rl->db,
                                wherefrom, whereto,
                                &deleted);
        updateStatsOnUnblock(receiver, 0, elapsedUs(replyTimer),
server.stat_total_error_replies != prev_error_replies);
        unblockClient(receiver);
        afterCommand(receiver);
        server.current_client = old_client;

        if (dstkey) decrRefCount(dstkey);

        /* The list is empty and has been deleted. */
        if (deleted) break;
    }
}

/* Helper function for handleClientsBlockedOnKeys(). This function is called
 * when there may be clients blocked on a sorted set key, and there may be new
 * data to fetch (the key is ready). */
void serveClientsBlockedOnSortedSetKey(robj *o, readyList *rl) {
    /* Optimization: If no clients are in type BLOCKED_ZSET,
     * we can skip this loop. */
    if (!server.blocked_clients_by_type[BLOCKED_ZSET]) return;

    /* We serve clients in the same order they blocked for
     * this key, from the first blocked to the last. */
    dictEntry *de = dictFind(rl->db->blocking_keys,rl->key);
    if (de) {
        list *clients = dictGetVal(de);
        listNode *ln;
        listIter li;
        listRewind(clients,&li);

```



```

while((ln = listNext(&li))) {
    client *receiver = listNodeValue(ln);
    if (receiver->btype != BLOCKED_ZSET) continue;

    int deleted = 0;
    long llen = zsetLength(o);
    long count = receiver->bpop.count;
    int where = receiver->bpop.blockpos.wherefrom;
    int use_nested_array = (receiver->lastcmd &&
                           receiver->lastcmd->proc == bzmpopCommand)
                           ? 1 : 0;
    int reply_nil_when_empty = use_nested_array;

    long long prev_error_replies = server.stat_total_error_replies;
    client *old_client = server.current_client;
    server.current_client = receiver;
    monotime replyTimer;
    elapsedStart(&replyTimer);
    genericZpopCommand(receiver, &rl->key, 1, where, 1, count,
use_nested_array, reply_nil_when_empty, &deleted);

    /* Replicate the command. */
    int argc = 2;
    robj *argv[3];
    argv[0] = where == ZSET_MIN ? shared.zpopmin : shared.zpopmax;
    argv[1] = rl->key;
    incrRefCount(rl->key);
    if (count != -1) {
        /* Replicate it as command with COUNT. */
        robj *count_obj = createStringObjectFromLongLong((count > llen)
? llen : count);
        argv[2] = count_obj;
        argc++;
    }
    alsoPropagate(receiver->db->id, argv, argc,
PROPAGATE_AOF|PROPAGATE_REPL);
    decrRefCount(argv[1]);
    if (count != -1) decrRefCount(argv[2]);

    updateStatsOnUnblock(receiver, 0, elapsedUs(replyTimer),
server.stat_total_error_replies != prev_error_replies);
    unblockClient(receiver);
    afterCommand(receiver);
    server.current_client = old_client;

    /* The zset is empty and has been deleted. */
    if (deleted) break;
}
}
}

```

```

/* Helper function for handleClientsBlockedOnKeys(). This function is called
 * when there may be clients blocked on a stream key, and there may be new
 * data to fetch (the key is ready). */
void serveClientsBlockedOnStreamKey(roboj *o, readyList *rl) {
    /* Optimization: If no clients are in type BLOCKED_STREAM,
     * we can skip this loop. */
    if (!server.blocked_clients_by_type[BLOCKED_STREAM]) return;

    dictEntry *de = dictFind(rl->db->blocking_keys,rl->key);
    stream *s = o->ptr;

    /* We need to provide the new data arrived on the stream
     * to all the clients that are waiting for an offset smaller
     * than the current top item. */
    if (de) {
        list *clients = dictGetVal(de);
        listNode *ln;
        listIter li;
        listRewind(clients,&li);

        while((ln = listNext(&li))) {
            client *receiver = listNodeValue(ln);
            if (receiver->btype != BLOCKED_STREAM) continue;
            bkiinfo *bki = dictFetchValue(receiver->bpop.keys,rl->key);
            streamID *gt = &bki->stream_id;

            long long prev_error_replies = server.stat_total_error_replies;
            client *old_client = server.current_client;
            server.current_client = receiver;
            monotime replyTimer;
            elapsedStart(&replyTimer);

            /* If we blocked in the context of a consumer
             * group, we need to resolve the group and update the
             * last ID the client is blocked for: this is needed
             * because serving other clients in the same consumer
             * group will alter the "last ID" of the consumer
             * group, and clients blocked in a consumer group are
             * always blocked for the ">" ID: we need to deliver
             * only new messages and avoid unblocking the client
             * otherwise. */
            streamCG *group = NULL;
            if (receiver->bpop.xread_group) {
                group = streamLookupCG(s,
                    receiver->bpop.xread_group->ptr);
                /* If the group was not found, send an error
                 * to the consumer. */
                if (!group) {
                    addReplyError(receiver,
                        "-NOGROUP the consumer group this client ")

```

```

        "was blocked on no longer exists");
        goto unblock_receiver;
    } else {
        *gt = group->last_id;
    }
}

if (streamCompareID(&s->last_id, gt) > 0) {
    streamID start = *gt;
    streamIncrID(&start);

    /* Lookup the consumer for the group, if any. */
    streamConsumer *consumer = NULL;
    int noack = 0;

    if (group) {
        noack = receiver->bpop.xread_group_noack;
        sds name = receiver->bpop.xread_consumer->ptr;
        consumer = streamLookupConsumer(group, name, SLC_DEFAULT);
        if (consumer == NULL) {
            consumer = streamCreateConsumer(group, name, rl->key,
                                             rl->db-
>id, SCC_DEFAULT);

            if (noack) {
                streamPropagateConsumerCreation(receiver, rl->key,
>bpop.xread_group,
                                             consumer->name);
            }
        }
    }

    /* Emit the two elements sub-array consisting of
     * the name of the stream and the data we
     * extracted from it. Wrapped in a single-item
     * array, since we have just one key. */
    if (receiver->resp == 2) {
        addReplyArrayLen(receiver, 1);
        addReplyArrayLen(receiver, 2);
    } else {
        addReplyMapLen(receiver, 1);
    }
    addReplyBulk(receiver, rl->key);

    streamPropInfo pi = {
        rl->key,
        receiver->bpop.xread_group
    };
    streamReplyWithRange(receiver, s, &start, NULL,
        receiver->bpop.xread_count,
        0, group, consumer, noack, &pi);
}

```

```

        /* Note that after we unblock the client, 'gt'
        * and other receiver->bpop stuff are no longer
        * valid, so we must do the setup above before
        * the unblockClient call. */

unblock_receiver:
    updateStatsOnUnblock(receiver, 0, elapsedUs(replyTimer),
server.stat_total_error_replies != prev_error_replies);
    unblockClient(receiver);
    afterCommand(receiver);
    server.current_client = old_client;
    }
    }
}

/* Helper function for handleClientsBlockedOnKeys(). This function is called
 * in order to check if we can serve clients blocked by modules using
 * RM_BlockClientOnKeys(), when the corresponding key was signaled as ready:
 * our goal here is to call the RedisModuleBlockedClient reply() callback to
 * see if the key is really able to serve the client, and in that case,
 * unblock it. */
void serveClientsBlockedOnKeyByModule(readyList *rl) {
    /* Optimization: If no clients are in type BLOCKED_MODULE,
    * we can skip this loop. */
    if (!server.blocked_clients_by_type[BLOCKED_MODULE]) return;

    /* We serve clients in the same order they blocked for
    * this key, from the first blocked to the last. */
    dictEntry *de = dictFind(rl->db->blocking_keys,rl->key);
    if (de) {
        list *clients = dictGetVal(de);
        listNode *ln;
        listIter li;
        listRewind(clients,&li);

        while((ln = listNext(&li))) {
            client *receiver = listNodeValue(ln);
            if (receiver->btype != BLOCKED_MODULE) continue;

            /* Note that if *this* client cannot be served by this key,
            * it does not mean that another client that is next into the
            * list cannot be served as well: they may be blocked by
            * different modules with different triggers to consider if a key
            * is ready or not. This means we can't exit the loop but need
            * to continue after the first failure. */
            long long prev_error_replies = server.stat_total_error_replies;
            client *old_client = server.current_client;
            server.current_client = receiver;
            monotime replyTimer;
            elapsedStart(&replyTimer);

```

```

        if (!moduleTryServeClientBlockedOnKey(receiver, rl->key)) continue;
        updateStatsOnUnblock(receiver, 0, elapsedUs(replyTimer),
server.stat_total_error_replies != prev_error_replies);
        moduleUnblockClient(receiver);
        afterCommand(receiver);
        server.current_client = old_client;
    }
}

/* Helper function for handleClientsBlockedOnKeys(). This function is called
 * when there may be clients blocked, via XREADGROUP, on an existing stream
which
 * was deleted. We need to unblock the clients in that case.
 * The idea is that a client that is blocked via XREADGROUP is different from
 * any other blocking type in the sense that it depends on the existence of
both
 * the key and the group. Even if the key is deleted and then revived with XADD
 * it won't help any clients blocked on XREADGROUP because the group no longer
 * exist, so they would fail with -NOGROUP anyway.
 * The conclusion is that it's better to unblock these client (with error) upon
 * the deletion of the key, rather than waiting for the first XADD. */
void unblockDeletedStreamReadgroupClients(readyList *rl) {
    /* Optimization: If no clients are in type BLOCKED_STREAM,
     * we can skip this loop. */
    if (!server.blocked_clients_by_type[BLOCKED_STREAM]) return;

    /* We serve clients in the same order they blocked for
     * this key, from the first blocked to the last. */
    dictEntry *de = dictFind(rl->db->blocking_keys,rl->key);
    if (de) {
        list *clients = dictGetVal(de);
        listNode *ln;
        listIter li;
        listRewind(clients,&li);

        while((ln = listNext(&li))) {
            client *receiver = listNodeValue(ln);
            if (receiver->btype != BLOCKED_STREAM || !receiver->
>bpop.xread_group)
                continue;

            long long prev_error_replies = server.stat_total_error_replies;
            client *old_client = server.current_client;
            server.current_client = receiver;
            monotime replyTimer;
            elapsedStart(&replyTimer);
            addReplyError(receiver, "-UNBLOCKED the stream key no longer
exists");
            updateStatsOnUnblock(receiver, 0, elapsedUs(replyTimer),
server.stat_total_error_replies != prev_error_replies);

```

```

        unblockClient(receiver);
        afterCommand(receiver);
        server.current_client = old_client;
    }
}

/* This function should be called by Redis every time a single command,
 * a MULTI/EXEC block, or a Lua script, terminated its execution after
 * being called by a client. It handles serving clients blocked in
 * lists, streams, and sorted sets, via a blocking commands.
 *
 * All the keys with at least one client blocked that received at least
 * one new element via some write operation are accumulated into
 * the server.ready_keys list. This function will run the list and will
 * serve clients accordingly. Note that the function will iterate again and
 * again as a result of serving BLMOVE we can have new blocking clients
 * to serve because of the PUSH side of BLMOVE.
 *
 * This function is normally "fair", that is, it will server clients
 * using a FIFO behavior. However this fairness is violated in certain
 * edge cases, that is, when we have clients blocked at the same time
 * in a sorted set and in a list, for the same key (a very odd thing to
 * do client side, indeed!). Because mismatching clients (blocking for
 * a different type compared to the current key type) are moved in the
 * other side of the linked list. However as long as the key starts to
 * be used only for a single type, like virtually any Redis application will
 * do, the function is already fair. */
void handleClientsBlockedOnKeys(void) {
    /* This function is called only when also_propagate is in its basic state
     * (i.e. not from call(), module context, etc.) */
    serverAssert(server.also_propagate.numops == 0);
    server.core_propagates = 1;

    while(listLength(server.ready_keys) != 0) {
        list *l;

        /* Point server.ready_keys to a fresh list and save the current one
         * locally. This way as we run the old list we are free to call
         * signalKeyAsReady() that may push new elements in server.ready_keys
         * when handling clients blocked into BLMOVE. */
        l = server.ready_keys;
        server.ready_keys = listCreate();

        while(listLength(l) != 0) {
            listNode *ln = listFirst(l);
            readyList *rl = ln->value;

            /* First of all remove this key from db->ready_keys so that
             * we can safely call signalKeyAsReady() against this key. */
            dictDelete(rl->db->ready_keys,rl->key);

```

```

/* Even if we are not inside call(), increment the call depth
 * in order to make sure that keys are expired against a fixed
 * reference time, and not against the wallclock time. This
 * way we can lookup an object multiple times (BLMOVE does
 * that) without the risk of it being freed in the second
 * lookup, invalidating the first one.
 * See https://github.com/redis/redis/pull/6554. */
server.fixed_time_expire++;
updateCachedTime(0);

/* Serve clients blocked on the key. */
robject *o = lookupKeyReadWithFlags(rl->db, rl->key, LOOKUP_NONOTIFY |
LOOKUP_NOSTATS);
if (o != NULL) {
    int objtype = o->type;
    if (objtype == OBJ_LIST)
        serveClientsBlockedOnListKey(o,rl);
    else if (objtype == OBJ_ZSET)
        serveClientsBlockedOnSortedSetKey(o,rl);
    else if (objtype == OBJ_STREAM)
        serveClientsBlockedOnStreamKey(o,rl);
    /* We want to serve clients blocked on module keys
     * regardless of the object type: we don't know what the
     * module is trying to accomplish right now. */
    serveClientsBlockedOnKeyByModule(rl);
    /* If we have XREADGROUP clients blocked on this key, and
     * the key is not a stream, it must mean that the key was
     * overwritten by either SET or something like
     * (MULTI, DEL key, SADD key e, EXEC).
     * In this case we need to unblock all these clients. */
    if (objtype != OBJ_STREAM)
        unblockDeletedStreamReadgroupClients(rl);
} else {
    /* Unblock all XREADGROUP clients of this deleted key */
    unblockDeletedStreamReadgroupClients(rl);
    /* Edge case: If lookupKeyReadWithFlags decides to expire the
key we have to
     * take care of the propagation here, because afterCommand
wasn't called */
    if (server.also_propagate.numops > 0)
        propagatePendingCommands();
}
server.fixed_time_expire--;

/* Free this item. */
decrRefCount(rl->key);
zfree(rl);
listDelNode(l,ln);
}
listRelease(l); /* We have the new list on place at this point. */

```

```

    }

    serverAssert(server.core_propagates); /* This function should not be re-
entrant */

    server.core_propagates = 0;
}

/* This is how the current blocking lists/sorted sets/streams work, we use
 * BLPPOP as example, but the concept is the same for other list ops, sorted
 * sets and XREAD.
 * - If the user calls BLPPOP and the key exists and contains a non empty list
 *   then LPOP is called instead. So BLPPOP is semantically the same as LPOP
 *   if blocking is not required.
 * - If instead BLPPOP is called and the key does not exists or the list is
 *   empty we need to block. In order to do so we remove the notification for
 *   new data to read in the client socket (so that we'll not serve new
 *   requests if the blocking request is not served). Also we put the client
 *   in a dictionary (db->blocking_keys) mapping keys to a list of clients
 *   blocking for this keys.
 * - If a PUSH operation against a key with blocked clients waiting is
 *   performed, we mark this key as "ready", and after the current command,
 *   MULTI/EXEC block, or script, is executed, we serve all the clients waiting
 *   for this list, from the one that blocked first, to the last, accordingly
 *   to the number of elements we have in the ready list.
 */

/* Set a client in blocking mode for the specified key (list, zset or stream),
 * with the specified timeout. The 'type' argument is BLOCKED_LIST,
 * BLOCKED_ZSET or BLOCKED_STREAM depending on the kind of operation we are
 * waiting for an empty key in order to awake the client. The client is blocked
 * for all the 'numkeys' keys as in the 'keys' argument. When we block for
 * stream keys, we also provide an array of streamID structures: clients will
 * be unblocked only when items with an ID greater or equal to the specified
 * one is appended to the stream.
 *
 * 'count' for those commands that support the optional count argument.
 * Otherwise the value is 0. */
void blockForKeys(client *c, int btype, robj **keys, int numkeys, long count,
mstime_t timeout, robj *target, struct blockPos *blockpos, streamID *ids) {
    dictEntry *de;
    list *l;
    int j;

    c->bpop.count = count;
    c->bpop.timeout = timeout;
    c->bpop.target = target;

    if (blockpos != NULL) c->bpop.blockpos = *blockpos;

    if (target != NULL) incrRefCount(target);

```



```

for (j = 0; j < numkeys; j++) {
    /* Allocate our bkinfo structure, associated to each key the client
     * is blocked for. */
    bkinfo *bki = zmalloc(sizeof(*bki));
    if (btype == BLOCKED_STREAM)
        bki->stream_id = ids[j];

    /* If the key already exists in the dictionary ignore it. */
    if (dictAdd(c->bpop.keys,keys[j],bki) != DICT_OK) {
        zfree(bki);
        continue;
    }
    incrRefCount(keys[j]);

    /* And in the other "side", to map keys -> clients */
    de = dictFind(c->db->blocking_keys,keys[j]);
    if (de == NULL) {
        int retval;

        /* For every key we take a list of clients blocked for it */
        l = listCreate();
        retval = dictAdd(c->db->blocking_keys,keys[j],l);
        incrRefCount(keys[j]);
        serverAssertWithInfo(c,keys[j],retval == DICT_OK);
    } else {
        l = dictGetVal(de);
    }
    listAddNodeTail(l,c);
    bki->listnode = listLast(l);
}
blockClient(c,btype);
}

/* Unblock a client that's waiting in a blocking operation such as BLP0P.
 * You should never call this function directly, but unblockClient() instead.
 */
void unblockClientWaitingData(client *c) {
    dictEntry *de;
    dictIterator *di;
    list *l;

    serverAssertWithInfo(c,NULL,dictSize(c->bpop.keys) != 0);
    di = dictGetIterator(c->bpop.keys);
    /* The client may wait for multiple keys, so unblock it for every key. */
    while((de = dictNext(di)) != NULL) {
        robj *key = dictGetKey(de);
        bkinfo *bki = dictGetVal(de);

        /* Remove this client from the list of clients waiting for this key. */
        l = dictFetchValue(c->db->blocking_keys,key);

```

```

        serverAssertWithInfo(c,key,l != NULL);
        listDelNode(l,bki->listnode);
        /* If the list is empty we need to remove it to avoid wasting memory */
        if (listLength(l) == 0)
            dictDelete(c->db->blocking_keys,key);
    }
    dictReleaseIterator(di);

    /* Cleanup the client structure */
    dictEmpty(c->bpop.keys,NULL);
    if (c->bpop.target) {
        decrRefCount(c->bpop.target);
        c->bpop.target = NULL;
    }
    if (c->bpop.xread_group) {
        decrRefCount(c->bpop.xread_group);
        decrRefCount(c->bpop.xread_consumer);
        c->bpop.xread_group = NULL;
        c->bpop.xread_consumer = NULL;
    }
}

static int getBlockedTypeByType(int type) {
    switch (type) {
        case OBJ_LIST: return BLOCKED_LIST;
        case OBJ_ZSET: return BLOCKED_ZSET;
        case OBJ_MODULE: return BLOCKED_MODULE;
        case OBJ_STREAM: return BLOCKED_STREAM;
        default: return BLOCKED_NONE;
    }
}

/* If the specified key has clients blocked waiting for list pushes, this
 * function will put the key reference into the server.ready_keys list.
 * Note that db->ready_keys is a hash table that allows us to avoid putting
 * the same key again and again in the list in case of multiple pushes
 * made by a script or in the context of MULTI/EXEC.
 *
 * The list will be finally processed by handleClientsBlockedOnKeys() */
void signalKeyAsReady(redisDb *db, robj *key, int type) {
    readyList *rl;

    /* Quick returns. */
    int btype = getBlockedTypeByType(type);
    if (btype == BLOCKED_NONE) {
        /* The type can never block. */
        return;
    }
    if (!server.blocked_clients_by_type[btype] &&
        !server.blocked_clients_by_type[BLOCKED_MODULE]) {
        /* No clients block on this type. Note: Blocked modules are represented

```

```

        * by BLOCKED_MODULE, even if the intention is to wake up by normal
        * types (list, zset, stream), so we need to check that there are no
        * blocked modules before we do a quick return here. */
    return;
}

/* No clients blocking for this key? No need to queue it. */
if (dictFind(db->blocking_keys,key) == NULL) return;

/* Key was already signaled? No need to queue it again. */
if (dictFind(db->ready_keys,key) != NULL) return;

/* Ok, we need to queue this key into server.ready_keys. */
rl = zmalloc(sizeof(*rl));
rl->key = key;
rl->db = db;
incrRefCount(key);
listAddNodeTail(server.ready_keys,rl);

/* We also add the key in the db->ready_keys dictionary in order
 * to avoid adding it multiple times into a list with a simple O(1)
 * check. */
incrRefCount(key);
serverAssert(dictAdd(db->ready_keys,key,NULL) == DICT_OK);
}

```

[/call_reply.c](#)

[to top](#)

```

/*
 * Copyright (c) 2009–2021, Redis Labs Ltd.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "server.h"
#include "call_reply.h"

#define REPLY_FLAG_ROOT (1<<0)
#define REPLY_FLAG_PARSED (1<<1)
#define REPLY_FLAG_RESP3 (1<<2)

/* -----
 * An opaque struct used to parse a RESP protocol reply and
 * represent it. Used when parsing replies such as in RM_Call
 * or Lua scripts.
 * ----- */
struct CallReply {
    void *private_data;
    sds original_proto; /* Available only for root reply. */
    const char *proto;
    size_t proto_len;
    int type;           /* REPLY_... */
    int flags;          /* REPLY_FLAG... */
    size_t len;         /* Length of a string, or the number elements in an array.
 */

```

```

union {
    const char *str; /* String pointer for string and error replies. This
                     * does not need to be freed, always points inside
                     * a reply->proto buffer of the reply object or, in
                     * case of array elements, of parent reply objects. */

    struct {
        const char *str;
        const char *format;
    } verbatim_str; /* Reply value for verbatim string */
    long long ll;    /* Reply value for integer reply. */
    double d;        /* Reply value for double reply. */
    struct CallReply *array; /* Array of sub-reply elements. used for set,
                             array, map, and attribute */
} val;
list *deferred_error_list; /* list of errors in sds form or NULL */
struct CallReply *attribute; /* attribute reply, NULL if not exists */
};

static void callReplySetSharedData(CallReply *rep, int type, const char *proto,
size_t proto_len, int extra_flags) {
    rep->type = type;
    rep->proto = proto;
    rep->proto_len = proto_len;
    rep->flags |= extra_flags;
}

static void callReplyNull(void *ctx, const char *proto, size_t proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_NULL, proto, proto_len,
REPLY_FLAG_RESP3);
}

static void callReplyNullBulkString(void *ctx, const char *proto, size_t
proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_NULL, proto, proto_len, 0);
}

static void callReplyNullArray(void *ctx, const char *proto, size_t proto_len)
{
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_NULL, proto, proto_len, 0);
}

static void callReplyBulkString(void *ctx, const char *str, size_t len, const
char *proto, size_t proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_STRING, proto, proto_len, 0);
    rep->len = len;
    rep->val.str = str;
}

```

```

static void callReplyError(void *ctx, const char *str, size_t len, const char
*proto, size_t proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_ERROR, proto, proto_len, 0);
    rep->len = len;
    rep->val.str = str;
}

static void callReplySimpleStr(void *ctx, const char *str, size_t len, const
char *proto, size_t proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_STRING, proto, proto_len, 0);
    rep->len = len;
    rep->val.str = str;
}

static void callReplyLong(void *ctx, long long val, const char *proto, size_t
proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_INTEGER, proto, proto_len,
0);
    rep->val.ll = val;
}

static void callReplyDouble(void *ctx, double val, const char *proto, size_t
proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_DOUBLE, proto, proto_len,
REPLY_FLAG_RESP3);
    rep->val.d = val;
}

static void callReplyVerbatimString(void *ctx, const char *format, const char
*str, size_t len, const char *proto, size_t proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_VERBATIM_STRING, proto,
proto_len, REPLY_FLAG_RESP3);
    rep->len = len;
    rep->val.verbatim_str.str = str;
    rep->val.verbatim_str.format = format;
}

static void callReplyBigNumber(void *ctx, const char *str, size_t len, const
char *proto, size_t proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_BIG_NUMBER, proto, proto_len,
REPLY_FLAG_RESP3);
    rep->len = len;
    rep->val.str = str;
}

```

```

static void callReplyBool(void *ctx, int val, const char *proto, size_t
proto_len) {
    CallReply *rep = ctx;
    callReplySetSharedData(rep, REDISMODULE_REPLY_BOOL, proto, proto_len,
REPLY_FLAG_RESP3);
    rep->val.ll = val;
}

static void callReplyParseCollection(ReplyParser *parser, CallReply *rep,
size_t len, const char *proto, size_t elements_per_entry) {
    rep->len = len;
    rep->val.array = zcalloc(elements_per_entry * len * sizeof(CallReply));
    for (size_t i = 0; i < len * elements_per_entry; i += elements_per_entry) {
        for (size_t j = 0; j < elements_per_entry; ++j) {
            rep->val.array[i + j].private_data = rep->private_data;
            parseReply(parser, rep->val.array + i + j);
            rep->val.array[i + j].flags |= REPLY_FLAG_PARSED;
            if (rep->val.array[i + j].flags & REPLY_FLAG_RESP3) {
                /* If one of the sub-replies is RESP3, then the current reply
is also RESP3. */
                rep->flags |= REPLY_FLAG_RESP3;
            }
        }
    }
    rep->proto = proto;
    rep->proto_len = parser->curr_location - proto;
}

static void callReplyAttribute(ReplyParser *parser, void *ctx, size_t len,
const char *proto) {
    CallReply *rep = ctx;
    rep->attribute = zcalloc(sizeof(CallReply));

    /* Continue parsing the attribute reply */
    rep->attribute->len = len;
    rep->attribute->type = REDISMODULE_REPLY_ATTRIBUTE;
    callReplyParseCollection(parser, rep->attribute, len, proto, 2);
    rep->attribute->flags |= REPLY_FLAG_PARSED | REPLY_FLAG_RESP3;
    rep->attribute->private_data = rep->private_data;

    /* Continue parsing the reply */
    parseReply(parser, rep);

    /* In this case we need to fix the proto address and len, it should start
from the attribute */
    rep->proto = proto;
    rep->proto_len = parser->curr_location - proto;
    rep->flags |= REPLY_FLAG_RESP3;
}

```

```

static void callReplyArray(ReplyParser *parser, void *ctx, size_t len, const
char *proto) {
    CallReply *rep = ctx;
    rep->type = REDISMODULE_REPLY_ARRAY;
    callReplyParseCollection(parser, rep, len, proto, 1);
}

static void callReplySet(ReplyParser *parser, void *ctx, size_t len, const char
*proto) {
    CallReply *rep = ctx;
    rep->type = REDISMODULE_REPLY_SET;
    callReplyParseCollection(parser, rep, len, proto, 1);
    rep->flags |= REPLY_FLAG_RESP3;
}

static void callReplyMap(ReplyParser *parser, void *ctx, size_t len, const char
*proto) {
    CallReply *rep = ctx;
    rep->type = REDISMODULE_REPLY_MAP;
    callReplyParseCollection(parser, rep, len, proto, 2);
    rep->flags |= REPLY_FLAG_RESP3;
}

static void callReplyParseError(void *ctx) {
    CallReply *rep = ctx;
    rep->type = REDISMODULE_REPLY_UNKNOWN;
}

/* Recursively free the current call reply and its sub-replies. */
static void freeCallReplyInternal(CallReply *rep) {
    if (rep->type == REDISMODULE_REPLY_ARRAY || rep->type ==
REDISMODULE_REPLY_SET) {
        for (size_t i = 0 ; i < rep->len ; ++i) {
            freeCallReplyInternal(rep->val.array + i);
        }
        zfree(rep->val.array);
    }

    if (rep->type == REDISMODULE_REPLY_MAP || rep->type ==
REDISMODULE_REPLY_ATTRIBUTE) {
        for (size_t i = 0 ; i < rep->len ; ++i) {
            freeCallReplyInternal(rep->val.array + i * 2);
            freeCallReplyInternal(rep->val.array + i * 2 + 1);
        }
        zfree(rep->val.array);
    }

    if (rep->attribute) {
        freeCallReplyInternal(rep->attribute);
        zfree(rep->attribute);
    }
}

```



```

}

/* Free the given call reply and its children (in case of nested reply)
recursively.
 * If private data was set when the CallReply was created it will not be freed,
as it's
 * the caller's responsibility to free it before calling freeCallReply(). */
void freeCallReply(CallReply *rep) {
    if (!(rep->flags & REPLY_FLAG_ROOT)) {
        return;
    }
    if (rep->flags & REPLY_FLAG_PARSED) {
        freeCallReplyInternal(rep);
    }
    sdsfree(rep->original_proto);
    if (rep->deferred_error_list)
        listRelease(rep->deferred_error_list);
    zfree(rep);
}

static const ReplyParserCallbacks DefaultParserCallbacks = {
    .null_callback = callReplyNull,
    .bulk_string_callback = callReplyBulkString,
    .null_bulk_string_callback = callReplyNullBulkString,
    .null_array_callback = callReplyNullArray,
    .error_callback = callReplyError,
    .simple_str_callback = callReplySimpleStr,
    .long_callback = callReplyLong,
    .array_callback = callReplyArray,
    .set_callback = callReplySet,
    .map_callback = callReplyMap,
    .double_callback = callReplyDouble,
    .bool_callback = callReplyBool,
    .big_number_callback = callReplyBigNumber,
    .verbatim_string_callback = callReplyVerbatimString,
    .attribute_callback = callReplyAttribute,
    .error = callReplyParseError,
};

/* Parse the buffer located in rep->original_proto and update the CallReply
 * structure to represent its contents. */
static void callReplyParse(CallReply *rep) {
    if (rep->flags & REPLY_FLAG_PARSED) {
        return;
    }

    ReplyParser parser = {.curr_location = rep->proto, .callbacks =
DefaultParserCallbacks};

    parseReply(&parser, rep);
    rep->flags |= REPLY_FLAG_PARSED;
}

```

```

}

/* Return the call reply type (REDISMODULE_REPLY_...). */
int callReplyType(CallReply *rep) {
    if (!rep) return REDISMODULE_REPLY_UNKNOWN;
    callReplyParse(rep);
    return rep->type;
}

/* Return reply string as buffer and len. Applicable to:
 * - REDISMODULE_REPLY_STRING
 * - REDISMODULE_REPLY_ERROR
 *
 * The return value is borrowed from CallReply, so it must not be freed
 * explicitly or used after CallReply itself is freed.
 *
 * The returned value is not NULL terminated and its length is returned by
 * reference through len, which must not be NULL.
 */
const char *callReplyGetString(CallReply *rep, size_t *len) {
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_STRING &&
        rep->type != REDISMODULE_REPLY_ERROR) return NULL;
    if (len) *len = rep->len;
    return rep->val.str;
}

/* Return a long long reply value. Applicable to:
 * - REDISMODULE_REPLY_INTEGER
 */
long long callReplyGetLongLong(CallReply *rep) {
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_INTEGER) return LLONG_MIN;
    return rep->val.ll;
}

/* Return a double reply value. Applicable to:
 * - REDISMODULE_REPLY_DOUBLE
 */
double callReplyGetDouble(CallReply *rep) {
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_DOUBLE) return LLONG_MIN;
    return rep->val.d;
}

/* Return a reply Boolean value. Applicable to:
 * - REDISMODULE_REPLY_BOOL
 */
int callReplyGetBool(CallReply *rep) {
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_BOOL) return INT_MIN;

```

```

        return rep->val.ll;
    }

/* Return reply length. Applicable to:
 * - REDISMODULE_REPLY_STRING
 * - REDISMODULE_REPLY_ERROR
 * - REDISMODULE_REPLY_ARRAY
 * - REDISMODULE_REPLY_SET
 * - REDISMODULE_REPLY_MAP
 * - REDISMODULE_REPLY_ATTRIBUTE
 */
size_t callReplyGetLen(CallReply *rep) {
    callReplyParse(rep);
    switch(rep->type) {
        case REDISMODULE_REPLY_STRING:
        case REDISMODULE_REPLY_ERROR:
        case REDISMODULE_REPLY_ARRAY:
        case REDISMODULE_REPLY_SET:
        case REDISMODULE_REPLY_MAP:
        case REDISMODULE_REPLY_ATTRIBUTE:
            return rep->len;
        default:
            return 0;
    }
}

static CallReply *callReplyGetCollectionElement(CallReply *rep, size_t idx, int
elements_per_entry) {
    if (idx >= rep->len * elements_per_entry) return NULL; // real len is rep-
>len * elements_per_entry
    return rep->val.array+idx;
}

/* Return a reply array element at a given index. Applicable to:
 * - REDISMODULE_REPLY_ARRAY
 *
 * The return value is borrowed from CallReply, so it must not be freed
 * explicitly or used after CallReply itself is freed.
 */
CallReply *callReplyGetArrayElement(CallReply *rep, size_t idx) {
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_ARRAY) return NULL;
    return callReplyGetCollectionElement(rep, idx, 1);
}

/* Return a reply set element at a given index. Applicable to:
 * - REDISMODULE_REPLY_SET
 *
 * The return value is borrowed from CallReply, so it must not be freed
 * explicitly or used after CallReply itself is freed.
 */

```

```

CallReply *callReplyGetSetElement(CallReply *rep, size_t idx) {
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_SET) return NULL;
    return callReplyGetCollectionElement(rep, idx, 1);
}

static int callReplyGetMapElementInternal(CallReply *rep, size_t idx, CallReply
**key, CallReply **val, int type) {
    callReplyParse(rep);
    if (rep->type != type) return C_ERR;
    if (idx >= rep->len) return C_ERR;
    if (key) *key = callReplyGetCollectionElement(rep, idx * 2, 2);
    if (val) *val = callReplyGetCollectionElement(rep, idx * 2 + 1, 2);
    return C_OK;
}

/* Retrieve a map reply key and value at a given index. Applicable to:
 * - REDISMODULE_REPLY_MAP
 *
 * The key and value are returned by reference through key and val,
 * which may also be NULL if not needed.
 *
 * Returns C_OK on success or C_ERR if reply type mismatches, or if idx is out
 * of range.
 *
 * The returned values are borrowed from CallReply, so they must not be freed
 * explicitly or used after CallReply itself is freed.
 */
int callReplyGetMapElement(CallReply *rep, size_t idx, CallReply **key,
CallReply **val) {
    return callReplyGetMapElementInternal(rep, idx, key, val,
REDISMODULE_REPLY_MAP);
}

/* Return reply attribute, or NULL if it does not exist. Applicable to all
replies.
 *
 * The returned values are borrowed from CallReply, so they must not be freed
 * explicitly or used after CallReply itself is freed.
 */
CallReply *callReplyGetAttribute(CallReply *rep) {
    return rep->attribute;
}

/* Retrieve attribute reply key and value at a given index. Applicable to:
 * - REDISMODULE_REPLY_ATTRIBUTE
 *
 * The key and value are returned by reference through key and val,
 * which may also be NULL if not needed.
 *
 * Returns C_OK on success or C_ERR if reply type mismatches, or if idx is out

```

```

* of range.
*
* The returned values are borrowed from CallReply, so they must not be freed
* explicitly or used after CallReply itself is freed.
*/
int callReplyGetAttributeElement(CallReply *rep, size_t idx, CallReply **key,
CallReply **val) {
    return callReplyGetMapElementInternal(rep, idx, key, val,
REDISMODULE_REPLY_MAP);
}

/* Return a big number reply value. Applicable to:
* - REDISMODULE_REPLY_BIG_NUMBER
*
* The returned values are borrowed from CallReply, so they must not be freed
* explicitly or used after CallReply itself is freed.
*
* The return value is guaranteed to be a big number, as described in the RESP3
* protocol specifications.
*
* The returned value is not NULL terminated and its length is returned by
* reference through len, which must not be NULL.
*/
const char *callReplyGetBigNumber(CallReply *rep, size_t *len) {
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_BIG_NUMBER) return NULL;
    *len = rep->len;
    return rep->val.str;
}

/* Return a verbatim string reply value. Applicable to:
* - REDISMODULE_REPLY_VERBATIM_STRING
*
* If format is non-NULL, the verbatim reply format is also returned by value.
*
* The optional output argument can be given to get a verbatim reply
* format, or can be set NULL if not needed.
*
* The return value is borrowed from CallReply, so it must not be freed
* explicitly or used after CallReply itself is freed.
*
* The returned value is not NULL terminated and its length is returned by
* reference through len, which must not be NULL.
*/
const char *callReplyGetVerbatim(CallReply *rep, size_t *len, const char
**format){
    callReplyParse(rep);
    if (rep->type != REDISMODULE_REPLY_VERBATIM_STRING) return NULL;
    *len = rep->len;
    if (format) *format = rep->val.verbatim_str.format;
    return rep->val.verbatim_str.str;
}

```

```

}

/* Return the current reply blob.
 *
 * The return value is borrowed from CallReply, so it must not be freed
 * explicitly or used after CallReply itself is freed.
 */
const char *callReplyGetProto(CallReply *rep, size_t *proto_len) {
    *proto_len = rep->proto_len;
    return rep->proto;
}

/* Return CallReply private data, as set by the caller on callReplyCreate().
 */
void *callReplyGetPrivateData(CallReply *rep) {
    return rep->private_data;
}

/* Return true if the reply or one of its sub-replies is RESP3 formatted. */
int callReplyIsResp3(CallReply *rep) {
    return rep->flags & REPLY_FLAG_RESP3;
}

/* Returns a list of errors in sds form, or NULL. */
list *callReplyDeferredErrorList(CallReply *rep) {
    return rep->deferred_error_list;
}

/* Create a new CallReply struct from the reply blob.
 *
 * The function will own the reply blob, so it must not be used or freed by
 * the caller after passing it to this function.
 *
 * The reply blob will be freed when the returned CallReply struct is later
 * freed using freeCallReply().
 *
 * The deferred_error_list is an optional list of errors that are present
 * in the reply blob, if given, this function will take ownership on it.
 *
 * The private_data is optional and can later be accessed using
 * callReplyGetPrivateData().
 *
 * NOTE: The parser used for parsing the reply and producing CallReply is
 * designed to handle valid replies created by Redis itself. IT IS NOT
 * DESIGNED TO HANDLE USER INPUT and using it to parse invalid replies is
 * unsafe.
 */
CallReply *callReplyCreate(sds reply, list *deferred_error_list, void
*private_data) {
    CallReply *res = zmalloc(sizeof(*res));
    res->flags = REPLY_FLAG_ROOT;

```

```

    res->original_proto = reply;
    res->proto = reply;
    res->proto_len = sdslen(reply);
    res->private_data = private_data;
    res->attribute = NULL;
    res->deferred_error_list = deferred_error_list;
    return res;
}

/* Create a new CallReply struct from the reply blob representing an error
message.
 * Automatically creating deferred_error_list and set a copy of the reply in
it.
 * Refer to callReplyCreate for detailed explanation. */
CallReply *callReplyCreateError(sds reply, void *private_data) {
    sds err_buff = reply;
    if (err_buff[0] != '-') {
        err_buff = sdscatfmt(sdsempty(), "-ERR %S\r\n", reply);
        sdsfree(reply);
    }
    list *deferred_error_list = listCreate();
    listSetFreeMethod(deferred_error_list, (void (*)(void*))sdsfree);
    listAddNodeTail(deferred_error_list, sdsnew(err_buff));
    return callReplyCreate(err_buff, deferred_error_list, private_data);
}

```

/call_reply.h

[to top](#)

```

/*
 * Copyright (c) 2009-2021, Redis Labs Ltd.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef SRC_CALL_REPLY_H_
#define SRC_CALL_REPLY_H_

#include "resp_parser.h"

typedef struct CallReply CallReply;

CallReply *callReplyCreate(sds reply, list *deferred_error_list, void
*private_data);
CallReply *callReplyCreateError(sds reply, void *private_data);
int callReplyType(CallReply *rep);
const char *callReplyGetString(CallReply *rep, size_t *len);
long long callReplyGetLongLong(CallReply *rep);
double callReplyGetDouble(CallReply *rep);
int callReplyGetBool(CallReply *rep);
size_t callReplyGetLen(CallReply *rep);
CallReply *callReplyGetArrayElement(CallReply *rep, size_t idx);
CallReply *callReplyGetSetElement(CallReply *rep, size_t idx);
int callReplyGetMapElement(CallReply *rep, size_t idx, CallReply **key,
CallReply **val);
CallReply *callReplyGetAttribute(CallReply *rep);

```



```
int callReplyGetAttributeElement(CallReply *rep, size_t idx, CallReply **key,
CallReply **val);
const char *callReplyGetBigNumber(CallReply *rep, size_t *len);
const char *callReplyGetVerbatim(CallReply *rep, size_t *len, const char
**format);
const char *callReplyGetProto(CallReply *rep, size_t *len);
void *callReplyGetPrivateData(CallReply *rep);
int callReplyIsResp3(CallReply *rep);
list *callReplyDeferredErrorList(CallReply *rep);
void freeCallReply(CallReply *rep);

#endif /* SRC_CALL_REPLY_H_ */
```

/childinfo.c

[to top](#)

```

/*
 * Copyright (c) 2016, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "server.h"
#include <unistd.h>
#include <fcntl.h>

typedef struct {
    size_t keys;
    size_t cow;
    monotime cow_updated;
    double progress;
    childInfoType information_type; /* Type of information */
} child_info_data;

/* Open a child-parent channel used in order to move information about the
 * RDB / AOF saving process from the child to the parent (for instance
 * the amount of copy on write memory used) */
void openChildInfoPipe(void) {
    if (anetPipe(server.child_info_pipe, 0_NONBLOCK, 0) == -1) {
        /* On error our two file descriptors should be still set to -1,
         * but we call anyway closeChildInfoPipe() since can't hurt. */
        closeChildInfoPipe();
    } else {

```

```

        server.child_info_nread = 0;
    }
}

/* Close the pipes opened with openChildInfoPipe(). */
void closeChildInfoPipe(void) {
    if (server.child_info_pipe[0] != -1 ||
        server.child_info_pipe[1] != -1)
    {
        close(server.child_info_pipe[0]);
        close(server.child_info_pipe[1]);
        server.child_info_pipe[0] = -1;
        server.child_info_pipe[1] = -1;
        server.child_info_nread = 0;
    }
}

/* Send save data to parent. */
void sendChildInfoGeneric(childInfoType info_type, size_t keys, double
progress, char *pname) {
    if (server.child_info_pipe[1] == -1) return;

    static monotime cow_updated = 0;
    static uint64_t cow_update_cost = 0;
    static size_t cow = 0;
    static size_t peak_cow = 0;
    static size_t update_count = 0;
    static unsigned long long sum_cow = 0;

    child_info_data data = {0}; /* zero everything, including padding to
satisfy valgrind */

    /* When called to report current info, we need to throttle down CoW updates
as they
    * can be very expensive. To do that, we measure the time it takes to get a
reading
    * and schedule the next reading to happen not before
time*CHILD_COW_COST_FACTOR
    * passes. */

    monotime now = getMonotonicUs();
    if (info_type != CHILD_INFO_TYPE_CURRENT_INFO ||
        !cow_updated ||
        now - cow_updated > cow_update_cost * CHILD_COW_DUTY_CYCLE)
    {
        cow = zmalloc_get_private_dirty(-1);
        cow_updated = getMonotonicUs();
        cow_update_cost = cow_updated - now;
        if (cow > peak_cow) peak_cow = cow;
        sum_cow += cow;
        update_count++;
    }
}

```

```

    int cow_info = (info_type != CHILD_INFO_TYPE_CURRENT_INFO);
    if (cow || cow_info) {
        serverLog(cow_info ? LL_NOTICE : LL_VERBOSE,
                  "Fork CoW for %s: current %zu MB, peak %zu MB, average
%llu MB",
                  pname, cow>>20, peak_cow>>20,
(sum_cow/update_count)>>20);
    }
}

data.information_type = info_type;
data.keys = keys;
data.cow = cow;
data.cow_updated = cow_updated;
data.progress = progress;

ssize_t wlen = sizeof(data);

if (write(server.child_info_pipe[1], &data, wlen) != wlen) {
    /* Nothing to do on error, this will be detected by the other side. */
}
}

/* Update Child info. */
void updateChildInfo(childInfoType information_type, size_t cow, monotime
cow_updated, size_t keys, double progress) {
    if (cow > server.stat_current_cow_peak) server.stat_current_cow_peak = cow;

    if (information_type == CHILD_INFO_TYPE_CURRENT_INFO) {
        server.stat_current_cow_bytes = cow;
        server.stat_current_cow_updated = cow_updated;
        server.stat_current_save_keys_processed = keys;
        if (progress != -1) server.stat_module_progress = progress;
    } else if (information_type == CHILD_INFO_TYPE_AOF_COW_SIZE) {
        server.stat_aof_cow_bytes = server.stat_current_cow_peak;
    } else if (information_type == CHILD_INFO_TYPE_RDB_COW_SIZE) {
        server.stat_rdb_cow_bytes = server.stat_current_cow_peak;
    } else if (information_type == CHILD_INFO_TYPE_MODULE_COW_SIZE) {
        server.stat_module_cow_bytes = server.stat_current_cow_peak;
    }
}

/* Read child info data from the pipe.
 * if complete data read into the buffer,
 * data is stored into *buffer, and returns 1.
 * otherwise, the partial data is left in the buffer, waiting for the next
read, and returns 0. */
int readChildInfo(childInfoType *information_type, size_t *cow, monotime
*cow_updated, size_t *keys, double* progress) {
    /* We are using here a static buffer in combination with the

```

```

server.child_info_nread to handle short reads */
static child_info_data buffer;
ssize_t wlen = sizeof(buffer);

/* Do not overlap */
if (server.child_info_nread == wlen) server.child_info_nread = 0;

int nread = read(server.child_info_pipe[0], (char *)&buffer +
server.child_info_nread, wlen - server.child_info_nread);
if (nread > 0) {
    server.child_info_nread += nread;
}

/* We have complete child info */
if (server.child_info_nread == wlen) {
    *information_type = buffer.information_type;
    *cow = buffer.cow;
    *cow_updated = buffer.cow_updated;
    *keys = buffer.keys;
    *progress = buffer.progress;
    return 1;
} else {
    return 0;
}
}

/* Receive info data from child. */
void receiveChildInfo(void) {
    if (server.child_info_pipe[0] == -1) return;

    size_t cow;
    monotime cow_updated;
    size_t keys;
    double progress;
    childInfoType information_type;

    /* Drain the pipe and update child info so that we get the final message.
    */
    while (readChildInfo(&information_type, &cow, &cow_updated, &keys,
&progress)) {
        updateChildInfo(information_type, cow, cow_updated, keys, progress);
    }
}

```

/cli_common.c

[to top](#)

```

/* CLI (command line interface) common methods
 *
 * Copyright (c) 2020, Redis Labs
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "fmacros.h"
#include "cli_common.h"
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <hiredis.h>
#include <sdscompat.h> /* Use hiredis' sds compat header that maps sds calls to
their hi_ variants */
#include <sds.h> /* use sds.h from hiredis, so that only one set of sds
functions will be present in the binary */
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#ifdef USE_OPENSSL
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <hiredis_ssl.h>
#endif

```

```

#define UNUSED(V) ((void) V)

/* Wrapper around redisSecureConnection to avoid hiredis_ssl dependencies if
 * not building with TLS support.
 */
int cliSecureConnection(redisContext *c, cliSSLconfig config, const char **err)
{
#ifdef USE_OPENSSL
    static SSL_CTX *ssl_ctx = NULL;

    if (!ssl_ctx) {
        ssl_ctx = SSL_CTX_new(SSLv23_client_method());
        if (!ssl_ctx) {
            *err = "Failed to create SSL_CTX";
            goto error;
        }
        SSL_CTX_set_options(ssl_ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3);
        SSL_CTX_set_verify(ssl_ctx, config.skip_cert_verify ? SSL_VERIFY_NONE :
SSL_VERIFY_PEER, NULL);

        if (config.cacert || config.cacertdir) {
            if (!SSL_CTX_load_verify_locations(ssl_ctx, config.cacert,
config.cacertdir)) {
                *err = "Invalid CA Certificate File/Directory";
                goto error;
            }
        } else {
            if (!SSL_CTX_set_default_verify_paths(ssl_ctx)) {
                *err = "Failed to use default CA paths";
                goto error;
            }
        }

        if (config.cert && !SSL_CTX_use_certificate_chain_file(ssl_ctx,
config.cert)) {
            *err = "Invalid client certificate";
            goto error;
        }

        if (config.key && !SSL_CTX_use_PrivateKey_file(ssl_ctx, config.key,
SSL_FILETYPE_PEM)) {
            *err = "Invalid private key";
            goto error;
        }

        if (config.ciphers && !SSL_CTX_set_cipher_list(ssl_ctx,
config.ciphers)) {
            *err = "Error while configuring ciphers";
            goto error;
        }
    }
#ifdef TLS1_3_VERSION
    if (config.ciphersuites && !SSL_CTX_set_ciphersuites(ssl_ctx,

```

```

config.ciphersuites)) {
    *err = "Error while setting cypher suites";
    goto error;
}
#endif
}

SSL *ssl = SSL_new(ssl_ctx);
if (!ssl) {
    *err = "Failed to create SSL object";
    return REDIS_ERR;
}

if (config.sni && !SSL_set_tlsext_host_name(ssl, config.sni)) {
    *err = "Failed to configure SNI";
    SSL_free(ssl);
    return REDIS_ERR;
}

return redisInitiateSSL(c, ssl);

error:
    SSL_CTX_free(ssl_ctx);
    ssl_ctx = NULL;
    return REDIS_ERR;
#else
    (void) config;
    (void) c;
    (void) err;
    return REDIS_OK;
#endif
}

/* Wrapper around hiredis to allow arbitrary reads and writes.
 *
 * We piggybacks on top of hiredis to achieve transparent TLS support,
 * and use its internal buffers so it can co-exist with commands
 * previously/later issued on the connection.
 *
 * Interface is close to enough to read()/write() so things should mostly
 * work transparently.
 */

/* Write a raw buffer through a redisContext. If we already have something
 * in the buffer (leftovers from hiredis operations) it will be written
 * as well.
 */
ssize_t cliWriteConn(redisContext *c, const char *buf, size_t buf_len)
{
    int done = 0;

```



```

/* Append data to buffer which is *usually* expected to be empty
 * but we don't assume that, and write.
 */
c->obuf = sdscatlen(c->obuf, buf, buf_len);
if (redisBufferWrite(c, &done) == REDIS_ERR) {
    if (!(c->flags & REDIS_BLOCK))
        errno = EAGAIN;

    /* On error, we assume nothing was written and we roll back the
     * buffer to its original state.
     */
    if (sdslen(c->obuf) > buf_len)
        sdsrange(c->obuf, 0, -(buf_len+1));
    else
        sdsclear(c->obuf);

    return -1;
}

/* If we're done, free up everything. We may have written more than
 * buf_len (if c->obuf was not initially empty) but we don't have to
 * tell.
 */
if (done) {
    sdsclear(c->obuf);
    return buf_len;
}

/* Write was successful but we have some leftovers which we should
 * remove from the buffer.
 *
 * Do we still have data that was there prior to our buf? If so,
 * restore buffer to it's original state and report no new data was
 * written.
 */
if (sdslen(c->obuf) > buf_len) {
    sdsrange(c->obuf, 0, -(buf_len+1));
    return 0;
}

/* At this point we're sure no prior data is left. We flush the buffer
 * and report how much we've written.
 */
size_t left = sdslen(c->obuf);
sdsclear(c->obuf);
return buf_len - left;
}

/* Wrapper around OpenSSL (libssl and libcrypto) initialisation
 */
int cliSecureInit()

```

```

{
#ifdef USE_OPENSSL
    ERR_load_crypto_strings();
    SSL_load_error_strings();
    SSL_library_init();
#endif
    return REDIS_OK;
}

/* Create an sds from stdin */
sds readArgFromStdin(void) {
    char buf[1024];
    sds arg = sdsempty();

    while(1) {
        int nread = read(fileno(stdin),buf,1024);

        if (nread == 0) break;
        else if (nread == -1) {
            perror("Reading from standard input");
            exit(1);
        }
        arg = sdscatlen(arg,buf,nread);
    }
    return arg;
}

/* Create an sds array from argv, either as-is or by dequoting every
 * element. When quoted is non-zero, may return a NULL to indicate an
 * invalid quoted string.
 *
 * The caller should free the resulting array of sds strings with
 * sdsfreesplitres().
 */
sds *getSdsArrayFromArgv(int argc,char **argv, int quoted) {
    sds *res = sds_malloc(sizeof(sds) * argc);

    for (int j = 0; j < argc; j++) {
        if (quoted) {
            sds unquoted = unquoteCString(argv[j]);
            if (!unquoted) {
                while (--j >= 0) sdsfree(res[j]);
                sds_free(res);
                return NULL;
            }
            res[j] = unquoted;
        } else {
            res[j] = sdsnew(argv[j]);
        }
    }
}

```

```

    return res;
}

/* Unquote a null-terminated string and return it as a binary-safe sds. */
sds unquoteCString(char *str) {
    int count;
    sds *unquoted = sdssplitargs(str, &count);
    sds res = NULL;

    if (unquoted && count == 1) {
        res = unquoted[0];
        unquoted[0] = NULL;
    }

    if (unquoted)
        sdsfreesplitres(unquoted, count);

    return res;
}

/* URL-style percent decoding. */
#define isHexChar(c) (isdigit(c) || ((c) >= 'a' && (c) <= 'f'))
#define decodeHexChar(c) (isdigit(c) ? (c) - '0' : (c) - 'a' + 10)
#define decodeHex(h, l) ((decodeHexChar(h) << 4) + decodeHexChar(l))

static sds percentDecode(const char *pe, size_t len) {
    const char *end = pe + len;
    sds ret = sdsempty();
    const char *curr = pe;

    while (curr < end) {
        if (*curr == '%') {
            if ((end - curr) < 2) {
                fprintf(stderr, "Incomplete URI encoding\n");
                exit(1);
            }

            char h = tolower(*(++curr));
            char l = tolower(*(++curr));
            if (!isHexChar(h) || !isHexChar(l)) {
                fprintf(stderr, "Illegal character in URI encoding\n");
                exit(1);
            }
            char c = decodeHex(h, l);
            ret = sdscatlen(ret, &c, 1);
            curr++;
        } else {
            ret = sdscatlen(ret, curr++, 1);
        }
    }
}

```

```

    return ret;
}

/* Parse a URI and extract the server connection information.
 * URI scheme is based on the provisional specification[1] excluding support
 * for query parameters. Valid URIs are:
 *   scheme:      "redis://"
 *   authority: [[<username> ":"<password> "@"] [<hostname> [":"<port>]]
 *   path:        ["/" [<db>]]
 *
 * [1]: https://www.iana.org/assignments/uri-schemes/prov/redis */
void parseRedisUri(const char *uri, const char* tool_name, cliConnInfo
*connInfo, int *tls_flag) {
#ifdef USE_OPENSSL
    UNUSED(tool_name);
#else
    UNUSED(tls_flag);
#endif

    const char *scheme = "redis://";
    const char *tlsscheme = "rediss://";
    const char *curr = uri;
    const char *end = uri + strlen(uri);
    const char *userinfo, *username, *port, *host, *path;

    /* URI must start with a valid scheme. */
    if (!strncasecmp(tlsscheme, curr, strlen(tlsscheme))) {
#ifdef USE_OPENSSL
        *tls_flag = 1;
        curr += strlen(tlsscheme);
#else
        fprintf(stderr, "rediss:// is only supported when %s is compiled with\n", tool_name);
        exit(1);
#endif
    } else if (!strncasecmp(scheme, curr, strlen(scheme))) {
        curr += strlen(scheme);
    } else {
        fprintf(stderr, "Invalid URI scheme\n");
        exit(1);
    }

    if (curr == end) return;

    /* Extract user info. */
    if ((userinfo = strchr(curr, '@'))) {
        if ((username = strchr(curr, ':')) && username < userinfo) {
            connInfo->user = percentDecode(curr, username - curr);
            curr = username + 1;
        }
    }

```

```

        connInfo->auth = percentDecode(curr, userinfo - curr);
        curr = userinfo + 1;
    }
    if (curr == end) return;

    /* Extract host and port. */
    path = strchr(curr, '/');
    if (*curr != '/') {
        host = path ? path - 1 : end;
        if ((port = strchr(curr, ':')) {
            connInfo->hostport = atoi(port + 1);
            host = port - 1;
        }
        sdsfree(connInfo->hostip);
        connInfo->hostip = sdsnewlen(curr, host - curr + 1);
    }
    curr = path ? path + 1 : end;
    if (curr == end) return;

    /* Extract database number. */
    connInfo->input_dbnum = atoi(curr);
}

void freeCliConnInfo(cliConnInfo connInfo){
    if (connInfo.hostip) sdsfree(connInfo.hostip);
    if (connInfo.auth) sdsfree(connInfo.auth);
    if (connInfo.user) sdsfree(connInfo.user);
}

/*
 * Escape a Unicode string for JSON output (--json), following RFC 7159:
 * https://datatracker.ietf.org/doc/html/rfc7159#section-7
 */
sds escapeJsonString(sds s, const char *p, size_t len) {
    s = sdscatlen(s, "\"", 1);
    while(len--) {
        switch(*p) {
            case '\\':
            case '\"':
                s = sdscatprintf(s, "\\%c", *p);
                break;
            case '\n': s = sdscatlen(s, "\\n", 2); break;
            case '\f': s = sdscatlen(s, "\\f", 2); break;
            case '\r': s = sdscatlen(s, "\\r", 2); break;
            case '\t': s = sdscatlen(s, "\\t", 2); break;
            case '\b': s = sdscatlen(s, "\\b", 2); break;
            default:
                s = sdscatprintf(s, *(unsigned char *)p <= 0x1f ? "\\u%04x" :
"%c", *p);
        }
        p++;
    }
}

```

```
    }  
    return sdscatlen(s,"\\",1);  
}
```

/cli_common.h

[to top](#)

```

#ifndef __CLICOMMON_H
#define __CLICOMMON_H

#include <hiredis.h>
#include <sdscompat.h> /* Use hiredis' sds compat header that maps sds calls to
their hi_ variants */

typedef struct cliSSLconfig {
    /* Requested SNI, or NULL */
    char *sni;
    /* CA Certificate file, or NULL */
    char *cacert;
    /* Directory where trusted CA certificates are stored, or NULL */
    char *cacertdir;
    /* Skip server certificate verification. */
    int skip_cert_verify;
    /* Client certificate to authenticate with, or NULL */
    char *cert;
    /* Private key file to authenticate with, or NULL */
    char *key;
    /* Preferred cipher list, or NULL (applies only to <= TLSv1.2) */
    char* ciphers;
    /* Preferred ciphersuites list, or NULL (applies only to TLSv1.3) */
    char* ciphersuites;
} cliSSLconfig;

/* server connection information object, used to describe an ip:port pair, db
num user input, and user:pass. */
typedef struct cliConnInfo {
    char *hostip;
    int hostport;
    int input_dbnum;
    char *auth;
    char *user;
} cliConnInfo;

int cliSecureConnection(redisContext *c, cliSSLconfig config, const char
**err);

ssize_t cliWriteConn(redisContext *c, const char *buf, size_t buf_len);

int cliSecureInit();

sds readArgFromStdin(void);

sds *getSdsArrayFromArgv(int argc, char **argv, int quoted);

sds unquoteCString(char *str);

```

```
void parseRedisUri(const char *uri, const char* tool_name, cliConnInfo
*connInfo, int *tls_flag);

void freeCliConnInfo(cliConnInfo connInfo);

sds escapeJsonString(sds s, const char *p, size_t len);

#endif /* __CLICOMMON_H */
```

/cluster.c

[to top](#)


```

/* Redis Cluster implementation.
 *
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "server.h"
#include "cluster.h"
#include "endianconv.h"

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <math.h>

/* A global reference to myself is handy to make code more clear.
 * Myself always points to server.cluster->myself, that is, the clusterNode
 * that represents this node. */
clusterNode *myself = NULL;

clusterNode *createClusterNode(char *nodename, int flags);
void clusterAddNode(clusterNode *node);

```

```

void clusterAcceptHandler(aeEventLoop *el, int fd, void *privdata, int mask);
void clusterReadHandler(connection *conn);
void clusterSendPing(clusterLink *link, int type);
void clusterSendFail(char *nodename);
void clusterSendFailoverAuthIfNeeded(clusterNode *node, clusterMsg *request);
void clusterUpdateState(void);
int clusterNodeGetSlotBit(clusterNode *n, int slot);
sds clusterGenNodesDescription(int filter, int use_pport);
list *clusterGetNodesServingMySlots(clusterNode *node);
int clusterNodeAddSlave(clusterNode *master, clusterNode *slave);
int clusterAddSlot(clusterNode *n, int slot);
int clusterDelSlot(int slot);
int clusterDelNodeSlots(clusterNode *node);
int clusterNodeSetSlotBit(clusterNode *n, int slot);
void clusterSetMaster(clusterNode *n);
void clusterHandleSlaveFailover(void);
void clusterHandleSlaveMigration(int max_slaves);
int bitmapTestBit(unsigned char *bitmap, int pos);
void clusterDoBeforeSleep(int flags);
void clusterSendUpdate(clusterLink *link, clusterNode *node);
void resetManualFailover(void);
void clusterCloseAllSlots(void);
void clusterSetNodeAsMaster(clusterNode *n);
void clusterDelNode(clusterNode *delnode);
sds representClusterNodeFlags(sds ci, uint16_t flags);
sds representSlotInfo(sds ci, uint16_t *slot_info_pairs, int
slot_info_pairs_count);
void clusterFreeNodesSlotsInfo(clusterNode *n);
uint64_t clusterGetMaxEpoch(void);
int clusterBumpConfigEpochWithoutConsensus(void);
void moduleCallClusterReceivers(const char *sender_id, uint64_t module_id,
uint8_t type, const unsigned char *payload, uint32_t len);
const char *clusterGetMessageTypeString(int type);
void removeChannelsInSlot(unsigned int slot);
unsigned int countKeysInSlot(unsigned int hashslot);
unsigned int countChannelsInSlot(unsigned int hashslot);
unsigned int delKeysInSlot(unsigned int hashslot);

/* Links to the next and previous entries for keys in the same slot are stored
 * in the dict entry metadata. See Slot to Key API below. */
#define dictEntryNextInSlot(de) \
    (((clusterDictEntryMetadata *)dictMetadata(de))->next)
#define dictEntryPrevInSlot(de) \
    (((clusterDictEntryMetadata *)dictMetadata(de))->prev)

#define RCVBUF_INIT_LEN 1024
#define RCVBUF_MAX_PREALLOC (1<20) /* 1MB */

/* Cluster nodes hash table, mapping nodes addresses 1.2.3.4:6379 to
 * clusterNode structures. */
dictType clusterNodesDictType = {

```

```

        dictSdsHash,                /* hash function */
        NULL,                       /* key dup */
        NULL,                       /* val dup */
        dictSdsKeyCompare,          /* key compare */
        dictSdsDestructor,          /* key destructor */
        NULL,                       /* val destructor */
        NULL                        /* allow to expand */
};

/* Cluster re-addition blacklist. This maps node IDs to the time
 * we can re-add this node. The goal is to avoid reading a removed
 * node for some time. */
dictType clusterNodesBlackListDictType = {
        dictSdsCaseHash,            /* hash function */
        NULL,                      /* key dup */
        NULL,                      /* val dup */
        dictSdsKeyCaseCompare,      /* key compare */
        dictSdsDestructor,          /* key destructor */
        NULL,                      /* val destructor */
        NULL                        /* allow to expand */
};

/* -----
 * Initialization
 * -----
*/

/* Load the cluster config from 'filename'.
 *
 * If the file does not exist or is zero-length (this may happen because
 * when we lock the nodes.conf file, we create a zero-length one for the
 * sake of locking if it does not already exist), C_ERR is returned.
 * If the configuration was loaded from the file, C_OK is returned. */
int clusterLoadConfig(char *filename) {
    FILE *fp = fopen(filename,"r");
    struct stat sb;
    char *line;
    int maxline, j;

    if (fp == NULL) {
        if (errno == ENOENT) {
            return C_ERR;
        } else {
            serverLog(LL_WARNING,
                "Loading the cluster node config from %s: %s",
                filename, strerror(errno));
            exit(1);
        }
    }
}

```

```

if (redis_fstat(fileno(fp), &sb) == -1) {
    serverLog(LL_WARNING,
        "Unable to obtain the cluster node config file stat %s: %s",
        filename, strerror(errno));
    exit(1);
}
/* Check if the file is zero-length: if so return C_ERR to signal
 * we have to write the config. */
if (sb.st_size == 0) {
    fclose(fp);
    return C_ERR;
}

/* Parse the file. Note that single lines of the cluster config file can
 * be really long as they include all the hash slots of the node.
 * This means in the worst possible case, half of the Redis slots will be
 * present in a single line, possibly in importing or migrating state, so
 * together with the node ID of the sender/receiver.
 *
 * To simplify we allocate 1024+CLUSTER_SLOTS*128 bytes per line. */
maxline = 1024+CLUSTER_SLOTS*128;
line = zmalloc(maxline);
while(fgets(line, maxline, fp) != NULL) {
    int argc;
    sds *argv;
    clusterNode *n, *master;
    char *p, *s;

    /* Skip blank lines, they can be created either by users manually
     * editing nodes.conf or by the config writing process if stopped
     * before the truncate() call. */
    if (line[0] == '\n' || line[0] == '\0') continue;

    /* Split the line into arguments for processing. */
    argv = sdssplitargs(line, &argc);
    if (argv == NULL) goto fmterr;

    /* Handle the special "vars" line. Don't pretend it is the last
     * line even if it actually is when generated by Redis. */
    if (strcasecmp(argv[0], "vars") == 0) {
        if (!(argc % 2)) goto fmterr;
        for (j = 1; j < argc; j += 2) {
            if (strcasecmp(argv[j], "currentEpoch") == 0) {
                server.cluster->currentEpoch =
                    strtoull(argv[j+1], NULL, 10);
            } else if (strcasecmp(argv[j], "lastVoteEpoch") == 0) {
                server.cluster->lastVoteEpoch =
                    strtoull(argv[j+1], NULL, 10);
            } else {
                serverLog(LL_WARNING,
                    "Skipping unknown cluster config variable '%s'",

```

```

        argv[j]);
    }
}
sdsfreesplitres(argv,argc);
continue;
}

/* Regular config lines have at least eight fields */
if (argc < 8) {
    sdsfreesplitres(argv,argc);
    goto fmterr;
}

/* Create this node if it does not exist */
if (verifyClusterNodeId(argv[0], sdslen(argv[0])) == C_ERR) {
    sdsfreesplitres(argv, argc);
    goto fmterr;
}
n = clusterLookupNode(argv[0], sdslen(argv[0]));
if (!n) {
    n = createClusterNode(argv[0],0);
    clusterAddNode(n);
}

/* Format for the node address information:
 * ip:port[@cport][,hostname] */

/* Hostname is an optional argument that defines the endpoint
 * that can be reported to clients instead of IP. */
char *hostname = strchr(argv[1], ',');
if (hostname) {
    *hostname = '\0';
    hostname++;
    n->hostname = sdscpy(n->hostname, hostname);
} else if (sdslen(n->hostname) != 0) {
    sdsclear(n->hostname);
}

/* Address and port */
if ((p = strrchr(argv[1],':')) == NULL) {
    sdsfreesplitres(argv,argc);
    goto fmterr;
}
*p = '\0';
memcpy(n->ip,argv[1],strlen(argv[1])+1);
char *port = p+1;
char *busp = strchr(port,'@');
if (busp) {
    *busp = '\0';
    busp++;
}
n->port = atoi(port);

```

```

/* In older versions of nodes.conf the "@busport" part is missing.
 * In this case we set it to the default offset of 10000 from the
 * base port. */
n->cport = busp ? atoi(busp) : n->port + CLUSTER_PORT_INCR;

/* The plaintext port for client in a TLS cluster (n->pport) is not
 * stored in nodes.conf. It is received later over the bus protocol. */

/* Parse flags */
p = s = argv[2];
while(p) {
    p = strchr(s,',');
    if (p) *p = '\0';
    if (!strcasecmp(s,"myself")) {
        serverAssert(server.cluster->myself == NULL);
        myself = server.cluster->myself = n;
        n->flags |= CLUSTER_NODE_MYSELF;
    } else if (!strcasecmp(s,"master")) {
        n->flags |= CLUSTER_NODE_MASTER;
    } else if (!strcasecmp(s,"slave")) {
        n->flags |= CLUSTER_NODE_SLAVE;
    } else if (!strcasecmp(s,"fail?")) {
        n->flags |= CLUSTER_NODE_PFAIL;
    } else if (!strcasecmp(s,"fail")) {
        n->flags |= CLUSTER_NODE_FAIL;
        n->fail_time = mstime();
    } else if (!strcasecmp(s,"handshake")) {
        n->flags |= CLUSTER_NODE_HANDSHAKE;
    } else if (!strcasecmp(s,"noaddr")) {
        n->flags |= CLUSTER_NODE_NOADDR;
    } else if (!strcasecmp(s,"nofailover")) {
        n->flags |= CLUSTER_NODE_NOFAILOVER;
    } else if (!strcasecmp(s,"noflags")) {
        /* nothing to do */
    } else {
        serverPanic("Unknown flag in redis cluster config file");
    }
    if (p) s = p+1;
}

/* Get master if any. Set the master and populate master's
 * slave list. */
if (argv[3][0] != '-') {
    if (verifyClusterNodeId(argv[3], sdslen(argv[3])) == C_ERR) {
        sdsfreesplitres(argv, argc);
        goto fmterr;
    }
    master = clusterLookupNode(argv[3], sdslen(argv[3]));
    if (!master) {
        master = createClusterNode(argv[3],0);
        clusterAddNode(master);
    }
}

```

```

    }
    n->slaveof = master;
    clusterNodeAddSlave(master,n);
}

/* Set ping sent / pong received timestamps */
if (atoi(argv[4])) n->ping_sent = mstime();
if (atoi(argv[5])) n->pong_received = mstime();

/* Set configEpoch for this node. */
n->configEpoch = strtoull(argv[6],NULL,10);

/* Populate hash slots served by this instance. */
for (j = 8; j < argc; j++) {
    int start, stop;

    if (argv[j][0] == '[') {
        /* Here we handle migrating / importing slots */
        int slot;
        char direction;
        clusterNode *cn;

        p = strchr(argv[j],'-');
        serverAssert(p != NULL);
        *p = '\0';
        direction = p[1]; /* Either '>' or '<' */
        slot = atoi(argv[j]+1);
        if (slot < 0 || slot >= CLUSTER_SLOTS) {
            sdsfreesplitres(argv,argc);
            goto fmterr;
        }
        p += 3;

        char *pr = strchr(p, ']');
        size_t node_len = pr - p;
        if (pr == NULL || verifyClusterNodeId(p, node_len) == C_ERR) {
            sdsfreesplitres(argv, argc);
            goto fmterr;
        }
        cn = clusterLookupNode(p, CLUSTER_NAMELEN);
        if (!cn) {
            cn = createClusterNode(p,0);
            clusterAddNode(cn);
        }
        if (direction == '>') {
            server.cluster->migrating_slots_to[slot] = cn;
        } else {
            server.cluster->importing_slots_from[slot] = cn;
        }
        continue;
    } else if ((p = strchr(argv[j],'-')) != NULL) {

```

```

        *p = '\0';
        start = atoi(argv[j]);
        stop = atoi(p+1);
    } else {
        start = stop = atoi(argv[j]);
    }
    if (start < 0 || start >= CLUSTER_SLOTS ||
        stop < 0 || stop >= CLUSTER_SLOTS)
    {
        sdsfreesplitres(argv,argc);
        goto fmterr;
    }
    while(start <= stop) clusterAddSlot(n, start++);
}

sdsfreesplitres(argv,argc);
}
/* Config sanity check */
if (server.cluster->myself == NULL) goto fmterr;

zfree(line);
fclose(fp);

serverLog(LL_NOTICE,"Node configuration loaded, I'm %.40s", myself->name);

/* Something that should never happen: currentEpoch smaller than
 * the max epoch found in the nodes configuration. However we handle this
 * as some form of protection against manual editing of critical files. */
if (clusterGetMaxEpoch() > server.cluster->currentEpoch) {
    server.cluster->currentEpoch = clusterGetMaxEpoch();
}
return C_OK;

fmterr:
    serverLog(LL_WARNING,
        "Unrecoverable error: corrupted cluster config file.");
    zfree(line);
    if (fp) fclose(fp);
    exit(1);
}

/* Cluster node configuration is exactly the same as CLUSTER NODES output.
 *
 * This function writes the node config and returns 0, on error -1
 * is returned.
 *
 * Note: we need to write the file in an atomic way from the point of view
 * of the POSIX filesystem semantics, so that if the server is stopped
 * or crashes during the write, we'll end with either the old file or the
 * new one. Since we have the full payload to write available we can use
 * a single write to write the whole file. If the pre-existing file was

```



```

    * bigger we pad our payload with newlines that are anyway ignored and truncate
    * the file afterward. */
int clusterSaveConfig(int do_fsync) {
    sds ci;
    size_t content_size;
    struct stat sb;
    int fd;

    server.cluster->todo_before_sleep &= ~CLUSTER_TODO_SAVE_CONFIG;

    /* Get the nodes description and concatenate our "vars" directive to
     * save currentEpoch and lastVoteEpoch. */
    ci = clusterGenNodesDescription(CLUSTER_NODE_HANDSHAKE, 0);
    ci = sdscatprintf(ci,"vars currentEpoch %llu lastVoteEpoch %llu\n",
        (unsigned long long) server.cluster->currentEpoch,
        (unsigned long long) server.cluster->lastVoteEpoch);
    content_size = sdslen(ci);

    if ((fd = open(server.cluster_configfile,O_WRONLY|O_CREAT,0644))
        == -1) goto err;

    if (redis_fstat(fd,&sb) == -1) goto err;

    /* Pad the new payload if the existing file length is greater. */
    if (sb.st_size > (off_t)content_size) {
        ci = sdsgrowzero(ci,sb.st_size);
        memset(ci+content_size,'\n',sb.st_size-content_size);
    }

    if (write(fd,ci,sdslen(ci)) != (ssize_t)sdslen(ci)) goto err;
    if (do_fsync) {
        server.cluster->todo_before_sleep &= ~CLUSTER_TODO_FSYNC_CONFIG;
        if (fsync(fd) == -1) goto err;
    }

    /* Truncate the file if needed to remove the final \n padding that
     * is just garbage. */
    if (content_size != sdslen(ci) && ftruncate(fd,content_size) == -1) {
        /* ftruncate() failing is not a critical error. */
    }
    close(fd);
    sdsfree(ci);
    return 0;

err:
    if (fd != -1) close(fd);
    sdsfree(ci);
    return -1;
}

void clusterSaveConfigOrDie(int do_fsync) {

```

```

    if (clusterSaveConfig(do_fsync) == -1) {
        serverLog(LL_WARNING,"Fatal: can't update cluster config file.");
        exit(1);
    }
}

/* Lock the cluster config using flock(), and leaks the file descriptor used to
 * acquire the lock so that the file will be locked forever.
 *
 * This works because we always update nodes.conf with a new version
 * in-place, reopening the file, and writing to it in place (later adjusting
 * the length with ftruncate()).
 *
 * On success C_OK is returned, otherwise an error is logged and
 * the function returns C_ERR to signal a lock was not acquired. */
int clusterLockConfig(char *filename) {
/* flock() does not exist on Solaris
 * and a fcntl-based solution won't help, as we constantly re-open that file,
 * which will release _all_ locks anyway
 */
#ifdef __sun
    /* To lock it, we need to open the file in a way it is created if
     * it does not exist, otherwise there is a race condition with other
     * processes. */
    int fd = open(filename,O_WRONLY|O_CREAT|O_CLOEXEC,0644);
    if (fd == -1) {
        serverLog(LL_WARNING,
            "Can't open %s in order to acquire a lock: %s",
            filename, strerror(errno));
        return C_ERR;
    }

    if (flock(fd,LOCK_EX|LOCK_NB) == -1) {
        if (errno == EWOULDBLOCK) {
            serverLog(LL_WARNING,
                "Sorry, the cluster configuration file %s is already used "
                "by a different Redis Cluster node. Please make sure that "
                "different nodes use different cluster configuration "
                "files.", filename);
        } else {
            serverLog(LL_WARNING,
                "Impossible to lock %s: %s", filename, strerror(errno));
        }
        close(fd);
        return C_ERR;
    }
    /* Lock acquired: leak the 'fd' by not closing it, so that we'll retain the
     * lock to the file as long as the process exists.
     *
     * After fork, the child process will get the fd opened by the parent
     process,

```

```

    * we need save `fd` to `cluster_config_file_lock_fd`, so that in
    redisFork(),
    * it will be closed in the child process.
    * If it is not closed, when the main process is killed -9, but the child
    process
    * (redis-aof-rewrite) is still alive, the fd(lock) will still be held by
    the
    * child process, and the main process will fail to get lock, means fail to
    start. */
    server.cluster_config_file_lock_fd = fd;
#else
    UNUSED(filename);
#endif /* __sun */

    return C_OK;
}

/* Derives our ports to be announced in the cluster bus. */
void deriveAnnouncedPorts(int *announced_port, int *announced_pport,
                           int *announced_cport) {
    int port = server.tls_cluster ? server.tls_port : server.port;
    /* Default announced ports. */
    *announced_port = port;
    *announced_pport = server.tls_cluster ? server.port : 0;
    *announced_cport = server.cluster_port ? server.cluster_port : port +
    CLUSTER_PORT_INCR;

    /* Config overriding announced ports. */
    if (server.tls_cluster && server.cluster_announce_tls_port) {
        *announced_port = server.cluster_announce_tls_port;
        *announced_pport = server.cluster_announce_port;
    } else if (server.cluster_announce_port) {
        *announced_port = server.cluster_announce_port;
    }
    if (server.cluster_announce_bus_port) {
        *announced_cport = server.cluster_announce_bus_port;
    }
}

/* Some flags (currently just the NOFAILOVER flag) may need to be updated
 * in the "myself" node based on the current configuration of the node,
 * that may change at runtime via CONFIG SET. This function changes the
 * set of flags in myself->flags accordingly. */
void clusterUpdateMyselfFlags(void) {
    if (!myself) return;
    int oldflags = myself->flags;
    int nofailover = server.cluster_slave_no_failover ?
        CLUSTER_NODE_NOFAILOVER : 0;
    myself->flags &= ~CLUSTER_NODE_NOFAILOVER;
    myself->flags |= nofailover;
    if (myself->flags != oldflags) {

```

```

        clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                             CLUSTER_TODO_UPDATE_STATE);
    }
}

/* We want to take myself->ip in sync with the cluster-announce-ip option.
 * The option can be set at runtime via CONFIG SET. */
void clusterUpdateMyselfIp(void) {
    if (!myself) return;
    static char *prev_ip = NULL;
    char *curr_ip = server.cluster_announce_ip;
    int changed = 0;

    if (prev_ip == NULL && curr_ip != NULL) changed = 1;
    else if (prev_ip != NULL && curr_ip == NULL) changed = 1;
    else if (prev_ip && curr_ip && strcmp(prev_ip,curr_ip)) changed = 1;

    if (changed) {
        if (prev_ip) zfree(prev_ip);
        prev_ip = curr_ip;

        if (curr_ip) {
            /* We always take a copy of the previous IP address, by
             * duplicating the string. This way later we can check if
             * the address really changed. */
            prev_ip = zstrdup(prev_ip);
            strncpy(myself->ip,server.cluster_announce_ip,NET_IP_STR_LEN-1);
            myself->ip[NET_IP_STR_LEN-1] = '\0';
        } else {
            myself->ip[0] = '\0'; /* Force autodetection. */
        }
    }
}

/* Update the hostname for the specified node with the provided C string. */
static void updateAnnouncedHostname(clusterNode *node, char *new) {
    /* Previous and new hostname are the same, no need to update. */
    if (new && !strcmp(new, node->hostname)) {
        return;
    }

    if (new) {
        node->hostname = sdscpy(node->hostname, new);
    } else if (sdslen(node->hostname) != 0) {
        sdsclear(node->hostname);
    }
}

/* Update my hostname based on server configuration values */
void clusterUpdateMyselfHostname(void) {

```

```

    if (!myself) return;
    updateAnnouncedHostname(myself, server.cluster_announce_hostname);
}

void clusterInit(void) {
    int saveconf = 0;

    server.cluster = zmalloc(sizeof(clusterState));
    server.cluster->myself = NULL;
    server.cluster->currentEpoch = 0;
    server.cluster->state = CLUSTER_FAIL;
    server.cluster->size = 1;
    server.cluster->todo_before_sleep = 0;
    server.cluster->nodes = dictCreate(&clusterNodesDictType);
    server.cluster->nodes_black_list =
        dictCreate(&clusterNodesBlackListDictType);
    server.cluster->failover_auth_time = 0;
    server.cluster->failover_auth_count = 0;
    server.cluster->failover_auth_rank = 0;
    server.cluster->failover_auth_epoch = 0;
    server.cluster->cant_failover_reason = CLUSTER_CANT_FAILOVER_NONE;
    server.cluster->lastVoteEpoch = 0;

    /* Initialize stats */
    for (int i = 0; i < CLUSTERMSG_TYPE_COUNT; i++) {
        server.cluster->stats_bus_messages_sent[i] = 0;
        server.cluster->stats_bus_messages_received[i] = 0;
    }
    server.cluster->stats_pfail_nodes = 0;
    server.cluster->stat_cluster_links_buffer_limit_exceeded = 0;

    memset(server.cluster->slots, 0, sizeof(server.cluster->slots));
    clusterCloseAllSlots();

    /* Lock the cluster config file to make sure every node uses
     * its own nodes.conf. */
    server.cluster_config_file_lock_fd = -1;
    if (clusterLockConfig(server.cluster_configfile) == C_ERR)
        exit(1);

    /* Load or create a new nodes configuration. */
    if (clusterLoadConfig(server.cluster_configfile) == C_ERR) {
        /* No configuration found. We will just use the random name provided
         * by the createClusterNode() function. */
        myself = server.cluster->myself =
            createClusterNode(NULL, CLUSTER_NODE_MYSELF|CLUSTER_NODE_MASTER);
        serverLog(LL_NOTICE, "No cluster configuration found, I'm %.40s",
            myself->name);
        clusterAddNode(myself);
        saveconf = 1;
    }
}

```

```

if (saveconf) clusterSaveConfigOrDie(1);

/* We need a listening TCP port for our cluster messaging needs. */
server.cfd.count = 0;

/* Port sanity check II
 * The other handshake port check is triggered too late to stop
 * us from trying to use a too-high cluster port number. */
int port = server.tls_cluster ? server.tls_port : server.port;
if (!server.cluster_port && port > (65535-CLUSTER_PORT_INCR)) {
    serverLog(LL_WARNING, "Redis port number too high. "
                "Cluster communication port is 10,000 port "
                "numbers higher than your Redis port. "
                "Your Redis port number must be 5535 or less.");
    exit(1);
}
if (!server.bindaddr_count) {
    serverLog(LL_WARNING, "No bind address is configured, but it is
required for the Cluster bus.");
    exit(1);
}
int cport = server.cluster_port ? server.cluster_port : port +
CLUSTER_PORT_INCR;
if (listenToPort(cport, &server.cfd) == C_ERR ) {
    /* Note: the following log text is matched by the test suite. */
    serverLog(LL_WARNING, "Failed listening on port %u (cluster),
aborting.", cport);
    exit(1);
}

if (createSocketAcceptHandler(&server.cfd, clusterAcceptHandler) != C_OK) {
    serverPanic("Unrecoverable error creating Redis Cluster socket accept
handler.");
}

/* Initialize data for the Slot to key API. */
slotToKeyInit(server.db);

/* The slots -> channels map is a radix tree. Initialize it here. */
server.cluster->slots_to_channels = raxNew();

/* Set myself->port/cport/pport to my listening ports, we'll just need to
 * discover the IP address via MEET messages. */
deriveAnnouncedPorts(&myself->port, &myself->pport, &myself->cport);

server.cluster->mf_end = 0;
server.cluster->mf_slave = NULL;
resetManualFailover();
clusterUpdateMyselfFlags();
clusterUpdateMyselfIp();
clusterUpdateMyselfHostname();

```

```

}

/* Reset a node performing a soft or hard reset:
 *
 * 1) All other nodes are forgotten.
 * 2) All the assigned / open slots are released.
 * 3) If the node is a slave, it turns into a master.
 * 4) Only for hard reset: a new Node ID is generated.
 * 5) Only for hard reset: currentEpoch and configEpoch are set to 0.
 * 6) The new configuration is saved and the cluster state updated.
 * 7) If the node was a slave, the whole data set is flushed away. */
void clusterReset(int hard) {
    dictIterator *di;
    dictEntry *de;
    int j;

    /* Turn into master. */
    if (nodeIsSlave(myself)) {
        clusterSetNodeAsMaster(myself);
        replicationUnsetMaster();
        emptyData(-1,EMPTYDB_NO_FLAGS,NULL);
    }

    /* Close slots, reset manual failover state. */
    clusterCloseAllSlots();
    resetManualFailover();

    /* Unassign all the slots. */
    for (j = 0; j < CLUSTER_SLOTS; j++) clusterDelSlot(j);

    /* Forget all the nodes, but myself. */
    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);

        if (node == myself) continue;
        clusterDelNode(node);
    }
    dictReleaseIterator(di);

    /* Hard reset only: set epochs to 0, change node ID. */
    if (hard) {
        sds oldname;

        server.cluster->currentEpoch = 0;
        server.cluster->lastVoteEpoch = 0;
        myself->configEpoch = 0;
        serverLog(LL_WARNING, "configEpoch set to 0 via CLUSTER RESET HARD");

        /* To change the Node ID we need to remove the old name from the
         * nodes table, change the ID, and re-add back with new name. */
    }
}

```

```

        oldname = sdsnewlen(myself->name, CLUSTER_NAMELEN);
        dictDelete(server.cluster->nodes,oldname);
        sdsfree(oldname);
        getRandomHexChars(myself->name, CLUSTER_NAMELEN);
        clusterAddNode(myself);
        serverLog(LL_NOTICE,"Node hard reset, now I'm %.40s", myself->name);
    }

    /* Make sure to persist the new config and update the state. */
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                        CLUSTER_TODO_UPDATE_STATE|
                        CLUSTER_TODO_FSYNC_CONFIG);
}

/* -----
 * CLUSTER communication link
 * -----
 */

clusterLink *createClusterLink(clusterNode *node) {
    clusterLink *link = zmalloc(sizeof(*link));
    link->ctime = mstime();
    link->sndbuf = sdsempty();
    link->rcvbuf = zmalloc(link->rcvbuf_alloc = RCVBUF_INIT_LEN);
    link->rcvbuf_len = 0;
    link->conn = NULL;
    link->node = node;
    /* Related node can only possibly be known at link creation time if this is
    an outbound link */
    link->inbound = (node == NULL);
    if (!link->inbound) {
        node->link = link;
    }
    return link;
}

/* Free a cluster link, but does not free the associated node of course.
 * This function will just make sure that the original node associated
 * with this link will have the 'link' field set to NULL. */
void freeClusterLink(clusterLink *link) {
    if (link->conn) {
        connClose(link->conn);
        link->conn = NULL;
    }
    sdsfree(link->sndbuf);
    zfree(link->rcvbuf);
    if (link->node) {
        if (link->node->link == link) {
            serverAssert(!link->inbound);
            link->node->link = NULL;
        }
    }
}

```



```

        } else if (link->node->inbound_link == link) {
            serverAssert(link->inbound);
            link->node->inbound_link = NULL;
        }
    }
    zfree(link);
}

void setClusterNodeToInboundClusterLink(clusterNode *node, clusterLink *link) {
    serverAssert(!link->node);
    serverAssert(link->inbound);
    if (node->inbound_link) {
        /* A peer may disconnect and then reconnect with us, and it's not
        guaranteed that
        * we would always process the disconnection of the existing inbound
        link before
        * accepting a new existing inbound link. Therefore, it's possible to
        have more than
        * one inbound link from the same node at the same time. */
        serverLog(LL_DEBUG, "Replacing inbound link fd %d from node %.40s with
        fd %d",
            node->inbound_link->conn->fd, node->name, link->conn->fd);
    }
    node->inbound_link = link;
    link->node = node;
}

static void clusterConnAcceptHandler(connection *conn) {
    clusterLink *link;

    if (connGetState(conn) != CONN_STATE_CONNECTED) {
        serverLog(LL_VERBOSE,
            "Error accepting cluster node connection: %s",
            connGetLastError(conn));
        connClose(conn);
        return;
    }

    /* Create a link object we use to handle the connection.
    * It gets passed to the readable handler when data is available.
    * Initially the link->node pointer is set to NULL as we don't know
    * which node is, but the right node is references once we know the
    * node identity. */
    link = createClusterLink(NULL);
    link->conn = conn;
    connSetPrivateData(conn, link);

    /* Register read handler */
    connSetReadHandler(conn, clusterReadHandler);
}

```

```

#define MAX_CLUSTER_ACCEPTS_PER_CALL 1000
void clusterAcceptHandler(aeEventLoop *el, int fd, void *privdata, int mask) {
    int cport, cfd;
    int max = MAX_CLUSTER_ACCEPTS_PER_CALL;
    char cip[NET_IP_STR_LEN];
    UNUSED(el);
    UNUSED(mask);
    UNUSED(privdata);

    /* If the server is starting up, don't accept cluster connections:
     * UPDATE messages may interact with the database content. */
    if (server.masterhost == NULL && server.loading) return;

    while(max-->0) {
        cfd = anetTcpAccept(server.neterr, fd, cip, sizeof(cip), &cport);
        if (cfd == ANET_ERR) {
            if (errno != EWOULDBLOCK)
                serverLog(LL_VERBOSE,
                    "Error accepting cluster node: %s", server.neterr);
            return;
        }

        connection *conn = server.tls_cluster ?
            connCreateAcceptedTLS(cfd, TLS_CLIENT_AUTH_YES) :
            connCreateAcceptedSocket(cfd);

        /* Make sure connection is not in an error state */
        if (connGetState(conn) != CONN_STATE_ACCEPTING) {
            serverLog(LL_VERBOSE,
                "Error creating an accepting connection for cluster node: %s",
                connGetLastError(conn));
            connClose(conn);
            return;
        }
        connEnableTcpNoDelay(conn);
        connKeepAlive(conn, server.cluster_node_timeout * 2);

        /* Use non-blocking I/O for cluster messages. */
        serverLog(LL_VERBOSE, "Accepting cluster node connection from %s:%d",
            cip, cport);

        /* Accept the connection now. connAccept() may call our handler
        directly
         * or schedule it for later depending on connection implementation.
         */
        if (connAccept(conn, clusterConnAcceptHandler) == C_ERR) {
            if (connGetState(conn) == CONN_STATE_ERROR)
                serverLog(LL_VERBOSE,
                    "Error accepting cluster node connection: %s",
                    connGetLastError(conn));
            connClose(conn);
        }
    }
}

```

```

        return;
    }
}

/* Return the approximated number of sockets we are using in order to
 * take the cluster bus connections. */
unsigned long getClusterConnectionsCount(void) {
    /* We decrement the number of nodes by one, since there is the
     * "myself" node too in the list. Each node uses two file descriptors,
     * one incoming and one outgoing, thus the multiplication by 2. */
    return server.cluster_enabled ?
        ((dictSize(server.cluster->nodes)-1)*2) : 0;
}

/* -----
 * Key space handling
 * -----
 */

/* We have 16384 hash slots. The hash slot of a given key is obtained
 * as the least significant 14 bits of the crc16 of the key.
 *
 * However if the key contains the {...} pattern, only the part between
 * { and } is hashed. This may be useful in the future to force certain
 * keys to be in the same node (assuming no resharding is in progress). */
unsigned int keyHashSlot(char *key, int keylen) {
    int s, e; /* start-end indexes of { and } */

    for (s = 0; s < keylen; s++)
        if (key[s] == '{') break;

    /* No '{' ? Hash the whole key. This is the base case. */
    if (s == keylen) return crc16(key,keylen) & 0x3FFF;

    /* '{' found? Check if we have the corresponding '}'. */
    for (e = s+1; e < keylen; e++)
        if (key[e] == '}') break;

    /* No '}' or nothing between {} ? Hash the whole key. */
    if (e == keylen || e == s+1) return crc16(key,keylen) & 0x3FFF;

    /* If we are here there is both a { and a } on its right. Hash
     * what is in the middle between { and }. */
    return crc16(key+s+1,e-s-1) & 0x3FFF;
}

/* -----
 * CLUSTER node API

```

```

* -----
*/

/* Create a new cluster node, with the specified flags.
 * If "nodename" is NULL this is considered a first handshake and a random
 * node name is assigned to this node (it will be fixed later when we'll
 * receive the first pong).
 *
 * The node is created and returned to the user, but it is not automatically
 * added to the nodes hash table. */
clusterNode *createClusterNode(char *nodename, int flags) {
    clusterNode *node = zmalloc(sizeof(*node));

    if (nodename)
        memcpy(node->name, nodename, CLUSTER_NAMELEN);
    else
        getRandomHexChars(node->name, CLUSTER_NAMELEN);
    node->ctime = mstime();
    node->configEpoch = 0;
    node->flags = flags;
    memset(node->slots, 0, sizeof(node->slots));
    node->slot_info_pairs = NULL;
    node->slot_info_pairs_count = 0;
    node->numslots = 0;
    node->numslaves = 0;
    node->slaves = NULL;
    node->slaveof = NULL;
    node->ping_sent = node->pong_received = 0;
    node->data_received = 0;
    node->fail_time = 0;
    node->link = NULL;
    node->inbound_link = NULL;
    memset(node->ip, 0, sizeof(node->ip));
    node->hostname = sdsempty();
    node->port = 0;
    node->cport = 0;
    node->pport = 0;
    node->fail_reports = listCreate();
    node->voted_time = 0;
    node->orphaned_time = 0;
    node->repl_offset_time = 0;
    node->repl_offset = 0;
    listSetFreeMethod(node->fail_reports, zfree);
    return node;
}

/* This function is called every time we get a failure report from a node.
 * The side effect is to populate the fail_reports list (or to update
 * the timestamp of an existing report).
 *
 * 'failing' is the node that is in failure state according to the

```

```

* 'sender' node.
*
* The function returns 0 if it just updates a timestamp of an existing
* failure report from the same sender. 1 is returned if a new failure
* report is created. */
int clusterNodeAddFailureReport(clusterNode *failing, clusterNode *sender) {
    list *l = failing->fail_reports;
    listNode *ln;
    listIter li;
    clusterNodeFailReport *fr;

    /* If a failure report from the same sender already exists, just update
    * the timestamp. */
    listRewind(l,&li);
    while ((ln = listNext(&li)) != NULL) {
        fr = ln->value;
        if (fr->node == sender) {
            fr->time = mstime();
            return 0;
        }
    }

    /* Otherwise create a new report. */
    fr = zmalloc(sizeof(*fr));
    fr->node = sender;
    fr->time = mstime();
    listAddNodeTail(l,fr);
    return 1;
}

/* Remove failure reports that are too old, where too old means reasonably
* older than the global node timeout. Note that anyway for a node to be
* flagged as FAIL we need to have a local PFAIL state that is at least
* older than the global node timeout, so we don't just trust the number
* of failure reports from other nodes. */
void clusterNodeCleanupFailureReports(clusterNode *node) {
    list *l = node->fail_reports;
    listNode *ln;
    listIter li;
    clusterNodeFailReport *fr;
    mstime_t maxtime = server.cluster_node_timeout *
        CLUSTER_FAIL_REPORT_VALIDITY_MULT;
    mstime_t now = mstime();

    listRewind(l,&li);
    while ((ln = listNext(&li)) != NULL) {
        fr = ln->value;
        if (now - fr->time > maxtime) listDelNode(l,ln);
    }
}

```

```

/* Remove the failing report for 'node' if it was previously considered
 * failing by 'sender'. This function is called when a node informs us via
 * gossip that a node is OK from its point of view (no FAIL or PFAIL flags).
 *
 * Note that this function is called relatively often as it gets called even
 * when there are no nodes failing, and is O(N), however when the cluster is
 * fine the failure reports list is empty so the function runs in constant
 * time.
 *
 * The function returns 1 if the failure report was found and removed.
 * Otherwise 0 is returned. */
int clusterNodeDelFailureReport(clusterNode *node, clusterNode *sender) {
    list *l = node->fail_reports;
    listNode *ln;
    listIter li;
    clusterNodeFailReport *fr;

    /* Search for a failure report from this sender. */
    listRewind(l,&li);
    while ((ln = listNext(&li)) != NULL) {
        fr = ln->value;
        if (fr->node == sender) break;
    }
    if (!ln) return 0; /* No failure report from this sender. */

    /* Remove the failure report. */
    listDelNode(l,ln);
    clusterNodeCleanupFailureReports(node);
    return 1;
}

/* Return the number of external nodes that believe 'node' is failing,
 * not including this node, that may have a PFAIL or FAIL state for this
 * node as well. */
int clusterNodeFailureReportsCount(clusterNode *node) {
    clusterNodeCleanupFailureReports(node);
    return listLength(node->fail_reports);
}

int clusterNodeRemoveSlave(clusterNode *master, clusterNode *slave) {
    int j;

    for (j = 0; j < master->numslaves; j++) {
        if (master->slaves[j] == slave) {
            if ((j+1) < master->numslaves) {
                int remaining_slaves = (master->numslaves - j) - 1;
                memmove(master->slaves+j, master->slaves+(j+1),
                    (sizeof(*master->slaves) * remaining_slaves));
            }
            master->numslaves--;
            if (master->numslaves == 0)

```

```

        master->flags &= ~CLUSTER_NODE_MIGRATE_T0;
        return C_OK;
    }
}
return C_ERR;
}

int clusterNodeAddSlave(clusterNode *master, clusterNode *slave) {
    int j;

    /* If it's already a slave, don't add it again. */
    for (j = 0; j < master->numslaves; j++)
        if (master->slaves[j] == slave) return C_ERR;
    master->slaves = zrealloc(master->slaves,
        sizeof(clusterNode)*(master->numslaves+1));
    master->slaves[master->numslaves] = slave;
    master->numslaves++;
    master->flags |= CLUSTER_NODE_MIGRATE_T0;
    return C_OK;
}

int clusterCountNonFailingSlaves(clusterNode *n) {
    int j, okslaves = 0;

    for (j = 0; j < n->numslaves; j++)
        if (!nodeFailed(n->slaves[j])) okslaves++;
    return okslaves;
}

/* Low level cleanup of the node structure. Only called by clusterDelNode(). */
void freeClusterNode(clusterNode *n) {
    sds nodename;
    int j;

    /* If the node has associated slaves, we have to set
     * all the slaves->slaveof fields to NULL (unknown). */
    for (j = 0; j < n->numslaves; j++)
        n->slaves[j]->slaveof = NULL;

    /* Remove this node from the list of slaves of its master. */
    if (nodeIsSlave(n) && n->slaveof) clusterNodeRemoveSlave(n->slaveof,n);

    /* Unlink from the set of nodes. */
    nodename = sdsnewlen(n->name, CLUSTER_NAMELEN);
    serverAssert(dictDelete(server.cluster->nodes,nodename) == DICT_OK);
    sdsfree(nodename);
    sdsfree(n->hostname);

    /* Release links and associated data structures. */
    if (n->link) freeClusterLink(n->link);
    if (n->inbound_link) freeClusterLink(n->inbound_link);
}

```

```

    listRelease(n->fail_reports);
    zfree(n->slaves);
    zfree(n);
}

/* Add a node to the nodes hash table */
void clusterAddNode(clusterNode *node) {
    int retval;

    retval = dictAdd(server.cluster->nodes,
                     sdsnewlen(node->name, CLUSTER_NAMELEN), node);
    serverAssert(retval == DICT_OK);
}

/* Remove a node from the cluster. The function performs the high level
 * cleanup, calling freeClusterNode() for the low level cleanup.
 * Here we do the following:
 *
 * 1) Mark all the slots handled by it as unassigned.
 * 2) Remove all the failure reports sent by this node and referenced by
 *    other nodes.
 * 3) Free the node with freeClusterNode() that will in turn remove it
 *    from the hash table and from the list of slaves of its master, if
 *    it is a slave node.
 */
void clusterDelNode(clusterNode *delnode) {
    int j;
    dictIterator *di;
    dictEntry *de;

    /* 1) Mark slots as unassigned. */
    for (j = 0; j < CLUSTER_SLOTS; j++) {
        if (server.cluster->importing_slots_from[j] == delnode)
            server.cluster->importing_slots_from[j] = NULL;
        if (server.cluster->migrating_slots_to[j] == delnode)
            server.cluster->migrating_slots_to[j] = NULL;
        if (server.cluster->slots[j] == delnode)
            clusterDelSlot(j);
    }

    /* 2) Remove failure reports. */
    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);

        if (node == delnode) continue;
        clusterNodeDelFailureReport(node, delnode);
    }
    dictReleaseIterator(di);

    /* 3) Free the node, unlinking it from the cluster. */

```



```

    freeClusterNode(delnode);
}

/* Cluster node sanity check. Returns C_OK if the node id
 * is valid an C_ERR otherwise. */
int verifyClusterNodeId(const char *name, int length) {
    if (length != CLUSTER_NAMELEN) return C_ERR;
    for (int i = 0; i < length; i++) {
        if (name[i] >= 'a' && name[i] <= 'z') continue;
        if (name[i] >= '0' && name[i] <= '9') continue;
        return C_ERR;
    }
    return C_OK;
}

/* Node lookup by name */
clusterNode *clusterLookupNode(const char *name, int length) {
    if (verifyClusterNodeId(name, length) != C_OK) return NULL;
    sds s = sdsnewlen(name, length);
    dictEntry *de = dictFind(server.cluster->nodes, s);
    sdsfree(s);
    if (de == NULL) return NULL;
    return dictGetVal(de);
}

/* Get all the nodes serving the same slots as myself. */
list *clusterGetNodesServingMySlots(clusterNode *node) {
    list *nodes_for_slot = listCreate();
    clusterNode *my_primary = nodeIsMaster(node) ? node : node->slaveof;
    listAddNodeTail(nodes_for_slot, my_primary);
    for (int i=0; i < my_primary->numslaves; i++) {
        listAddNodeTail(nodes_for_slot, my_primary->slaves[i]);
    }
    return nodes_for_slot;
}

/* This is only used after the handshake. When we connect a given IP/PORT
 * as a result of CLUSTER MEET we don't have the node name yet, so we
 * pick a random one, and will fix it when we receive the PONG request using
 * this function. */
void clusterRenameNode(clusterNode *node, char *newname) {
    int retval;
    sds s = sdsnewlen(node->name, CLUSTER_NAMELEN);

    serverLog(LL_DEBUG, "Renaming node %.40s into %.40s",
        node->name, newname);
    retval = dictDelete(server.cluster->nodes, s);
    sdsfree(s);
    serverAssert(retval == DICT_OK);
    memcpy(node->name, newname, CLUSTER_NAMELEN);
    clusterAddNode(node);
}

```

```

}

/* -----
-
* CLUSTER config epoch handling
* -----
*/

/* Return the greatest configEpoch found in the cluster, or the current
 * epoch if greater than any node configEpoch. */
uint64_t clusterGetMaxEpoch(void) {
    uint64_t max = 0;
    dictIterator *di;
    dictEntry *de;

    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);
        if (node->configEpoch > max) max = node->configEpoch;
    }
    dictReleaseIterator(di);
    if (max < server.cluster->currentEpoch) max = server.cluster->currentEpoch;
    return max;
}

/* If this node epoch is zero or is not already the greatest across the
 * cluster (from the POV of the local configuration), this function will:
 *
 * 1) Generate a new config epoch, incrementing the current epoch.
 * 2) Assign the new epoch to this node, WITHOUT any consensus.
 * 3) Persist the configuration on disk before sending packets with the
 *    new configuration.
 *
 * If the new config epoch is generated and assigned, C_OK is returned,
 * otherwise C_ERR is returned (since the node has already the greatest
 * configuration around) and no operation is performed.
 *
 * Important note: this function violates the principle that config epochs
 * should be generated with consensus and should be unique across the cluster.
 * However Redis Cluster uses this auto-generated new config epochs in two
 * cases:
 *
 * 1) When slots are closed after importing. Otherwise resharding would be
 *    too expensive.
 * 2) When CLUSTER FAILOVER is called with options that force a slave to
 *    failover its master even if there is not master majority able to
 *    create a new configuration epoch.
 *
 * Redis Cluster will not explode using this function, even in the case of
 * a collision between this node and another node, generating the same
 * configuration epoch unilaterally, because the config epoch conflict

```

```

* resolution algorithm will eventually move colliding nodes to different
* config epochs. However using this function may violate the "last failover
* wins" rule, so should only be used with care. */
int clusterBumpConfigEpochWithoutConsensus(void) {
    uint64_t maxEpoch = clusterGetMaxEpoch();

    if (myself->configEpoch == 0 ||
        myself->configEpoch != maxEpoch)
    {
        server.cluster->currentEpoch++;
        myself->configEpoch = server.cluster->currentEpoch;
        clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                             CLUSTER_TODO_FSYNC_CONFIG);
        serverLog(LL_WARNING,
                  "New configEpoch set to %llu",
                  (unsigned long long) myself->configEpoch);
        return C_OK;
    } else {
        return C_ERR;
    }
}

/* This function is called when this node is a master, and we receive from
 * another master a configuration epoch that is equal to our configuration
 * epoch.
 *
 * * BACKGROUND
 *
 * * It is not possible that different slaves get the same config
 * * epoch during a failover election, because the slaves need to get voted
 * * by a majority. However when we perform a manual resharding of the cluster
 * * the node will assign a configuration epoch to itself without to ask
 * * for agreement. Usually resharding happens when the cluster is working well
 * * and is supervised by the sysadmin, however it is possible for a failover
 * * to happen exactly while the node we are resharding a slot to assigns itself
 * * a new configuration epoch, but before it is able to propagate it.
 *
 * * So technically it is possible in this condition that two nodes end with
 * * the same configuration epoch.
 *
 * * Another possibility is that there are bugs in the implementation causing
 * * this to happen.
 *
 * * Moreover when a new cluster is created, all the nodes start with the same
 * * configEpoch. This collision resolution code allows nodes to automatically
 * * end with a different configEpoch at startup automatically.
 *
 * * In all the cases, we want a mechanism that resolves this issue automatically
 * * as a safeguard. The same configuration epoch for masters serving different
 * * set of slots is not harmful, but it is if the nodes end serving the same
 * * slots for some reason (manual errors or software bugs) without a proper

```

```

* failover procedure.
*
* In general we want a system that eventually always ends with different
* masters having different configuration epochs whatever happened, since
* nothing is worse than a split-brain condition in a distributed system.
*
* BEHAVIOR
*
* When this function gets called, what happens is that if this node
* has the lexicographically smaller Node ID compared to the other node
* with the conflicting epoch (the 'sender' node), it will assign itself
* the greatest configuration epoch currently detected among nodes plus 1.
*
* This means that even if there are multiple nodes colliding, the node
* with the greatest Node ID never moves forward, so eventually all the nodes
* end with a different configuration epoch.
*/
void clusterHandleConfigEpochCollision(clusterNode *sender) {
    /* Prerequisites: nodes have the same configEpoch and are both masters. */
    if (sender->configEpoch != myself->configEpoch ||
        !nodeIsMaster(sender) || !nodeIsMaster(myself)) return;
    /* Don't act if the colliding node has a smaller Node ID. */
    if (memcmp(sender->name, myself->name, CLUSTER_NAMELEN) <= 0) return;
    /* Get the next ID available at the best of this node knowledge. */
    server.cluster->currentEpoch++;
    myself->configEpoch = server.cluster->currentEpoch;
    clusterSaveConfigOrDie(1);
    serverLog(LL_VERBOSE,
        "WARNING: configEpoch collision with node %.40s."
        " configEpoch set to %llu",
        sender->name,
        (unsigned long long) myself->configEpoch);
}

/* -----
-
* CLUSTER nodes blacklist
*
* The nodes blacklist is just a way to ensure that a given node with a given
* Node ID is not re-added before some time elapsed (this time is specified
* in seconds in CLUSTER_BLACKLIST_TTL).
*
* This is useful when we want to remove a node from the cluster completely:
* when CLUSTER FORGET is called, it also puts the node into the blacklist so
* that even if we receive gossip messages from other nodes that still remember
* about the node we want to remove, we don't re-add it before some time.
*
* Currently the CLUSTER_BLACKLIST_TTL is set to 1 minute, this means
* that redis-cli has 60 seconds to send CLUSTER FORGET messages to nodes
* in the cluster without dealing with the problem of other nodes re-adding
* back the node to nodes we already sent the FORGET command to.

```

```

*
* The data structure used is a hash table with an sds string representing
* the node ID as key, and the time when it is ok to re-add the node as
* value.
* -----
*/

#define CLUSTER_BLACKLIST_TTL 60      /* 1 minute. */

/* Before of the addNode() or Exists() operations we always remove expired
 * entries from the black list. This is an O(N) operation but it is not a
 * problem since add / exists operations are called very infrequently and
 * the hash table is supposed to contain very little elements at max.
 * However without the cleanup during long uptime and with some automated
 * node add/removal procedures, entries could accumulate. */
void clusterBlacklistCleanup(void) {
    dictIterator *di;
    dictEntry *de;

    di = dictGetSafeIterator(server.cluster->nodes_black_list);
    while((de = dictNext(di)) != NULL) {
        int64_t expire = dictGetUnsignedIntegerVal(de);

        if (expire < server.unixtime)
            dictDelete(server.cluster->nodes_black_list, dictGetKey(de));
    }
    dictReleaseIterator(di);
}

/* Cleanup the blacklist and add a new node ID to the black list. */
void clusterBlacklistAddNode(clusterNode *node) {
    dictEntry *de;
    sds id = sdsnewlen(node->name, CLUSTER_NAMELEN);

    clusterBlacklistCleanup();
    if (dictAdd(server.cluster->nodes_black_list, id, NULL) == DICT_OK) {
        /* If the key was added, duplicate the sds string representation of
         * the key for the next lookup. We'll free it at the end. */
        id = sdsdup(id);
    }
    de = dictFind(server.cluster->nodes_black_list, id);
    dictSetUnsignedIntegerVal(de, time(NULL) + CLUSTER_BLACKLIST_TTL);
    sdsfree(id);
}

/* Return non-zero if the specified node ID exists in the blacklist.
 * You don't need to pass an sds string here, any pointer to 40 bytes
 * will work. */
int clusterBlacklistExists(char *nodeid) {
    sds id = sdsnewlen(nodeid, CLUSTER_NAMELEN);

```

```

    int retval;

    clusterBlacklistCleanup();
    retval = dictFind(server.cluster->nodes_black_list,id) != NULL;
    sdsfree(id);
    return retval;
}

/* -----
-
* CLUSTER messages exchange - PING/PONG and gossip
* -----
*/

/* This function checks if a given node should be marked as FAIL.
* It happens if the following conditions are met:
*
* 1) We received enough failure reports from other master nodes via gossip.
*    Enough means that the majority of the masters signaled the node is
*    down recently.
* 2) We believe this node is in PFAIL state.
*
* If a failure is detected we also inform the whole cluster about this
* event trying to force every other node to set the FAIL flag for the node.
*
* Note that the form of agreement used here is weak, as we collect the
majority
* of masters state during some time, and even if we force agreement by
* propagating the FAIL message, because of partitions we may not reach every
* node. However:
*
* 1) Either we reach the majority and eventually the FAIL state will propagate
*    to all the cluster.
* 2) Or there is no majority so no slave promotion will be authorized and the
*    FAIL flag will be cleared after some time.
*/
void markNodeAsFailingIfNeeded(clusterNode *node) {
    int failures;
    int needed_quorum = (server.cluster->size / 2) + 1;

    if (!nodeTimedOut(node)) return; /* We can reach it. */
    if (nodeFailed(node)) return; /* Already FAILing. */

    failures = clusterNodeFailureReportsCount(node);
    /* Also count myself as a voter if I'm a master. */
    if (nodeIsMaster(myself)) failures++;
    if (failures < needed_quorum) return; /* No weak agreement from masters. */

    serverLog(LL_NOTICE,
        "Marking node %.40s as failing (quorum reached).", node->name);
}

```

```

/* Mark the node as failing. */
node->flags &= ~CLUSTER_NODE_PFAIL;
node->flags |= CLUSTER_NODE_FAIL;
node->fail_time = mstime();

/* Broadcast the failing node name to everybody, forcing all the other
 * reachable nodes to flag the node as FAIL.
 * We do that even if this node is a replica and not a master: anyway
 * the failing state is triggered collecting failure reports from masters,
 * so here the replica is only helping propagating this status. */
clusterSendFail(node->name);
clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|CLUSTER_TODO_SAVE_CONFIG);
}

/* This function is called only if a node is marked as FAIL, but we are able
 * to reach it again. It checks if there are the conditions to undo the FAIL
 * state. */
void clearNodeFailureIfNeeded(clusterNode *node) {
    mstime_t now = mstime();

    serverAssert(nodeFailed(node));

    /* For slaves we always clear the FAIL flag if we can contact the
     * node again. */
    if (nodeIsSlave(node) || node->numslots == 0) {
        serverLog(LL_NOTICE,
            "Clear FAIL state for node %.40s: %s is reachable again.",
            node->name,
            nodeIsSlave(node) ? "replica" : "master without slots");
        node->flags &= ~CLUSTER_NODE_FAIL;

        clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|CLUSTER_TODO_SAVE_CONFIG);
    }

    /* If it is a master and...
     * 1) The FAIL state is old enough.
     * 2) It is yet serving slots from our point of view (not failed over).
     * Apparently no one is going to fix these slots, clear the FAIL flag. */
    if (nodeIsMaster(node) && node->numslots > 0 &&
        (now - node->fail_time) >
        (server.cluster_node_timeout * CLUSTER_FAIL_UNDO_TIME_MULT))
    {
        serverLog(LL_NOTICE,
            "Clear FAIL state for node %.40s: is reachable again and nobody is
            serving its slots after some time.",
            node->name);
        node->flags &= ~CLUSTER_NODE_FAIL;

        clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|CLUSTER_TODO_SAVE_CONFIG);
    }
}

```

```

/* Return true if we already have a node in HANDSHAKE state matching the
 * specified ip address and port number. This function is used in order to
 * avoid adding a new handshake node for the same address multiple times. */
int clusterHandshakeInProgress(char *ip, int port, int cport) {
    dictIterator *di;
    dictEntry *de;

    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);

        if (!nodeInHandshake(node)) continue;
        if (!strcasecmp(node->ip, ip) &&
            node->port == port &&
            node->cport == cport) break;
    }
    dictReleaseIterator(di);
    return de != NULL;
}

/* Start a handshake with the specified address if there is not one
 * already in progress. Returns non-zero if the handshake was actually
 * started. On error zero is returned and errno is set to one of the
 * following values:
 *
 * *
 * * EAGAIN - There is already a handshake in progress for this address.
 * * EINVAL - IP or port are not valid. */
int clusterStartHandshake(char *ip, int port, int cport) {
    clusterNode *n;
    char norm_ip[NET_IP_STR_LEN];
    struct sockaddr_storage sa;

    /* IP sanity check */
    if (inet_pton(AF_INET, ip,
        &(((struct sockaddr_in *)&sa)->sin_addr)))
    {
        sa.ss_family = AF_INET;
    } else if (inet_pton(AF_INET6, ip,
        &(((struct sockaddr_in6 *)&sa)->sin6_addr)))
    {
        sa.ss_family = AF_INET6;
    } else {
        errno = EINVAL;
        return 0;
    }

    /* Port sanity check */
    if (port <= 0 || port > 65535 || cport <= 0 || cport > 65535) {
        errno = EINVAL;
        return 0;
    }
}

```



```

}

/* Set norm_ip as the normalized string representation of the node
 * IP address. */
memset(norm_ip,0,NET_IP_STR_LEN);
if (sa.ss_family == AF_INET)
    inet_ntop(AF_INET,
        (void*)&(((struct sockaddr_in *)&sa)->sin_addr),
        norm_ip,NET_IP_STR_LEN);
else
    inet_ntop(AF_INET6,
        (void*)&(((struct sockaddr_in6 *)&sa)->sin6_addr),
        norm_ip,NET_IP_STR_LEN);

if (clusterHandshakeInProgress(norm_ip,port,cport)) {
    errno = EAGAIN;
    return 0;
}

/* Add the node with a random address (NULL as first argument to
 * createClusterNode()). Everything will be fixed during the
 * handshake. */
n = createClusterNode(NULL,CLUSTER_NODE_HANDSHAKE|CLUSTER_NODE_MEET);
memcpy(n->ip,norm_ip,sizeof(n->ip));
n->port = port;
n->cport = cport;
clusterAddNode(n);
return 1;
}

/* Process the gossip section of PING or PONG packets.
 * Note that this function assumes that the packet is already sanity-checked
 * by the caller, not in the content of the gossip section, but in the
 * length. */
void clusterProcessGossipSection(clusterMsg *hdr, clusterLink *link) {
    uint16_t count = ntohs(hdr->count);
    clusterMsgDataGossip *g = (clusterMsgDataGossip*) hdr->data.ping.gossip;
    clusterNode *sender = link->node ? link->node : clusterLookupNode(hdr->sender, CLUSTER_NAMELEN);

    while(count--) {
        uint16_t flags = ntohs(g->flags);
        clusterNode *node;
        sds ci;

        if (server.verbosity == LL_DEBUG) {
            ci = representClusterNodeFlags(sdsempty(), flags);
            serverLog(LL_DEBUG,"GOSSIP %.40s %s:%d@%d %s",
                g->nodename,
                g->ip,
                ntohs(g->port),

```

```

        ntohs(g->cport),
        ci);
    sdsfree(ci);
}

/* Update our state accordingly to the gossip sections */
node = clusterLookupNode(g->nodename, CLUSTER_NAMELEN);
if (node) {
    /* We already know this node.
       Handle failure reports, only when the sender is a master. */
    if (sender && nodeIsMaster(sender) && node != myself) {
        if (flags & (CLUSTER_NODE_FAIL|CLUSTER_NODE_PFAIL)) {
            if (clusterNodeAddFailureReport(node, sender)) {
                serverLog(LL_VERBOSE,
                    "Node %.40s reported node %.40s as not reachable.",
                    sender->name, node->name);
            }
            markNodeAsFailingIfNeeded(node);
        } else {
            if (clusterNodeDelFailureReport(node, sender)) {
                serverLog(LL_VERBOSE,
                    "Node %.40s reported node %.40s is back online.",
                    sender->name, node->name);
            }
        }
    }
}

/* If from our POV the node is up (no failure flags are set),
 * we have no pending ping for the node, nor we have failure
 * reports for this node, update the last pong time with the
 * one we see from the other nodes. */
if (!(flags & (CLUSTER_NODE_FAIL|CLUSTER_NODE_PFAIL)) &&
    node->ping_sent == 0 &&
    clusterNodeFailureReportsCount(node) == 0)
{
    mstime_t pongtime = ntohl(g->pong_received);
    pongtime *= 1000; /* Convert back to milliseconds. */

    /* Replace the pong time with the received one only if
     * it's greater than our view but is not in the future
     * (with 500 milliseconds tolerance) from the POV of our
     * clock. */
    if (pongtime <= (server.mstime+500) &&
        pongtime > node->pong_received)
    {
        node->pong_received = pongtime;
    }
}

/* If we already know this node, but it is not reachable, and
 * we see a different address in the gossip section of a node that

```

```

    * can talk with this other node, update the address, disconnect
    * the old link if any, so that we'll attempt to connect with the
    * new address. */
    if (node->flags & (CLUSTER_NODE_FAIL|CLUSTER_NODE_PFAIL) &&
        !(flags & CLUSTER_NODE_NOADDR) &&
        !(flags & (CLUSTER_NODE_FAIL|CLUSTER_NODE_PFAIL)) &&
        (strcasecmp(node->ip,g->ip) ||
         node->port != ntohs(g->port) ||
         node->cport != ntohs(g->cport)))
    {
        if (node->link) freeClusterLink(node->link);
        memcpy(node->ip,g->ip,NET_IP_STR_LEN);
        node->port = ntohs(g->port);
        node->pport = ntohs(g->pport);
        node->cport = ntohs(g->cport);
        node->flags &= ~CLUSTER_NODE_NOADDR;
    }
} else {
    /* If it's not in NOADDR state and we don't have it, we
     * add it to our trusted dict with exact nodeid and flag.
     * Note that we cannot simply start a handshake against
     * this IP/PORT pairs, since IP/PORT can be reused already,
     * otherwise we risk joining another cluster.
     *
     * Note that we require that the sender of this gossip message
     * is a well known node in our cluster, otherwise we risk
     * joining another cluster. */
    if (sender &&
        !(flags & CLUSTER_NODE_NOADDR) &&
        !clusterBlacklistExists(g->nodename))
    {
        clusterNode *node;
        node = createClusterNode(g->nodename, flags);
        memcpy(node->ip,g->ip,NET_IP_STR_LEN);
        node->port = ntohs(g->port);
        node->pport = ntohs(g->pport);
        node->cport = ntohs(g->cport);
        clusterAddNode(node);
    }
}

/* Next node */
g++;
}
}

/* IP -> string conversion. 'buf' is supposed to at least be 46 bytes.
 * If 'announced_ip' length is non-zero, it is used instead of extracting
 * the IP from the socket peer address. */
void nodeIp2String(char *buf, clusterLink *link, char *announced_ip) {
    if (announced_ip[0] != '\0') {

```

```

        memcpy(buf,announced_ip,NET_IP_STR_LEN);
        buf[NET_IP_STR_LEN-1] = '\\0'; /* We are not sure the input is sane. */
    } else {
        connPeerToString(link->conn, buf, NET_IP_STR_LEN, NULL);
    }
}

/* Update the node address to the IP address that can be extracted
 * from link->fd, or if hdr->myip is non empty, to the address the node
 * is announcing us. The port is taken from the packet header as well.
 *
 * If the address or port changed, disconnect the node link so that we'll
 * connect again to the new address.
 *
 * If the ip/port pair are already correct no operation is performed at
 * all.
 *
 * The function returns 0 if the node address is still the same,
 * otherwise 1 is returned. */
int nodeUpdateAddressIfNeeded(clusterNode *node, clusterLink *link,
                             clusterMsg *hdr)
{
    char ip[NET_IP_STR_LEN] = {0};
    int port = ntohs(hdr->port);
    int pport = ntohs(hdr->pport);
    int cport = ntohs(hdr->cport);

    /* We don't proceed if the link is the same as the sender link, as this
     * function is designed to see if the node link is consistent with the
     * symmetric link that is used to receive PINGs from the node.
     *
     * As a side effect this function never frees the passed 'link', so
     * it is safe to call during packet processing. */
    if (link == node->link) return 0;

    nodeIp2String(ip,link,hdr->myip);
    if (node->port == port && node->cport == cport && node->pport == pport &&
        strcmp(ip,node->ip) == 0) return 0;

    /* IP / port is different, update it. */
    memcpy(node->ip,ip,sizeof(ip));
    node->port = port;
    node->pport = pport;
    node->cport = cport;
    if (node->link) freeClusterLink(node->link);
    node->flags &= ~CLUSTER_NODE_NOADDR;
    serverLog(LL_WARNING,"Address updated for node %.40s, now %s:%d",
              node->name, node->ip, node->port);

    /* Check if this is our master and we have to change the
     * replication target as well. */

```

```

    if (nodeIsSlave(myself) && myself->slaveof == node)
        replicationSetMaster(node->ip, node->port);
    return 1;
}

/* Reconfigure the specified node 'n' as a master. This function is called when
 * a node that we believed to be a slave is now acting as master in order to
 * update the state of the node. */
void clusterSetNodeAsMaster(clusterNode *n) {
    if (nodeIsMaster(n)) return;

    if (n->slaveof) {
        clusterNodeRemoveSlave(n->slaveof,n);
        if (n != myself) n->flags |= CLUSTER_NODE_MIGRATE_TO;
    }
    n->flags &= ~CLUSTER_NODE_SLAVE;
    n->flags |= CLUSTER_NODE_MASTER;
    n->slaveof = NULL;

    /* Update config and state. */
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                        CLUSTER_TODO_UPDATE_STATE);
}

/* This function is called when we receive a master configuration via a
 * PING, PONG or UPDATE packet. What we receive is a node, a configEpoch of the
 * node, and the set of slots claimed under this configEpoch.
 *
 * What we do is to rebind the slots with newer configuration compared to our
 * local configuration, and if needed, we turn ourself into a replica of the
 * node (see the function comments for more info).
 *
 * The 'sender' is the node for which we received a configuration update.
 * Sometimes it is not actually the "Sender" of the information, like in the
 * case we receive the info via an UPDATE packet. */
void clusterUpdateSlotsConfigWith(clusterNode *sender, uint64_t
senderConfigEpoch, unsigned char *slots) {
    int j;
    clusterNode *curmaster = NULL, *newmaster = NULL;
    /* The dirty slots list is a list of slots for which we lose the ownership
     * while having still keys inside. This usually happens after a failover
     * or after a manual cluster reconfiguration operated by the admin.
     *
     * If the update message is not able to demote a master to slave (in this
     * case we'll resync with the master updating the whole key space), we
     * need to delete all the keys in the slots we lost ownership. */
    uint16_t dirty_slots[CLUSTER_SLOTS];
    int dirty_slots_count = 0;

    /* We should detect if sender is new master of our shard.
     * We will know it if all our slots were migrated to sender, and sender

```

```

    * has no slots except ours */
int sender_slots = 0;
int migrated_our_slots = 0;

/* Here we set curmaster to this node or the node this node
 * replicates to if it's a slave. In the for loop we are
 * interested to check if slots are taken away from curmaster. */
curmaster = nodeIsMaster(myself) ? myself : myself->slaveof;

if (sender == myself) {
    serverLog(LL_WARNING, "Discarding UPDATE message about myself.");
    return;
}

for (j = 0; j < CLUSTER_SLOTS; j++) {
    if (bitmapTestBit(slots, j)) {
        sender_slots++;

        /* The slot is already bound to the sender of this message. */
        if (server.cluster->slots[j] == sender) continue;

        /* The slot is in importing state, it should be modified only
         * manually via redis-cli (example: a resharding is in progress
         * and the migrating side slot was already closed and is
advertising
         * a new config. We still want the slot to be closed manually). */
        if (server.cluster->importing_slots_from[j]) continue;

        /* We rebind the slot to the new node claiming it if:
         * 1) The slot was unassigned or the new node claims it with a
         *    greater configEpoch.
         * 2) We are not currently importing the slot. */
        if (server.cluster->slots[j] == NULL ||
            server.cluster->slots[j]->configEpoch < senderConfigEpoch)
        {
            /* Was this slot mine, and still contains keys? Mark it as
             * a dirty slot. */
            if (server.cluster->slots[j] == myself &&
                countKeysInSlot(j) &&
                sender != myself)
            {
                dirty_slots[dirty_slots_count] = j;
                dirty_slots_count++;
            }

            if (server.cluster->slots[j] == curmaster) {
                newmaster = sender;
                migrated_our_slots++;
            }
            clusterDelSlot(j);
            clusterAddSlot(sender, j);
        }
    }
}

```

```

        clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                             CLUSTER_TODO_UPDATE_STATE|
                             CLUSTER_TODO_FSYNC_CONFIG);
    }
}

/* After updating the slots configuration, don't do any actual change
 * in the state of the server if a module disabled Redis Cluster
 * keys redirections. */
if (server.cluster_module_flags & CLUSTER_MODULE_FLAG_NO_REDIRECTION)
    return;

/* If at least one slot was reassigned from a node to another node
 * with a greater configEpoch, it is possible that:
 * 1) We are a master left without slots. This means that we were
 *    failed over and we should turn into a replica of the new
 *    master.
 * 2) We are a slave and our master is left without slots. We need
 *    to replicate to the new slots owner. */
if (newmaster && curmaster->numslots == 0 &&
    (server.cluster_allow_replica_migration ||
     sender_slots == migrated_our_slots)) {
    serverLog(LL_WARNING,
              "Configuration change detected. Reconfiguring myself "
              "as a replica of %.40s", sender->name);
    clusterSetMaster(sender);
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                         CLUSTER_TODO_UPDATE_STATE|
                         CLUSTER_TODO_FSYNC_CONFIG);
} else if (myself->slaveof && myself->slaveof->slaveof) {
    /* Safeguard against sub-replicas. A replica's master can turn itself
     * into a replica if its last slot is removed. If no other node takes
     * over the slot, there is nothing else to trigger replica migration.
    */
    serverLog(LL_WARNING,
              "I'm a sub-replica! Reconfiguring myself as a replica of
grandmaster %.40s",
              myself->slaveof->slaveof->name);
    clusterSetMaster(myself->slaveof->slaveof);
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                         CLUSTER_TODO_UPDATE_STATE|
                         CLUSTER_TODO_FSYNC_CONFIG);
} else if (dirty_slots_count) {
    /* If we are here, we received an update message which removed
     * ownership for certain slots we still have keys about, but still
     * we are serving some slots, so this master node was not demoted to
     * a slave.
     *
     * In order to maintain a consistent state between keys and slots
     * we need to remove all the keys from the slots we lost. */

```

```

        for (j = 0; j < dirty_slots_count; j++)
            delKeysInSlot(dirty_slots[j]);
    }
}

/* Cluster ping extensions.
 *
 * The ping/pong/meet messages support arbitrary extensions to add additional
 * metadata to the messages that are sent between the various nodes in the
 * cluster. The extensions take the form:
 * [ Header length + type (8 bytes) ]
 * [ Extension information (Arbitrary length, but must be 8 byte padded) ]
 */

/* Returns the length of a given extension */
static uint32_t getPingExtLength(clusterMsgPingExt *ext) {
    return ntohl(ext->length);
}

/* Returns the initial position of ping extensions. May return an invalid
 * address if there are no ping extensions. */
static clusterMsgPingExt *getInitialPingExt(clusterMsg *hdr, uint16_t count) {
    clusterMsgPingExt *initial = (clusterMsgPingExt*) &(hdr->data.ping.gossip[count]);
    return initial;
}

/* Given a current ping extension, returns the start of the next extension. May
 * return
 * an invalid address if there are no further ping extensions. */
static clusterMsgPingExt *getNextPingExt(clusterMsgPingExt *ext) {
    clusterMsgPingExt *next = (clusterMsgPingExt *) (((char *) ext) +
    getPingExtLength(ext));
    return next;
}

/* Returns the exact size needed to store the hostname. The returned value
 * will be 8 byte padded. */
int getHostnamePingExtSize() {
    /* If hostname is not set, we don't send this extension */
    if (sdslen(myself->hostname) == 0) return 0;

    int totlen = sizeof(clusterMsgPingExt) + EIGHT_BYTE_ALIGN(sdslen(myself->hostname) + 1);
    return totlen;
}

/* Write the hostname ping extension at the start of the cursor. This function
 * will update the cursor to point to the end of the written extension and
 * will return the amount of bytes written. */

```



```

int writeHostnamePingExt(clusterMsgPingExt **cursor) {
    /* If hostname is not set, we don't send this extension */
    if (sdslen(myself->hostname) == 0) return 0;

    /* Add the hostname information at the extension cursor */
    clusterMsgPingExtHostname *ext = &(*cursor)->ext[0].hostname;
    memcpy(ext->hostname, myself->hostname, sdslen(myself->hostname));
    uint32_t extension_size = getHostnamePingExtSize();

    /* Move the write cursor */
    (*cursor)->type = CLUSTERMSG_EXT_TYPE_HOSTNAME;
    (*cursor)->length = htonl(extension_size);
    /* Make sure the string is NULL terminated by adding 1 */
    *cursor = (clusterMsgPingExt *) (ext->hostname +
    EIGHT_BYTE_ALIGN(sdslen(myself->hostname) + 1));
    return extension_size;
}

/* We previously validated the extensions, so this function just needs to
 * handle the extensions. */
void clusterProcessPingExtensions(clusterMsg *hdr, clusterLink *link) {
    clusterNode *sender = link->node ? link->node : clusterLookupNode(hdr->
    >sender, CLUSTER_NAMELEN);
    char *ext_hostname = NULL;
    uint16_t extensions = ntohs(hdr->extensions);
    /* Loop through all the extensions and process them */
    clusterMsgPingExt *ext = getInitialPingExt(hdr, ntohs(hdr->count));
    while (extensions--) {
        uint16_t type = ntohs(ext->type);
        if (type == CLUSTERMSG_EXT_TYPE_HOSTNAME) {
            clusterMsgPingExtHostname *hostname_ext =
            (clusterMsgPingExtHostname *) &(ext->ext[0].hostname);
            ext_hostname = hostname_ext->hostname;
        } else {
            /* Unknown type, we will ignore it but log what happened. */
            serverLog(LL_WARNING, "Received unknown extension type %d", type);
        }

        /* We know this will be valid since we validated it ahead of time */
        ext = getNextPingExt(ext);
    }
    /* If the node did not send us a hostname extension, assume
     * they don't have an announced hostname. Otherwise, we'll
     * set it now. */
    updateAnnouncedHostname(sender, ext_hostname);
}

static clusterNode *getNodeFromLinkAndMsg(clusterLink *link, clusterMsg *hdr) {
    clusterNode *sender;
    if (link->node && !nodeInHandshake(link->node)) {
        /* If the link has an associated node, use that so that we don't have

```

```

to look it
    * up every time, except when the node is still in handshake, the node
still has
    * a random name thus not truly "known". */
    sender = link->node;
} else {
    /* Otherwise, fetch sender based on the message */
    sender = clusterLookupNode(hdr->sender, CLUSTER_NAMELEN);
    /* We know the sender node but haven't associate it with the link. This
must
    * be an inbound link because only for inbound links we didn't know
which node
    * to associate when they were created. */
    if (sender && !link->node) {
        setClusterNodeToInboundClusterLink(sender, link);
    }
}
return sender;
}

/* When this function is called, there is a packet to process starting
* at link->rcvbuf. Releasing the buffer is up to the caller, so this
* function should just handle the higher level stuff of processing the
* packet, modifying the cluster state if needed.
*
* The function returns 1 if the link is still valid after the packet
* was processed, otherwise 0 if the link was freed since the packet
* processing lead to some inconsistency error (for instance a PONG
* received from the wrong sender ID). */
int clusterProcessPacket(clusterLink *link) {
    clusterMsg *hdr = (clusterMsg*) link->rcvbuf;
    uint32_t totlen = ntohl(hdr->totlen);
    uint16_t type = ntohs(hdr->type);
    mstime_t now = mstime();

    if (type < CLUSTERMSG_TYPE_COUNT)
        server.cluster->stats_bus_messages_received[type]++;
    serverLog(LL_DEBUG, "---- Processing packet of type %s, %lu bytes",
        clusterGetMessageTypeString(type), (unsigned long) totlen);

    /* Perform sanity checks */
    if (totlen < 16) return 1; /* At least signature, version, totlen, count.
*/
    if (totlen > link->rcvbuf_len) return 1;

    if (ntohs(hdr->ver) != CLUSTER_PROTO_VER) {
        /* Can't handle messages of different versions. */
        return 1;
    }

    if (type == server.cluster_drop_packet_filter) {

```

```

        serverLog(LL_WARNING, "Dropping packet that matches debug drop
filter");
        return 1;
    }

    uint16_t flags = ntohs(hdr->flags);
    uint16_t extensions = ntohs(hdr->extensions);
    uint64_t senderCurrentEpoch = 0, senderConfigEpoch = 0;
    uint32_t explen; /* expected length of this packet */
    clusterNode *sender;

    if (type == CLUSTERMSG_TYPE_PING || type == CLUSTERMSG_TYPE_PONG ||
        type == CLUSTERMSG_TYPE_MEET)
    {
        uint16_t count = ntohs(hdr->count);

        explen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
        explen += (sizeof(clusterMsgDataGossip)*count);

        /* If there is extension data, which doesn't have a fixed length,
         * loop through them and validate the length of it now. */
        if (hdr->mflags[0] & CLUSTERMSG_FLAG0_EXT_DATA) {
            clusterMsgPingExt *ext = getInitialPingExt(hdr, count);
            while (extensions--) {
                uint16_t extlen = getPingExtLength(ext);
                if (extlen % 8 != 0) {
                    serverLog(LL_WARNING, "Received a %s packet without proper
padding (%d bytes)",
                        clusterGetMessageTypeString(type), (int) extlen);
                    return 1;
                }
                if ((totlen - explen) < extlen) {
                    serverLog(LL_WARNING, "Received invalid %s packet with
extension data that exceeds "
                        "total packet length (%lld)",
clusterGetMessageTypeString(type),
                        (unsigned long long) totlen);
                    return 1;
                }
                explen += extlen;
                ext = getNextPingExt(ext);
            }
        }
    }
    else if (type == CLUSTERMSG_TYPE_FAIL) {
        explen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
        explen += sizeof(clusterMsgDataFail);
    }
    else if (type == CLUSTERMSG_TYPE_PUBLISH || type ==
CLUSTERMSG_TYPE_PUBLISHSHARD) {
        explen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
        explen += sizeof(clusterMsgDataPublish) -
            8 +

```

```

        ntohs(hdr->data.publish.msg.channel_len) +
        ntohs(hdr->data.publish.msg.message_len);
} else if (type == CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST ||
        type == CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK ||
        type == CLUSTERMSG_TYPE_MFSTART)
{
    explen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
} else if (type == CLUSTERMSG_TYPE_UPDATE) {
    explen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
    explen += sizeof(clusterMsgDataUpdate);
} else if (type == CLUSTERMSG_TYPE_MODULE) {
    explen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
    explen += sizeof(clusterMsgModule) -
        3 + ntohs(hdr->data.module.msg.len);
} else {
    /* We don't know this type of packet, so we assume it's well formed. */
    explen = totlen;
}

if (totlen != explen) {
    serverLog(LL_WARNING, "Received invalid %s packet of length %lld but
expected length %lld",
        clusterGetMessageTypeString(type), (unsigned long long) totlen,
(unsigned long long) explen);
    return 1;
}

sender = getNodeFromLinkAndMsg(link, hdr);

/* Update the last time we saw any data from this node. We
 * use this in order to avoid detecting a timeout from a node that
 * is just sending a lot of data in the cluster bus, for instance
 * because of Pub/Sub. */
if (sender) sender->data_received = now;

if (sender && !nodeInHandshake(sender)) {
    /* Update our currentEpoch if we see a newer epoch in the cluster. */
    senderCurrentEpoch = ntohu64(hdr->currentEpoch);
    senderConfigEpoch = ntohu64(hdr->configEpoch);
    if (senderCurrentEpoch > server.cluster->currentEpoch)
        server.cluster->currentEpoch = senderCurrentEpoch;
    /* Update the sender configEpoch if it is publishing a newer one. */
    if (senderConfigEpoch > sender->configEpoch) {
        sender->configEpoch = senderConfigEpoch;
        clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
            CLUSTER_TODO_FSYNC_CONFIG);
    }
    /* Update the replication offset info for this node. */
    sender->repl_offset = ntohu64(hdr->offset);
    sender->repl_offset_time = now;
    /* If we are a slave performing a manual failover and our master

```

```

    * sent its offset while already paused, populate the MF state. */
if (server.cluster->mf_end &&
    nodeIsSlave(myself) &&
    myself->slaveof == sender &&
    hdr->mflags[0] & CLUSTERMSG_FLAG0_PAUSED &&
    server.cluster->mf_master_offset == -1)
{
    server.cluster->mf_master_offset = sender->repl_offset;
    clusterDoBeforeSleep(CLUSTER_TODO_HANDLE_MANUALFAILOVER);
    serverLog(LL_WARNING,
        "Received replication offset for paused "
        "master manual failover: %lld",
        server.cluster->mf_master_offset);
}
}

/* Initial processing of PING and MEET requests replying with a PONG. */
if (type == CLUSTERMSG_TYPE_PING || type == CLUSTERMSG_TYPE_MEET) {
    /* We use incoming MEET messages in order to set the address
     * for 'myself', since only other cluster nodes will send us
     * MEET messages on handshakes, when the cluster joins, or
     * later if we changed address, and those nodes will use our
     * official address to connect to us. So by obtaining this address
     * from the socket is a simple way to discover / update our own
     * address in the cluster without it being hardcoded in the config.
     *
     * However if we don't have an address at all, we update the address
     * even with a normal PING packet. If it's wrong it will be fixed
     * by MEET later. */
    if ((type == CLUSTERMSG_TYPE_MEET || myself->ip[0] == '\0') &&
        server.cluster_announce_ip == NULL)
    {
        char ip[NET_IP_STR_LEN];

        if (connSockName(link->conn, ip, sizeof(ip), NULL) != -1 &&
            strcmp(ip, myself->ip))
        {
            memcpy(myself->ip, ip, NET_IP_STR_LEN);
            serverLog(LL_WARNING, "IP address for this node updated to %s",
                myself->ip);
            clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG);
        }
    }

    /* Add this node if it is new for us and the msg type is MEET.
     * In this stage we don't try to add the node with the right
     * flags, slaveof pointer, and so forth, as this details will be
     * resolved when we'll receive PONGs from the node. */
    if (!sender && type == CLUSTERMSG_TYPE_MEET) {
        clusterNode *node;

```

```

        node = createClusterNode(NULL, CLUSTER_NODE_HANDSHAKE);
        nodeIp2String(node->ip, link, hdr->myip);
        node->port = ntohs(hdr->port);
        node->pport = ntohs(hdr->pport);
        node->cport = ntohs(hdr->cport);
        clusterAddNode(node);
        clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG);
    }

    /* If this is a MEET packet from an unknown node, we still process
     * the gossip section here since we have to trust the sender because
     * of the message type. */
    if (!sender && type == CLUSTERMSG_TYPE_MEET)
        clusterProcessGossipSection(hdr, link);

    /* Anyway reply with a PONG */
    clusterSendPing(link, CLUSTERMSG_TYPE_PONG);
}

/* PING, PONG, MEET: process config information. */
if (type == CLUSTERMSG_TYPE_PING || type == CLUSTERMSG_TYPE_PONG ||
    type == CLUSTERMSG_TYPE_MEET)
{
    serverLog(LL_DEBUG, "%s packet received: %.40s",
        clusterGetMessageTypeString(type),
        link->node ? link->node->name : "NULL");
    if (!link->inbound) {
        if (nodeInHandshake(link->node)) {
            /* If we already have this node, try to change the
             * IP/port of the node with the new one. */
            if (sender) {
                serverLog(LL_VERBOSE,
                    "Handshake: we already know node %.40s, "
                    "updating the address if needed.", sender->name);
                if (nodeUpdateAddressIfNeeded(sender, link, hdr))
                {
                    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG |
                                         CLUSTER_TODO_UPDATE_STATE);
                }
                /* Free this node as we already have it. This will
                 * cause the link to be freed as well. */
                clusterDelNode(link->node);
                return 0;
            }
        }

        /* First thing to do is replacing the random name with the
         * right node name if this was a handshake stage. */
        clusterRenameNode(link->node, hdr->sender);
        serverLog(LL_DEBUG, "Handshake with node %.40s completed.",
            link->node->name);
        link->node->flags &= ~CLUSTER_NODE_HANDSHAKE;
    }
}

```

```

        link->node->flags |= flags &
(CLUSTER_NODE_MASTER|CLUSTER_NODE_SLAVE);
        clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG);
    } else if (memcmp(link->node->name,hdr->sender,
        CLUSTER_NAMELEN) != 0)
    {
        /* If the reply has a non matching node ID we
         * disconnect this node and set it as not having an associated
         * address. */
        serverLog(LL_DEBUG,"PONG contains mismatching sender ID. About
node %.40s added %d ms ago, having flags %d",
            link->node->name,
            (int)(now-(link->node->ctime)),
            link->node->flags);
        link->node->flags |= CLUSTER_NODE_NOADDR;
        link->node->ip[0] = '\0';
        link->node->port = 0;
        link->node->pport = 0;
        link->node->cport = 0;
        freeClusterLink(link);
        clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG);
        return 0;
    }
}

/* Copy the CLUSTER_NODE_NOFAILOVER flag from what the sender
 * announced. This is a dynamic flag that we receive from the
 * sender, and the latest status must be trusted. We need it to
 * be propagated because the slave ranking used to understand the
 * delay of each slave in the voting process, needs to know
 * what are the instances really competing. */
if (sender) {
    int nofailover = flags & CLUSTER_NODE_NOFAILOVER;
    sender->flags &= ~CLUSTER_NODE_NOFAILOVER;
    sender->flags |= nofailover;
}

/* Update the node address if it changed. */
if (sender && type == CLUSTERMSG_TYPE_PING &&
    !nodeInHandshake(sender) &&
    nodeUpdateAddressIfNeeded(sender,link,hdr))
{
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
        CLUSTER_TODO_UPDATE_STATE);
}

/* Update our info about the node */
if (!link->inbound && type == CLUSTERMSG_TYPE_PONG) {
    link->node->pong_received = now;
    link->node->ping_sent = 0;
}

```

```

/* The PFAIL condition can be reversed without external
 * help if it is momentary (that is, if it does not
 * turn into a FAIL state).
 *
 * The FAIL condition is also reversible under specific
 * conditions detected by clearNodeFailureIfNeeded(). */
if (nodeTimedOut(link->node)) {
    link->node->flags &= ~CLUSTER_NODE_PFAIL;
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                        CLUSTER_TODO_UPDATE_STATE);
} else if (nodeFailed(link->node)) {
    clearNodeFailureIfNeeded(link->node);
}
}

/* Check for role switch: slave -> master or master -> slave. */
if (sender) {
    if (!memcmp(hdr->slaveof, CLUSTER_NODE_NULL_NAME,
                sizeof(hdr->slaveof)))
    {
        /* Node is a master. */
        clusterSetNodeAsMaster(sender);
    } else {
        /* Node is a slave. */
        clusterNode *master = clusterLookupNode(hdr->slaveof,
        CLUSTER_NAMELEN);

        if (nodeIsMaster(sender)) {
            /* Master turned into a slave! Reconfigure the node. */
            clusterDelNodeSlots(sender);
            sender->flags &= ~(CLUSTER_NODE_MASTER|
                            CLUSTER_NODE_MIGRATE_TO);
            sender->flags |= CLUSTER_NODE_SLAVE;

            /* Update config and state. */
            clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                                CLUSTER_TODO_UPDATE_STATE);
        }

        /* Master node changed for this slave? */
        if (master && sender->slaveof != master) {
            if (sender->slaveof)
                clusterNodeRemoveSlave(sender->slaveof, sender);
            clusterNodeAddSlave(master, sender);
            sender->slaveof = master;

            /* Update config. */
            clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG);
        }
    }
}
}

```



```

/* Update our info about served slots.
 *
 * Note: this MUST happen after we update the master/slave state
 * so that CLUSTER_NODE_MASTER flag will be set. */

/* Many checks are only needed if the set of served slots this
 * instance claims is different compared to the set of slots we have
 * for it. Check this ASAP to avoid other computational expensive
 * checks later. */
clusterNode *sender_master = NULL; /* Sender or its master if slave. */
int dirty_slots = 0; /* Sender claimed slots don't match my view? */

if (sender) {
    sender_master = nodeIsMaster(sender) ? sender : sender->slaveof;
    if (sender_master) {
        dirty_slots = memcmp(sender_master->slots,
                             hdr->myslots, sizeof(hdr->myslots)) != 0;
    }
}

/* 1) If the sender of the message is a master, and we detected that
 * the set of slots it claims changed, scan the slots to see if we
 * need to update our configuration. */
if (sender && nodeIsMaster(sender) && dirty_slots)
    clusterUpdateSlotsConfigWith(sender, senderConfigEpoch, hdr->myslots);

/* 2) We also check for the reverse condition, that is, the sender
 * claims to serve slots we know are served by a master with a
 * greater configEpoch. If this happens we inform the sender.
 *
 * This is useful because sometimes after a partition heals, a
 * reappearing master may be the last one to claim a given set of
 * hash slots, but with a configuration that other instances know to
 * be deprecated. Example:
 *
 * A and B are master and slave for slots 1,2,3.
 * A is partitioned away, B gets promoted.
 * B is partitioned away, and A returns available.
 *
 * Usually B would PING A publishing its set of served slots and its
 * configEpoch, but because of the partition B can't inform A of the
 * new configuration, so other nodes that have an updated table must
 * do it. In this way A will stop to act as a master (or can try to
 * failover if there are the conditions to win the election). */
if (sender && dirty_slots) {
    int j;

    for (j = 0; j < CLUSTER_SLOTS; j++) {
        if (bitmapTestBit(hdr->myslots, j)) {

```

```

        if (server.cluster->slots[j] == sender ||
            server.cluster->slots[j] == NULL) continue;
        if (server.cluster->slots[j]->configEpoch >
            senderConfigEpoch)
        {
            serverLog(LL_VERBOSE,
                "Node %.40s has old slots configuration, sending "
                "an UPDATE message about %.40s",
                sender->name, server.cluster->slots[j]->name);
            clusterSendUpdate(sender->link,
                server.cluster->slots[j]);

            /* TODO: instead of exiting the loop send every other
             * UPDATE packet for other nodes that are the new owner
             * of sender's slots. */
            break;
        }
    }
}

/* If our config epoch collides with the sender's try to fix
 * the problem. */
if (sender &&
    nodeIsMaster(myself) && nodeIsMaster(sender) &&
    senderConfigEpoch == myself->configEpoch)
{
    clusterHandleConfigEpochCollision(sender);
}

/* Get info from the gossip section */
if (sender) {
    clusterProcessGossipSection(hdr, link);
    clusterProcessPingExtensions(hdr, link);
}
} else if (type == CLUSTERMSG_TYPE_FAIL) {
    clusterNode *failing;

    if (sender) {
        failing = clusterLookupNode(hdr->data.fail.about.nodename,
            CLUSTER_NAMELEN);
        if (failing &&
            !((failing->flags & (CLUSTER_NODE_FAIL|CLUSTER_NODE_MYSELF)))
        {
            serverLog(LL_NOTICE,
                "FAIL message received from %.40s about %.40s",
                hdr->sender, hdr->data.fail.about.nodename);
            failing->flags |= CLUSTER_NODE_FAIL;
            failing->fail_time = now;
            failing->flags &= ~CLUSTER_NODE_PFAIL;
            clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|

```

```

                                CLUSTER_TODO_UPDATE_STATE);
    }
} else {
    serverLog(LL_NOTICE,
        "Ignoring FAIL message from unknown node %.40s about %.40s",
        hdr->sender, hdr->data.fail.about.nodename);
}
} else if (type == CLUSTERMSG_TYPE_PUBLISH || type ==
CLUSTERMSG_TYPE_PUBLISHSHARD) {
    if (!sender) return 1; /* We don't know that node. */

    robj *channel, *message;
    uint32_t channel_len, message_len;

    /* Don't bother creating useless objects if there are no
     * Pub/Sub subscribers. */
    if ((type == CLUSTERMSG_TYPE_PUBLISH
        && serverPubsubSubscriptionCount() > 0)
        || (type == CLUSTERMSG_TYPE_PUBLISHSHARD
            && serverPubsubShardSubscriptionCount() > 0))
    {
        channel_len = ntohl(hdr->data.publish.msg.channel_len);
        message_len = ntohl(hdr->data.publish.msg.message_len);
        channel = createStringObject(
            (char*)hdr->data.publish.msg.bulk_data, channel_len);
        message = createStringObject(
            (char*)hdr->data.publish.msg.bulk_data+channel_len,
            message_len);
        pubsubPublishMessage(channel, message, type ==
CLUSTERMSG_TYPE_PUBLISHSHARD);
        decrRefCount(channel);
        decrRefCount(message);
    }
} else if (type == CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST) {
    if (!sender) return 1; /* We don't know that node. */
    clusterSendFailoverAuthIfNeeded(sender, hdr);
} else if (type == CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK) {
    if (!sender) return 1; /* We don't know that node. */
    /* We consider this vote only if the sender is a master serving
     * a non zero number of slots, and its currentEpoch is greater or
     * equal to epoch where this node started the election. */
    if (nodeIsMaster(sender) && sender->numslots > 0 &&
        senderCurrentEpoch >= server.cluster->failover_auth_epoch)
    {
        server.cluster->failover_auth_count++;
        /* Maybe we reached a quorum here, set a flag to make sure
         * we check ASAP. */
        clusterDoBeforeSleep(CLUSTER_TODO_HANDLE_FAILOVER);
    }
} else if (type == CLUSTERMSG_TYPE_MFSTART) {
    /* This message is acceptable only if I'm a master and the sender

```

```

    * is one of my slaves. */
    if (!sender || sender->slaveof != myself) return 1;
    /* Manual failover requested from slaves. Initialize the state
    * accordingly. */
    resetManualFailover();
    server.cluster->mf_end = now + CLUSTER_MF_TIMEOUT;
    server.cluster->mf_slave = sender;
    pauseClients(PAUSE_DURING_FAILOVER,
                 now + (CLUSTER_MF_TIMEOUT * CLUSTER_MF_PAUSE_MULT),
                 CLIENT_PAUSE_WRITE);
    serverLog(LL_WARNING, "Manual failover requested by replica %.40s.",
              sender->name);
    /* We need to send a ping message to the replica, as it would carry
    * `server.cluster->mf_master_offset`, which means the master paused
clients
    * at offset `server.cluster->mf_master_offset`, so that the replica
would
    * know that it is safe to set its `server.cluster->mf_can_start` to 1
so as
    * to complete failover as quickly as possible. */
    clusterSendPing(link, CLUSTERMSG_TYPE_PING);
} else if (type == CLUSTERMSG_TYPE_UPDATE) {
    clusterNode *n; /* The node the update is about. */
    uint64_t reportedConfigEpoch =
        ntohu64(hdr->data.update.nodcfg.configEpoch);

    if (!sender) return 1; /* We don't know the sender. */
    n = clusterLookupNode(hdr->data.update.nodcfg.nodename,
CLUSTER_NAMELEN);
    if (!n) return 1; /* We don't know the reported node. */
    if (n->configEpoch >= reportedConfigEpoch) return 1; /* Nothing new. */

    /* If in our current config the node is a slave, set it as a master. */
    if (nodeIsSlave(n)) clusterSetNodeAsMaster(n);

    /* Update the node's configEpoch. */
    n->configEpoch = reportedConfigEpoch;
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                        CLUSTER_TODO_FSYNC_CONFIG);

    /* Check the bitmap of served slots and update our
    * config accordingly. */
    clusterUpdateSlotsConfigWith(n, reportedConfigEpoch,
                                hdr->data.update.nodcfg.slots);
} else if (type == CLUSTERMSG_TYPE_MODULE) {
    if (!sender) return 1; /* Protect the module from unknown nodes. */
    /* We need to route this message back to the right module subscribed
    * for the right message type. */
    uint64_t module_id = hdr->data.module.msg.module_id; /* Endian-safe ID
*/
    uint32_t len = ntohl(hdr->data.module.msg.len);

```

```

        uint8_t type = hdr->data.module.msg.type;
        unsigned char *payload = hdr->data.module.msg.bulk_data;
        moduleCallClusterReceivers(sender->name,module_id,type,payload,len);
    } else {
        serverLog(LL_WARNING,"Received unknown packet type: %d", type);
    }
    return 1;
}

/* This function is called when we detect the link with this node is lost.
   We set the node as no longer connected. The Cluster Cron will detect
   this connection and will try to get it connected again.

   Instead if the node is a temporary node used to accept a query, we
   completely free the node on error. */
void handleLinkIOError(clusterLink *link) {
    freeClusterLink(link);
}

/* Send data. This is handled using a trivial send buffer that gets
   * consumed by write(). We don't try to optimize this for speed too much
   * as this is a very low traffic channel. */
void clusterWriteHandler(connection *conn) {
    clusterLink *link = connGetPrivateData(conn);
    ssize_t nwritten;

    nwritten = connWrite(conn, link->sndbuf, sdslen(link->sndbuf));
    if (nwritten <= 0) {
        serverLog(LL_DEBUG,"I/O error writing to node link: %s",
            (nwritten == -1) ? connGetLastError(conn) : "short write");
        handleLinkIOError(link);
        return;
    }
    sdsrange(link->sndbuf,nwritten,-1);
    if (sdslen(link->sndbuf) == 0)
        connSetWriteHandler(link->conn, NULL);
}

/* A connect handler that gets called when a connection to another node
   * gets established.
   */
void clusterLinkConnectHandler(connection *conn) {
    clusterLink *link = connGetPrivateData(conn);
    clusterNode *node = link->node;

    /* Check if connection succeeded */
    if (connGetState(conn) != CONN_STATE_CONNECTED) {
        serverLog(LL_VERBOSE, "Connection with Node %.40s at %s:%d failed: %s",
            node->name, node->ip, node->cport,
            connGetLastError(conn));
        freeClusterLink(link);
    }
}

```

```

        return;
    }

    /* Register a read handler from now on */
    connSetReadHandler(conn, clusterReadHandler);

    /* Queue a PING in the new connection ASAP: this is crucial
     * to avoid false positives in failure detection.
     *
     * If the node is flagged as MEET, we send a MEET message instead
     * of a PING one, to force the receiver to add us in its node
     * table. */
    mstime_t old_ping_sent = node->ping_sent;
    clusterSendPing(link, node->flags & CLUSTER_NODE_MEET ?
        CLUSTERMSG_TYPE_MEET : CLUSTERMSG_TYPE_PING);
    if (old_ping_sent) {
        /* If there was an active ping before the link was
         * disconnected, we want to restore the ping time, otherwise
         * replaced by the clusterSendPing() call. */
        node->ping_sent = old_ping_sent;
    }

    /* We can clear the flag after the first packet is sent.
     * If we'll never receive a PONG, we'll never send new packets
     * to this node. Instead after the PONG is received and we
     * are no longer in meet/handshake status, we want to send
     * normal PING packets. */
    node->flags &= ~CLUSTER_NODE_MEET;

    serverLog(LL_DEBUG, "Connecting with Node %.40s at %s:%d",
        node->name, node->ip, node->cport);
}

/* Read data. Try to read the first field of the header first to check the
 * full length of the packet. When a whole packet is in memory this function
 * will call the function to process the packet. And so forth. */
void clusterReadHandler(connection *conn) {
    clusterMsg buf[1];
    ssize_t nread;
    clusterMsg *hdr;
    clusterLink *link = connGetPrivateData(conn);
    unsigned int readlen, rcvbuflen;

    while(1) { /* Read as long as there is data to read. */
        rcvbuflen = link->rcvbuf_len;
        if (rcvbuflen < 8) {
            /* First, obtain the first 8 bytes to get the full message
             * length. */
            readlen = 8 - rcvbuflen;
        } else {
            /* Finally read the full message. */
            hdr = (clusterMsg*) link->rcvbuf;

```

```

    if (rcvbuflen == 8) {
        /* Perform some sanity check on the message signature
         * and length. */
        if (memcmp(hdr->sig,"RCmb",4) != 0 ||
            ntohl(hdr->totlen) < CLUSTERMSG_MIN_LEN)
        {
            serverLog(LL_WARNING,
                "Bad message length or signature received "
                "from Cluster bus.");
            handleLinkIOError(link);
            return;
        }
    }
    readlen = ntohl(hdr->totlen) - rcvbuflen;
    if (readlen > sizeof(buf)) readlen = sizeof(buf);
}

nread = connRead(conn,buf,readlen);
if (nread == -1 && (connGetState(conn) == CONN_STATE_CONNECTED))
return; /* No more data ready. */

if (nread <= 0) {
    /* I/O error... */
    serverLog(LL_DEBUG,"I/O error reading from node link: %s",
        (nread == 0) ? "connection closed" : connGetLastError(conn));
    handleLinkIOError(link);
    return;
} else {
    /* Read data and recast the pointer to the new buffer. */
    size_t unused = link->rcvbuf_alloc - link->rcvbuf_len;
    if ((size_t)nread > unused) {
        size_t required = link->rcvbuf_len + nread;
        /* If less than 1mb, grow to twice the needed size, if larger
        grow by 1mb. */
        link->rcvbuf_alloc = required < RCVBUF_MAX_PREALLOC ? required
        * 2: required + RCVBUF_MAX_PREALLOC;
        link->rcvbuf = zrealloc(link->rcvbuf, link->rcvbuf_alloc);
    }
    memcpy(link->rcvbuf + link->rcvbuf_len, buf, nread);
    link->rcvbuf_len += nread;
    hdr = (clusterMsg*) link->rcvbuf;
    rcvbuflen += nread;
}

/* Total length obtained? Process this packet. */
if (rcvbuflen >= 8 && rcvbuflen == ntohl(hdr->totlen)) {
    if (clusterProcessPacket(link)) {
        if (link->rcvbuf_alloc > RCVBUF_INIT_LEN) {
            zfree(link->rcvbuf);
            link->rcvbuf = zmalloc(link->rcvbuf_alloc =
RCVBUF_INIT_LEN);

```

```

        }
        link->rcvbuf_len = 0;
    } else {
        return; /* Link no longer valid. */
    }
}
}

/* Put stuff into the send buffer.
 *
 * It is guaranteed that this function will never have as a side effect
 * the link to be invalidated, so it is safe to call this function
 * from event handlers that will do stuff with the same link later. */
void clusterSendMessage(clusterLink *link, unsigned char *msg, size_t msglen) {
    if (sdslen(link->sndbuf) == 0 && msglen != 0)
        connSetWriteHandlerWithBarrier(link->conn, clusterWriteHandler, 1);

    link->sndbuf = sdscatlen(link->sndbuf, msg, msglen);

    /* Populate sent messages stats. */
    clusterMsg *hdr = (clusterMsg*) msg;
    uint16_t type = ntohs(hdr->type);
    if (type < CLUSTERMSG_TYPE_COUNT)
        server.cluster->stats_bus_messages_sent[type]++;
}

/* Send a message to all the nodes that are part of the cluster having
 * a connected link.
 *
 * It is guaranteed that this function will never have as a side effect
 * some node->link to be invalidated, so it is safe to call this function
 * from event handlers that will do stuff with node links later. */
void clusterBroadcastMessage(void *buf, size_t len) {
    dictIterator *di;
    dictEntry *de;

    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);

        if (!node->link) continue;
        if (node->flags & (CLUSTER_NODE_MYSELF|CLUSTER_NODE_HANDSHAKE))
            continue;
        clusterSendMessage(node->link, buf, len);
    }
    dictReleaseIterator(di);
}

/* Build the message header. hdr must point to a buffer at least
 * sizeof(clusterMsg) in bytes. */

```



```

void clusterBuildMessageHdr(clusterMsg *hdr, int type) {
    int totlen = 0;
    uint64_t offset;
    clusterNode *master;

    /* If this node is a master, we send its slots bitmap and configEpoch.
     * If this node is a slave we send the master's information instead (the
     * node is flagged as slave so the receiver knows that it is NOT really
     * in charge for this slots. */
    master = (nodeIsSlave(myself) && myself->slaveof) ?
        myself->slaveof : myself;

    memset(hdr,0,sizeof(*hdr));
    hdr->ver = htons(CLUSTER_PROTO_VER);
    hdr->sig[0] = 'R';
    hdr->sig[1] = 'C';
    hdr->sig[2] = 'm';
    hdr->sig[3] = 'b';
    hdr->type = htons(type);
    memcpy(hdr->sender,myself->name,CLUSTER_NAMELEN);

    /* If cluster-announce-ip option is enabled, force the receivers of our
     * packets to use the specified address for this node. Otherwise if the
     * first byte is zero, they'll do auto discovery. */
    memset(hdr->myip,0,NET_IP_STR_LEN);
    if (server.cluster_announce_ip) {
        strncpy(hdr->myip,server.cluster_announce_ip,NET_IP_STR_LEN-1);
        hdr->myip[NET_IP_STR_LEN-1] = '\0';
    }

    /* Handle cluster-announce-[tls-|bus-]port. */
    int announced_port, announced_pport, announced_cport;
    deriveAnnouncedPorts(&announced_port, &announced_pport, &announced_cport);

    memcpy(hdr->myslots,master->slots,sizeof(hdr->myslots));
    memset(hdr->slaveof,0,CLUSTER_NAMELEN);
    if (myself->slaveof != NULL)
        memcpy(hdr->slaveof,myself->slaveof->name, CLUSTER_NAMELEN);
    hdr->port = htons(announced_port);
    hdr->pport = htons(announced_pport);
    hdr->cport = htons(announced_cport);
    hdr->flags = htons(myself->flags);
    hdr->state = server.cluster->state;

    /* Set the currentEpoch and configEpochs. */
    hdr->currentEpoch = htonu64(server.cluster->currentEpoch);
    hdr->configEpoch = htonu64(master->configEpoch);

    /* Set the replication offset. */
    if (nodeIsSlave(myself))
        offset = replicationGetSlaveOffset();
}

```

```

else
    offset = server.master_repl_offset;
hdr->offset = htonu64(offset);

/* Set the message flags. */
if (nodeIsMaster(myself) && server.cluster->mf_end)
    hdr->mflags[0] |= CLUSTERMSG_FLAG0_PAUSED;

/* Compute the message length for certain messages. For other messages
 * this is up to the caller. */
if (type == CLUSTERMSG_TYPE_FAIL) {
    totlen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
    totlen += sizeof(clusterMsgDataFail);
} else if (type == CLUSTERMSG_TYPE_UPDATE) {
    totlen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
    totlen += sizeof(clusterMsgDataUpdate);
}
hdr->totlen = htonl(totlen);
/* For PING, PONG, MEET and other variable length messages fixing the
 * totlen field is up to the caller. */
}

/* Return non zero if the node is already present in the gossip section of the
 * message pointed by 'hdr' and having 'count' gossip entries. Otherwise
 * zero is returned. Helper for clusterSendPing(). */
int clusterNodeIsInGossipSection(clusterMsg *hdr, int count, clusterNode *n) {
    int j;
    for (j = 0; j < count; j++) {
        if (memcmp(hdr->data.ping.gossip[j].nodename,n->name,
            CLUSTER_NAMELEN) == 0) break;
    }
    return j != count;
}

/* Set the i-th entry of the gossip section in the message pointed by 'hdr'
 * to the info of the specified node 'n'. */
void clusterSetGossipEntry(clusterMsg *hdr, int i, clusterNode *n) {
    clusterMsgDataGossip *gossip;
    gossip = &(hdr->data.ping.gossip[i]);
    memcpy(gossip->nodename,n->name,CLUSTER_NAMELEN);
    gossip->ping_sent = htonl(n->ping_sent/1000);
    gossip->pong_received = htonl(n->pong_received/1000);
    memcpy(gossip->ip,n->ip,sizeof(n->ip));
    gossip->port = htons(n->port);
    gossip->cport = htons(n->cport);
    gossip->flags = htons(n->flags);
    gossip->pport = htons(n->pport);
    gossip->notused1 = 0;
}

/* Send a PING or PONG packet to the specified node, making sure to add enough

```

```

    * gossip information. */
void clusterSendPing(clusterLink *link, int type) {
    unsigned char *buf;
    clusterMsg *hdr;
    int gossipcount = 0; /* Number of gossip sections added so far. */
    int wanted; /* Number of gossip sections we want to append if possible. */
    int estlen; /* Upper bound on estimated packet length */
    /* freshnodes is the max number of nodes we can hope to append at all:
     * nodes available minus two (ourselves and the node we are sending the
     * message to). However practically there may be less valid nodes since
     * nodes in handshake state, disconnected, are not considered. */
    int freshnodes = dictSize(server.cluster->nodes)-2;

    /* How many gossip sections we want to add? 1/10 of the number of nodes
     * and anyway at least 3. Why 1/10?
     *
     * If we have N masters, with N/10 entries, and we consider that in
     * node_timeout we exchange with each other node at least 4 packets
     * (we ping in the worst case in node_timeout/2 time, and we also
     * receive two pings from the host), we have a total of 8 packets
     * in the node_timeout*2 failure reports validity time. So we have
     * that, for a single PFAIL node, we can expect to receive the following
     * number of failure reports (in the specified window of time):
     *
     * PROB * GOSSIP_ENTRIES_PER_PACKET * TOTAL_PACKETS:
     *
     * PROB = probability of being featured in a single gossip entry,
     *       which is 1 / NUM_OF_NODES.
     * ENTRIES = 10.
     * TOTAL_PACKETS = 2 * 4 * NUM_OF_MASTERS.
     *
     * If we assume we have just masters (so num of nodes and num of masters
     * is the same), with 1/10 we always get over the majority, and
specifically
     * 80% of the number of nodes, to account for many masters failing at the
     * same time.
     *
     * Since we have non-voting slaves that lower the probability of an entry
     * to feature our node, we set the number of entries per packet as
     * 10% of the total nodes we have. */
    wanted = floor(dictSize(server.cluster->nodes)/10);
    if (wanted < 3) wanted = 3;
    if (wanted > freshnodes) wanted = freshnodes;

    /* Include all the nodes in PFAIL state, so that failure reports are
     * faster to propagate to go from PFAIL to FAIL state. */
    int pfail_wanted = server.cluster->stats_pfail_nodes;

    /* Compute the maximum estlen to allocate our buffer. We'll fix the estlen
     * later according to the number of gossip sections we really were able
     * to put inside the packet. */

```

```

estlen = sizeof(clusterMsg) - sizeof(union clusterMsgData);
estlen += (sizeof(clusterMsgDataGossip)*(wanted + pfail_wanted));
estlen += sizeof(clusterMsgPingExt) + getHostnamePingExtSize();

/* Note: clusterBuildMessageHdr() expects the buffer to be always at least
 * sizeof(clusterMsg) or more. */
if (estlen < (int)sizeof(clusterMsg)) estlen = sizeof(clusterMsg);
buf = zcalloc(estlen);
hdr = (clusterMsg*) buf;

/* Populate the header. */
if (!link->inbound && type == CLUSTERMSG_TYPE_PING)
    link->node->ping_sent = mstime();
clusterBuildMessageHdr(hdr,type);

/* Populate the gossip fields */
int maxiterations = wanted*3;
while(freshnodes > 0 && gossipcount < wanted && maxiterations--> {
    dictEntry *de = dictGetRandomKey(server.cluster->nodes);
    clusterNode *this = dictGetVal(de);

    /* Don't include this node: the whole packet header is about us
     * already, so we just gossip about other nodes. */
    if (this == myself) continue;

    /* PFAIL nodes will be added later. */
    if (this->flags & CLUSTER_NODE_PFAIL) continue;

    /* In the gossip section don't include:
     * 1) Nodes in HANDSHAKE state.
     * 3) Nodes with the NOADDR flag set.
     * 4) Disconnected nodes if they don't have configured slots.
     */
    if (this->flags & (CLUSTER_NODE_HANDSHAKE|CLUSTER_NODE_NOADDR) ||
        (this->link == NULL && this->numslots == 0))
    {
        freshnodes--; /* Technically not correct, but saves CPU. */
        continue;
    }

    /* Do not add a node we already have. */
    if (clusterNodeIsInGossipSection(hdr,gossipcount,this)) continue;

    /* Add it */
    clusterSetGossipEntry(hdr,gossipcount,this);
    freshnodes--;
    gossipcount++;
}

/* If there are PFAIL nodes, add them at the end. */
if (pfail_wanted) {

```

```

dictIterator *di;
dictEntry *de;

di = dictGetSafeIterator(server.cluster->nodes);
while((de = dictNext(di)) != NULL && pfail_wanted > 0) {
    clusterNode *node = dictGetVal(de);
    if (node->flags & CLUSTER_NODE_HANDSHAKE) continue;
    if (node->flags & CLUSTER_NODE_NOADDR) continue;
    if (!(node->flags & CLUSTER_NODE_PFAIL)) continue;
    clusterSetGossipEntry(hdr,gossipcount,node);
    freshnodes--;
    gossipcount++;
    /* We take the count of the slots we allocated, since the
     * PFAIL stats may not match perfectly with the current number
     * of PFAIL nodes. */
    pfail_wanted--;
}
dictReleaseIterator(di);
}

```

```

int totlen = 0;
int extensions = 0;
/* Set the initial extension position */
clusterMsgPingExt *cursor = getInitialPingExt(hdr, gossipcount);
/* Add in the extensions */
if (sdslen(myself->hostname) != 0) {
    hdr->mflags[0] |= CLUSTERMSG_FLAG0_EXT_DATA;
    totlen += writeHostnamePingExt(&cursor);
    extensions++;
}

```

```

/* Compute the actual total length and send! */
totlen += sizeof(clusterMsg)-sizeof(union clusterMsgData);
totlen += (sizeof(clusterMsgDataGossip)*gossipcount);
hdr->count = htons(gossipcount);
hdr->extensions = htons(extensions);
hdr->totlen = htonl(totlen);
clusterSendMessage(link,buf,totlen);
zfree(buf);
}

```

```

/* Send a PONG packet to every connected node that's not in handshake state
 * and for which we have a valid link.

```

```

 *

```

```

 * In Redis Cluster pongs are not used just for failure detection, but also
 * to carry important configuration information. So broadcasting a pong is
 * useful when something changes in the configuration and we want to make
 * the cluster aware ASAP (for instance after a slave promotion).

```

```

 *

```

```

 * The 'target' argument specifies the receiving instances using the

```

```

* defines below:
*
* CLUSTER_BROADCAST_ALL -> All known instances.
* CLUSTER_BROADCAST_LOCAL_SLAVES -> All slaves in my master-slaves ring.
*/
#define CLUSTER_BROADCAST_ALL 0
#define CLUSTER_BROADCAST_LOCAL_SLAVES 1
void clusterBroadcastPong(int target) {
    dictIterator *di;
    dictEntry *de;

    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);

        if (!node->link) continue;
        if (node == myself || nodeInHandshake(node)) continue;
        if (target == CLUSTER_BROADCAST_LOCAL_SLAVES) {
            int local_slave =
                nodeIsSlave(node) && node->slaveof &&
                (node->slaveof == myself || node->slaveof == myself->slaveof);
            if (!local_slave) continue;
        }
        clusterSendPing(node->link, CLUSTERMSG_TYPE_PONG);
    }
    dictReleaseIterator(di);
}

/* Send a PUBLISH message.
*
* If link is NULL, then the message is broadcasted to the whole cluster.
*
* Sanitizer suppression: In clusterMsgDataPublish, sizeof(bulk_data) is 8.
* As all the struct is used as a buffer, when more than 8 bytes are copied
into
* the 'bulk_data', sanitizer generates an out-of-bounds error which is a false
* positive in this context. */
REDIS_NO_SANITIZE("bounds")
void clusterSendPublish(clusterLink *link, robj *channel, robj *message,
uint16_t type) {
    unsigned char *payload;
    clusterMsg buf[1];
    clusterMsg *hdr = (clusterMsg*) buf;
    uint32_t totlen;
    uint32_t channel_len, message_len;

    channel = getDecodedObject(channel);
    message = getDecodedObject(message);
    channel_len = sdslen(channel->ptr);
    message_len = sdslen(message->ptr);

```

```

clusterBuildMessageHdr(hdr,type);
totlen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
totlen += sizeof(clusterMsgDataPublish) - 8 + channel_len + message_len;

hdr->data.publish.msg.channel_len = htonl(channel_len);
hdr->data.publish.msg.message_len = htonl(message_len);
hdr->totlen = htonl(totlen);

/* Try to use the local buffer if possible */
if (totlen < sizeof(buf)) {
    payload = (unsigned char*)buf;
} else {
    payload = zmalloc(totlen);
    memcpy(payload,hdr,sizeof(*hdr));
    hdr = (clusterMsg*) payload;
}
memcpy(hdr->data.publish.msg.bulk_data,channel->ptr,sdslen(channel->ptr));
memcpy(hdr->data.publish.msg.bulk_data+sdslen(channel->ptr),
    message->ptr,sdslen(message->ptr));

if (link)
    clusterSendMessage(link,payload,totlen);
else
    clusterBroadcastMessage(payload,totlen);

decrRefCount(channel);
decrRefCount(message);
if (payload != (unsigned char*)buf) zfree(payload);
}

/* Send a FAIL message to all the nodes we are able to contact.
 * The FAIL message is sent when we detect that a node is failing
 * (CLUSTER_NODE_PFAIL) and we also receive a gossip confirmation of this:
 * we switch the node state to CLUSTER_NODE_FAIL and ask all the other
 * nodes to do the same ASAP. */
void clusterSendFail(char *nodename) {
    clusterMsg buf[1];
    clusterMsg *hdr = (clusterMsg*) buf;

    clusterBuildMessageHdr(hdr,CLUSTERMSG_TYPE_FAIL);
    memcpy(hdr->data.fail.about.nodename,nodename,CLUSTER_NAMELEN);
    clusterBroadcastMessage(buf,ntohl(hdr->totlen));
}

/* Send an UPDATE message to the specified link carrying the specified 'node'
 * slots configuration. The node name, slots bitmap, and configEpoch info
 * are included. */
void clusterSendUpdate(clusterLink *link, clusterNode *node) {
    clusterMsg buf[1];
    clusterMsg *hdr = (clusterMsg*) buf;

```

```

    if (link == NULL) return;
    clusterBuildMessageHdr(hdr, CLUSTERMSG_TYPE_UPDATE);
    memcpy(hdr->data.update.nodcfg.nodename, node->name, CLUSTER_NAMELEN);
    hdr->data.update.nodcfg.configEpoch = htonu64(node->configEpoch);
    memcpy(hdr->data.update.nodcfg.slots, node->slots, sizeof(node->slots));
    clusterSendMessage(link, (unsigned char*)buf, ntohl(hdr->totlen));
}

/* Send a MODULE message.
 *
 * If link is NULL, then the message is broadcasted to the whole cluster. */
void clusterSendModule(clusterLink *link, uint64_t module_id, uint8_t type,
                      const char *payload, uint32_t len) {
    unsigned char *heapbuf;
    clusterMsg buf[1];
    clusterMsg *hdr = (clusterMsg*) buf;
    uint32_t totlen;

    clusterBuildMessageHdr(hdr, CLUSTERMSG_TYPE_MODULE);
    totlen = sizeof(clusterMsg) - sizeof(union clusterMsgData);
    totlen += sizeof(clusterMsgModule) - 3 + len;

    hdr->data.module.msg.module_id = module_id; /* Already endian adjusted. */
    hdr->data.module.msg.type = type;
    hdr->data.module.msg.len = htonl(len);
    hdr->totlen = htonl(totlen);

    /* Try to use the local buffer if possible */
    if (totlen < sizeof(buf)) {
        heapbuf = (unsigned char*)buf;
    } else {
        heapbuf = zmalloc(totlen);
        memcpy(heapbuf, hdr, sizeof(*hdr));
        hdr = (clusterMsg*) heapbuf;
    }
    memcpy(hdr->data.module.msg.bulk_data, payload, len);

    if (link)
        clusterSendMessage(link, heapbuf, totlen);
    else
        clusterBroadcastMessage(heapbuf, totlen);

    if (heapbuf != (unsigned char*)buf) zfree(heapbuf);
}

/* This function gets a cluster node ID string as target, the same way the
nodes
 * addresses are represented in the modules side, resolves the node, and sends
 * the message. If the target is NULL the message is broadcasted.
 *
 * The function returns C_OK if the target is valid, otherwise C_ERR is

```



```

* returned. */
int clusterSendMessageToTarget(const char *target, uint64_t module_id,
uint8_t type, const char *payload, uint32_t len) {
    clusterNode *node = NULL;

    if (target != NULL) {
        node = clusterLookupNode(target, strlen(target));
        if (node == NULL || node->link == NULL) return C_ERR;
    }

    clusterSendModule(target ? node->link : NULL,
                      module_id, type, payload, len);
    return C_OK;
}

/* -----
-
* CLUSTER Pub/Sub support
*
* If `sharded` is 0:
* For now we do very little, just propagating [S]PUBLISH messages across the
whole
* cluster. In the future we'll try to get smarter and avoiding propagating
those
* messages to hosts without receives for a given channel.
* Otherwise:
* Publish this message across the slot (primary/replica).
* -----
*/
void clusterPropagatePublish(robj *channel, robj *message, int sharded) {
    if (!sharded) {
        clusterSendPublish(NULL, channel, message, CLUSTERMSG_TYPE_PUBLISH);
        return;
    }

    list *nodes_for_slot = clusterGetNodesServingMySlots(server.cluster->myself);
    if (listLength(nodes_for_slot) != 0) {
        listIter li;
        listNode *ln;
        listRewind(nodes_for_slot, &li);
        while((ln = listNext(&li))) {
            clusterNode *node = listNodeValue(ln);
            if (node != myself) {
                clusterSendPublish(node->link, channel, message,
CLUSTERMSG_TYPE_PUBLISHSHARD);
            }
        }
        listRelease(nodes_for_slot);
    }
}

```

```

/* -----
-
* SLAVE node specific functions
* -----
*/

/* This function sends a FAILOVER_AUTH_REQUEST message to every node in order
to
* see if there is the quorum for this slave instance to failover its failing
* master.
*
* Note that we send the failover request to everybody, master and slave nodes,
* but only the masters are supposed to reply to our query. */
void clusterRequestFailoverAuth(void) {
    clusterMsg buf[1];
    clusterMsg *hdr = (clusterMsg*) buf;
    uint32_t totlen;

    clusterBuildMessageHdr(hdr, CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST);
    /* If this is a manual failover, set the CLUSTERMSG_FLAG0_FORCEACK bit
    * in the header to communicate the nodes receiving the message that
    * they should authorized the failover even if the master is working. */
    if (server.cluster->mf_end) hdr->mflags[0] |= CLUSTERMSG_FLAG0_FORCEACK;
    totlen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
    hdr->totlen = htonl(totlen);
    clusterBroadcastMessage(buf, totlen);
}

/* Send a FAILOVER_AUTH_ACK message to the specified node. */
void clusterSendFailoverAuth(clusterNode *node) {
    clusterMsg buf[1];
    clusterMsg *hdr = (clusterMsg*) buf;
    uint32_t totlen;

    if (!node->link) return;
    clusterBuildMessageHdr(hdr, CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK);
    totlen = sizeof(clusterMsg)-sizeof(union clusterMsgData);
    hdr->totlen = htonl(totlen);
    clusterSendMessage(node->link, (unsigned char*)buf, totlen);
}

/* Send a MFSTART message to the specified node. */
void clusterSendMFStart(clusterNode *node) {
    clusterMsg buf[1];
    clusterMsg *hdr = (clusterMsg*) buf;
    uint32_t totlen;

    if (!node->link) return;
    clusterBuildMessageHdr(hdr, CLUSTERMSG_TYPE_MFSTART);
    totlen = sizeof(clusterMsg)-sizeof(union clusterMsgData);

```

```

    hdr->totlen = htonl(totlen);
    clusterSendMessage(node->link, (unsigned char*)buf, totlen);
}

/* Vote for the node asking for our vote if there are the conditions. */
void clusterSendFailoverAuthIfNeeded(clusterNode *node, clusterMsg *request) {
    clusterNode *master = node->slaveof;
    uint64_t requestCurrentEpoch = ntohu64(request->currentEpoch);
    uint64_t requestConfigEpoch = ntohu64(request->configEpoch);
    unsigned char *claimed_slots = request->myslots;
    int force_ack = request->mflags[0] & CLUSTERMSG_FLAG0_FORCEACK;
    int j;

    /* IF we are not a master serving at least 1 slot, we don't have the
     * right to vote, as the cluster size in Redis Cluster is the number
     * of masters serving at least one slot, and quorum is the cluster
     * size + 1 */
    if (nodeIsSlave(myself) || myself->numslots == 0) return;

    /* Request epoch must be >= our currentEpoch.
     * Note that it is impossible for it to actually be greater since
     * our currentEpoch was updated as a side effect of receiving this
     * request, if the request epoch was greater. */
    if (requestCurrentEpoch < server.cluster->currentEpoch) {
        serverLog(LL_WARNING,
            "Failover auth denied to %.40s: reqEpoch (%llu) < curEpoch(%llu)",
            node->name,
            (unsigned long long) requestCurrentEpoch,
            (unsigned long long) server.cluster->currentEpoch);
        return;
    }

    /* I already voted for this epoch? Return ASAP. */
    if (server.cluster->lastVoteEpoch == server.cluster->currentEpoch) {
        serverLog(LL_WARNING,
            "Failover auth denied to %.40s: already voted for epoch %llu",
            node->name,
            (unsigned long long) server.cluster->currentEpoch);
        return;
    }

    /* Node must be a slave and its master down.
     * The master can be non failing if the request is flagged
     * with CLUSTERMSG_FLAG0_FORCEACK (manual failover). */
    if (nodeIsMaster(node) || master == NULL ||
        (!nodeFailed(master) && !force_ack))
    {
        if (nodeIsMaster(node)) {
            serverLog(LL_WARNING,
                "Failover auth denied to %.40s: it is a master node",
                node->name);
        }
    }
}

```

```

    } else if (master == NULL) {
        serverLog(LL_WARNING,
            "Failover auth denied to %.40s: I don't know its master",
            node->name);
    } else if (!nodeFailed(master)) {
        serverLog(LL_WARNING,
            "Failover auth denied to %.40s: its master is up",
            node->name);
    }
    return;
}

/* We did not voted for a slave about this master for two
 * times the node timeout. This is not strictly needed for correctness
 * of the algorithm but makes the base case more linear. */
if (mstime() - node->slaveof->voted_time < server.cluster_node_timeout * 2)
{
    serverLog(LL_WARNING,
        "Failover auth denied to %.40s: "
        "can't vote about this master before %lld milliseconds",
        node->name,
        (long long) ((server.cluster_node_timeout*2)-
            (mstime() - node->slaveof->voted_time)));
    return;
}

/* The slave requesting the vote must have a configEpoch for the claimed
 * slots that is >= the one of the masters currently serving the same
 * slots in the current configuration. */
for (j = 0; j < CLUSTER_SLOTS; j++) {
    if (bitmapTestBit(claimed_slots, j) == 0) continue;
    if (server.cluster->slots[j] == NULL ||
        server.cluster->slots[j]->configEpoch <= requestConfigEpoch)
    {
        continue;
    }
    /* If we reached this point we found a slot that in our current slots
     * is served by a master with a greater configEpoch than the one
claimed
     * by the slave requesting our vote. Refuse to vote for this slave. */
    serverLog(LL_WARNING,
        "Failover auth denied to %.40s: "
        "slot %d epoch (%llu) > reqEpoch (%llu)",
        node->name, j,
        (unsigned long long) server.cluster->slots[j]->configEpoch,
        (unsigned long long) requestConfigEpoch);
    return;
}

/* We can vote for this slave. */
server.cluster->lastVoteEpoch = server.cluster->currentEpoch;

```

```

node->slaveof->voted_time = mstime();
clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|CLUSTER_TODO_FSYNC_CONFIG);
clusterSendFailoverAuth(node);
serverLog(LL_WARNING, "Failover auth granted to %.40s for epoch %llu",
          node->name, (unsigned long long) server.cluster->currentEpoch);
}

```

```

/* This function returns the "rank" of this instance, a slave, in the context
 * of its master-slaves ring. The rank of the slave is given by the number of
 * other slaves for the same master that have a better replication offset
 * compared to the local one (better means, greater, so they claim more data).
 *
 * A slave with rank 0 is the one with the greatest (most up to date)
 * replication offset, and so forth. Note that because how the rank is computed
 * multiple slaves may have the same rank, in case they have the same offset.
 *
 * The slave rank is used to add a delay to start an election in order to
 * get voted and replace a failing master. Slaves with better replication
 * offsets are more likely to win. */

```

```

int clusterGetSlaveRank(void) {
    long long myoffset;
    int j, rank = 0;
    clusterNode *master;

    serverAssert(nodeIsSlave(myself));
    master = myself->slaveof;
    if (master == NULL) return 0; /* Never called by slaves without master. */

    myoffset = replicationGetSlaveOffset();
    for (j = 0; j < master->numslaves; j++)
        if (master->slaves[j] != myself &&
            !nodeCantFailover(master->slaves[j]) &&
            master->slaves[j]->repl_offset > myoffset) rank++;
    return rank;
}

```

```

/* This function is called by clusterHandleSlaveFailover() in order to
 * let the slave log why it is not able to failover. Sometimes there are
 * not the conditions, but since the failover function is called again and
 * again, we can't log the same things continuously.
 *
 * This function works by logging only if a given set of conditions are
 * true:
 *
 * 1) The reason for which the failover can't be initiated changed.
 *    The reasons also include a NONE reason we reset the state to
 *    when the slave finds that its master is fine (no FAIL flag).
 * 2) Also, the log is emitted again if the master is still down and
 *    the reason for not failing over is still the same, but more than
 *    CLUSTER_CANT_FAILOVER_RELOG_PERIOD seconds elapsed.
 * 3) Finally, the function only logs if the slave is down for more than

```

```

*   five seconds + NODE_TIMEOUT. This way nothing is logged when a
*   failover starts in a reasonable time.
*
* The function is called with the reason why the slave can't failover
* which is one of the integer macros CLUSTER_CANT_FAILOVER_*.
*
* The function is guaranteed to be called only if 'myself' is a slave. */
void clusterLogCantFailover(int reason) {
    char *msg;
    static time_t lastlog_time = 0;
    mstime_t nolog_fail_time = server.cluster_node_timeout + 5000;

    /* Don't log if we have the same reason for some time. */
    if (reason == server.cluster->cant_failover_reason &&
        time(NULL)-lastlog_time < CLUSTER_CANT_FAILOVER_RELOG_PERIOD)
        return;

    server.cluster->cant_failover_reason = reason;

    /* We also don't emit any log if the master failed no long ago, the
     * goal of this function is to log slaves in a stalled condition for
     * a long time. */
    if (myself->slaveof &&
        nodeFailed(myself->slaveof) &&
        (mstime() - myself->slaveof->fail_time) < nolog_fail_time) return;

    switch(reason) {
    case CLUSTER_CANT_FAILOVER_DATA_AGE:
        msg = "Disconnected from master for longer than allowed. "
            "Please check the 'cluster-replica-validity-factor' configuration
            "
            "option.";
        break;
    case CLUSTER_CANT_FAILOVER_WAITING_DELAY:
        msg = "Waiting the delay before I can start a new failover.";
        break;
    case CLUSTER_CANT_FAILOVER_EXPIRED:
        msg = "Failover attempt expired.";
        break;
    case CLUSTER_CANT_FAILOVER_WAITING_VOTES:
        msg = "Waiting for votes, but majority still not reached.";
        break;
    default:
        msg = "Unknown reason code.";
        break;
    }
    lastlog_time = time(NULL);
    serverLog(LL_WARNING, "Currently unable to failover: %s", msg);
}

/* This function implements the final part of automatic and manual failovers,

```

```

* where the slave grabs its master's hash slots, and propagates the new
* configuration.
*
* Note that it's up to the caller to be sure that the node got a new
* configuration epoch already. */
void clusterFailoverReplaceYourMaster(void) {
    int j;
    clusterNode *oldmaster = myself->slaveof;

    if (nodeIsMaster(myself) || oldmaster == NULL) return;

    /* 1) Turn this node into a master. */
    clusterSetNodeAsMaster(myself);
    replicationUnsetMaster();

    /* 2) Claim all the slots assigned to our master. */
    for (j = 0; j < CLUSTER_SLOTS; j++) {
        if (clusterNodeGetSlotBit(oldmaster,j)) {
            clusterDelSlot(j);
            clusterAddSlot(myself,j);
        }
    }

    /* 3) Update state and save config. */
    clusterUpdateState();
    clusterSaveConfigOrDie(1);

    /* 4) Pong all the other nodes so that they can update the state
     * accordingly and detect that we switched to master role. */
    clusterBroadcastPong(CLUSTER_BROADCAST_ALL);

    /* 5) If there was a manual failover in progress, clear the state. */
    resetManualFailover();
}

/* This function is called if we are a slave node and our master serving
 * a non-zero amount of hash slots is in FAIL state.
 *
 * The goal of this function is:
 * 1) To check if we are able to perform a failover, is our data updated?
 * 2) Try to get elected by masters.
 * 3) Perform the failover informing all the other nodes.
 */
void clusterHandleSlaveFailover(void) {
    mstime_t data_age;
    mstime_t auth_age = mstime() - server.cluster->failover_auth_time;
    int needed_quorum = (server.cluster->size / 2) + 1;
    int manual_failover = server.cluster->mf_end != 0 &&
                          server.cluster->mf_can_start;
    mstime_t auth_timeout, auth_retry_time;

```

```

server.cluster->todo_before_sleep &= ~CLUSTER_TODO_HANDLE_FAILOVER;

/* Compute the failover timeout (the max time we have to send votes
 * and wait for replies), and the failover retry time (the time to wait
 * before trying to get voted again).
 *
 * Timeout is MAX(NODE_TIMEOUT*2,2000) milliseconds.
 * Retry is two times the Timeout.
 */
auth_timeout = server.cluster_node_timeout*2;
if (auth_timeout < 2000) auth_timeout = 2000;
auth_retry_time = auth_timeout*2;

/* Pre conditions to run the function, that must be met both in case
 * of an automatic or manual failover:
 * 1) We are a slave.
 * 2) Our master is flagged as FAIL, or this is a manual failover.
 * 3) We don't have the no failover configuration set, and this is
 *    not a manual failover.
 * 4) It is serving slots. */
if (nodeIsMaster(myself) ||
    myself->slaveof == NULL ||
    (!nodeFailed(myself->slaveof) && !manual_failover) ||
    (server.cluster_slave_no_failover && !manual_failover) ||
    myself->slaveof->numslots == 0)
{
    /* There are no reasons to failover, so we set the reason why we
     * are returning without failing over to NONE. */
    server.cluster->cant_failover_reason = CLUSTER_CANT_FAILOVER_NONE;
    return;
}

/* Set data_age to the number of milliseconds we are disconnected from
 * the master. */
if (server.repl_state == REPL_STATE_CONNECTED) {
    data_age = (mstime_t)(server.unixtime - server.master->lastinteraction)
                * 1000;
} else {
    data_age = (mstime_t)(server.unixtime - server.repl_down_since) * 1000;
}

/* Remove the node timeout from the data age as it is fine that we are
 * disconnected from our master at least for the time it was down to be
 * flagged as FAIL, that's the baseline. */
if (data_age > server.cluster_node_timeout)
    data_age -= server.cluster_node_timeout;

/* Check if our data is recent enough according to the slave validity
 * factor configured by the user.
 *
 * Check bypassed for manual failovers. */

```



```

if (server.cluster_slave_validity_factor &&
    data_age >
    (((mstime_t)server.repl_ping_slave_period * 1000) +
     (server.cluster_node_timeout * server.cluster_slave_validity_factor)))
{
    if (!manual_failover) {
        clusterLogCantFailover(CLUSTER_CANT_FAILOVER_DATA_AGE);
        return;
    }
}

/* If the previous failover attempt timeout and the retry time has
 * elapsed, we can setup a new one. */
if (auth_age > auth_retry_time) {
    server.cluster->failover_auth_time = mstime() +
        500 + /* Fixed delay of 500 milliseconds, let FAIL msg propagate.
*/
        random() % 500; /* Random delay between 0 and 500 milliseconds. */
    server.cluster->failover_auth_count = 0;
    server.cluster->failover_auth_sent = 0;
    server.cluster->failover_auth_rank = clusterGetSlaveRank();
    /* We add another delay that is proportional to the slave rank.
     * Specifically 1 second * rank. This way slaves that have a probably
     * less updated replication offset, are penalized. */
    server.cluster->failover_auth_time +=
        server.cluster->failover_auth_rank * 1000;
    /* However if this is a manual failover, no delay is needed. */
    if (server.cluster->mf_end) {
        server.cluster->failover_auth_time = mstime();
        server.cluster->failover_auth_rank = 0;
        clusterDoBeforeSleep(CLUSTER_TODO_HANDLE_FAILOVER);
    }
    serverLog(LL_WARNING,
        "Start of election delayed for %lld milliseconds "
        "(rank #%d, offset %lld).",
        server.cluster->failover_auth_time - mstime(),
        server.cluster->failover_auth_rank,
        replicationGetSlaveOffset());
    /* Now that we have a scheduled election, broadcast our offset
     * to all the other slaves so that they'll updated their offsets
     * if our offset is better. */
    clusterBroadcastPong(CLUSTER_BROADCAST_LOCAL_SLAVES);
    return;
}

/* It is possible that we received more updated offsets from other
 * slaves for the same master since we computed our election delay.
 * Update the delay if our rank changed.
 *
 * Not performed if this is a manual failover. */
if (server.cluster->failover_auth_sent == 0 &&

```

```

server.cluster->mf_end == 0)
{
    int newrank = clusterGetSlaveRank();
    if (newrank > server.cluster->failover_auth_rank) {
        long long added_delay =
            (newrank - server.cluster->failover_auth_rank) * 1000;
        server.cluster->failover_auth_time += added_delay;
        server.cluster->failover_auth_rank = newrank;
        serverLog(LL_WARNING,
            "Replica rank updated to #%, added %lld milliseconds of
delay.",
            newrank, added_delay);
    }
}

/* Return ASAP if we can't still start the election. */
if (mstime() < server.cluster->failover_auth_time) {
    clusterLogCantFailover(CLUSTER_CANT_FAILOVER_WAITING_DELAY);
    return;
}

/* Return ASAP if the election is too old to be valid. */
if (auth_age > auth_timeout) {
    clusterLogCantFailover(CLUSTER_CANT_FAILOVER_EXPIRED);
    return;
}

/* Ask for votes if needed. */
if (server.cluster->failover_auth_sent == 0) {
    server.cluster->currentEpoch++;
    server.cluster->failover_auth_epoch = server.cluster->currentEpoch;
    serverLog(LL_WARNING, "Starting a failover election for epoch %llu.",
        (unsigned long long) server.cluster->currentEpoch);
    clusterRequestFailoverAuth();
    server.cluster->failover_auth_sent = 1;
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|
                        CLUSTER_TODO_UPDATE_STATE|
                        CLUSTER_TODO_FSYNC_CONFIG);
    return; /* Wait for replies. */
}

/* Check if we reached the quorum. */
if (server.cluster->failover_auth_count >= needed_quorum) {
    /* We have the quorum, we can finally failover the master. */

    serverLog(LL_WARNING,
        "Failover election won: I'm the new master.");

    /* Update my configEpoch to the epoch of the election. */
    if (myself->configEpoch < server.cluster->failover_auth_epoch) {
        myself->configEpoch = server.cluster->failover_auth_epoch;
    }
}

```

```

        serverLog(LL_WARNING,
                  "configEpoch set to %llu after successful failover",
                  (unsigned long long) myself->configEpoch);
    }

    /* Take responsibility for the cluster slots. */
    clusterFailoverReplaceYourMaster();
} else {
    clusterLogCantFailover(CLUSTER_CANT_FAILOVER_WAITING_VOTES);
}
}

/* -----
-
* CLUSTER slave migration
*
* Slave migration is the process that allows a slave of a master that is
* already covered by at least another slave, to "migrate" to a master that
* is orphaned, that is, left with no working slaves.
* ----- */

/* This function is responsible to decide if this replica should be migrated
* to a different (orphaned) master. It is called by the clusterCron() function
* only if:
*
* 1) We are a slave node.
* 2) It was detected that there is at least one orphaned master in
*    the cluster.
* 3) We are a slave of one of the masters with the greatest number of
*    slaves.
*
* This checks are performed by the caller since it requires to iterate
* the nodes anyway, so we spend time into clusterHandleSlaveMigration()
* if definitely needed.
*
* The function is called with a pre-computed max_slaves, that is the max
* number of working (not in FAIL state) slaves for a single master.
*
* Additional conditions for migration are examined inside the function.
*/
void clusterHandleSlaveMigration(int max_slaves) {
    int j, okslaves = 0;
    clusterNode *mymaster = myself->slaveof, *target = NULL, *candidate = NULL;
    dictIterator *di;
    dictEntry *de;

    /* Step 1: Don't migrate if the cluster state is not ok. */
    if (server.cluster->state != CLUSTER_OK) return;

    /* Step 2: Don't migrate if my master will not be left with at least
    *          'migration-barrier' slaves after my migration. */

```

```

if (mymaster == NULL) return;
for (j = 0; j < mymaster->numslaves; j++)
    if (!nodeFailed(mymaster->slaves[j]) &&
        !nodeTimedOut(mymaster->slaves[j])) okslaves++;
if (okslaves <= server.cluster_migration_barrier) return;

/* Step 3: Identify a candidate for migration, and check if among the
 * masters with the greatest number of ok slaves, I'm the one with the
 * smallest node ID (the "candidate slave").
 *
 * Note: this means that eventually a replica migration will occur
 * since slaves that are reachable again always have their FAIL flag
 * cleared, so eventually there must be a candidate.
 * There is a possible race condition causing multiple
 * slaves to migrate at the same time, but this is unlikely to
 * happen and relatively harmless when it does. */
candidate = myself;
di = dictGetSafeIterator(server.cluster->nodes);
while((de = dictNext(di)) != NULL) {
    clusterNode *node = dictGetVal(de);
    int okslaves = 0, is_orphaned = 1;

    /* We want to migrate only if this master is working, orphaned, and
     * used to have slaves or if failed over a master that had slaves
     * (MIGRATE_TO flag). This way we only migrate to instances that were
     * supposed to have replicas. */
    if (nodeIsSlave(node) || nodeFailed(node)) is_orphaned = 0;
    if (!(node->flags & CLUSTER_NODE_MIGRATE_TO)) is_orphaned = 0;

    /* Check number of working slaves. */
    if (nodeIsMaster(node)) okslaves = clusterCountNonFailingSlaves(node);
    if (okslaves > 0) is_orphaned = 0;

    if (is_orphaned) {
        if (!target && node->numslots > 0) target = node;

        /* Track the starting time of the orphaned condition for this
         * master. */
        if (!node->orphaned_time) node->orphaned_time = mstime();
    } else {
        node->orphaned_time = 0;
    }

    /* Check if I'm the slave candidate for the migration: attached
     * to a master with the maximum number of slaves and with the smallest
     * node ID. */
    if (okslaves == max_slaves) {
        for (j = 0; j < node->numslaves; j++) {
            if (memcmp(node->slaves[j]->name,
                candidate->name,
                CLUSTER_NAMELEN) < 0)

```

```

        {
            candidate = node->slaves[j];
        }
    }
}
dictReleaseIterator(di);

/* Step 4: perform the migration if there is a target, and if I'm the
 * candidate, but only if the master is continuously orphaned for a
 * couple of seconds, so that during failovers, we give some time to
 * the natural slaves of this instance to advertise their switch from
 * the old master to the new one. */
if (target && candidate == myself &&
    (mstime()-target->orphaned_time) > CLUSTER_SLAVE_MIGRATION_DELAY &&
    !(server.cluster_module_flags & CLUSTER_MODULE_FLAG_NO_FAILOVER))
{
    serverLog(LL_WARNING,"Migrating to orphaned master %.40s",
        target->name);
    clusterSetMaster(target);
}
}

/* -----
-
 * CLUSTER manual failover
 *
 * This are the important steps performed by slaves during a manual failover:
 * 1) User send CLUSTER FAILOVER command. The failover state is initialized
 *    setting mf_end to the millisecond unix time at which we'll abort the
 *    attempt.
 * 2) Slave sends a MFSTART message to the master requesting to pause clients
 *    for two times the manual failover timeout CLUSTER_MF_TIMEOUT.
 *    When master is paused for manual failover, it also starts to flag
 *    packets with CLUSTERMSG_FLAG0_PAUSED.
 * 3) Slave waits for master to send its replication offset flagged as PAUSED.
 * 4) If slave received the offset from the master, and its offset matches,
 *    mf_can_start is set to 1, and clusterHandleSlaveFailover() will perform
 *    the failover as usually, with the difference that the vote request
 *    will be modified to force masters to vote for a slave that has a
 *    working master.
 *
 * From the point of view of the master things are simpler: when a
 * PAUSE_CLIENTS packet is received the master sets mf_end as well and
 * the sender in mf_slave. During the time limit for the manual failover
 * the master will just send PINGs more often to this slave, flagged with
 * the PAUSED flag, so that the slave will set mf_master_offset when receiving
 * a packet from the master with this flag set.
 *
 * The goal of the manual failover is to perform a fast failover without
 * data loss due to the asynchronous master-slave replication.

```

```

* -----
*/

/* Reset the manual failover state. This works for both masters and slaves
 * as all the state about manual failover is cleared.
 *
 * The function can be used both to initialize the manual failover state at
 * startup or to abort a manual failover in progress. */
void resetManualFailover(void) {
    if (server.cluster->mf_slave) {
        /* We were a master failing over, so we paused clients. Regardless
         * of the outcome we unpause now to allow traffic again. */
        unpauseClients(PAUSE_DURING_FAILOVER);
    }
    server.cluster->mf_end = 0; /* No manual failover in progress. */
    server.cluster->mf_can_start = 0;
    server.cluster->mf_slave = NULL;
    server.cluster->mf_master_offset = -1;
}

/* If a manual failover timed out, abort it. */
void manualFailoverCheckTimeout(void) {
    if (server.cluster->mf_end && server.cluster->mf_end < mstime()) {
        serverLog(LL_WARNING, "Manual failover timed out.");
        resetManualFailover();
    }
}

/* This function is called from the cluster cron function in order to go
 * forward with a manual failover state machine. */
void clusterHandleManualFailover(void) {
    /* Return ASAP if no manual failover is in progress. */
    if (server.cluster->mf_end == 0) return;

    /* If mf_can_start is non-zero, the failover was already triggered so the
     * next steps are performed by clusterHandleSlaveFailover(). */
    if (server.cluster->mf_can_start) return;

    if (server.cluster->mf_master_offset == -1) return; /* Wait for offset...
*/

    if (server.cluster->mf_master_offset == replicationGetSlaveOffset()) {
        /* Our replication offset matches the master replication offset
         * announced after clients were paused. We can start the failover. */
        server.cluster->mf_can_start = 1;
        serverLog(LL_WARNING,
            "All master replication stream processed, "
            "manual failover can start.");
        clusterDoBeforeSleep(CLUSTER_TODO_HANDLE_FAILOVER);
        return;
    }
}

```

```

    clusterDoBeforeSleep(CLUSTER_TODO_HANDLE_MANUALFAILOVER);
}

/* -----
 * CLUSTER cron job
 * -----
 */

/* Check if the node is disconnected and re-establish the connection.
 * Also update a few stats while we are here, that can be used to make
 * better decisions in other part of the code. */
static int clusterNodeCronHandleReconnect(clusterNode *node, mstime_t
handshake_timeout, mstime_t now) {
    /* Not interested in reconnecting the link with myself or nodes
     * for which we have no address. */
    if (node->flags & (CLUSTER_NODE_MYSELF|CLUSTER_NODE_NOADDR)) return 1;

    if (node->flags & CLUSTER_NODE_PFAIL)
        server.cluster->stats_pfail_nodes++;

    /* A Node in HANDSHAKE state has a limited lifespan equal to the
     * configured node timeout. */
    if (nodeInHandshake(node) && now - node->ctime > handshake_timeout) {
        clusterDelNode(node);
        return 1;
    }

    if (node->link == NULL) {
        clusterLink *link = createClusterLink(node);
        link->conn = server.tls_cluster ? connCreateTLS() : connCreateSocket();
        connSetPrivateData(link->conn, link);
        if (connConnect(link->conn, node->ip, node->cport,
server.bind_source_addr,
            clusterLinkConnectHandler) == -1) {
            /* We got a synchronous error from connect before
             * clusterSendPing() had a chance to be called.
             * If node->ping_sent is zero, failure detection can't work,
             * so we claim we actually sent a ping now (that will
             * be really sent as soon as the link is obtained). */
            if (node->ping_sent == 0) node->ping_sent = mstime();
            serverLog(LL_DEBUG, "Unable to connect to "
                "Cluster Node [%s]:%d -> %s", node->ip,
                node->cport, server.neterr);

            freeClusterLink(link);
            return 0;
        }
    }
}
return 0;
}

```

```

static void resizeClusterLinkBuffer(clusterLink *link) {
    /* If unused space is a lot bigger than the used portion of the buffer
    then free up unused space.
    * We use a factor of 4 because of the greediness of sdsMakeRoomFor (used
    by sdscatlen). */
    if (link != NULL && sdsavail(link->sndbuf) / 4 > sdslen(link->sndbuf)) {
        link->sndbuf = sdsRemoveFreeSpace(link->sndbuf);
    }
}

/* Resize the send buffer of a node if it is wasting
* enough space. */
static void clusterNodeCronResizeBuffers(clusterNode *node) {
    resizeClusterLinkBuffer(node->link);
    resizeClusterLinkBuffer(node->inbound_link);
}

static void freeClusterLinkOnBufferLimitReached(clusterLink *link) {
    if (link == NULL || server.cluster_link_sndbuf_limit_bytes == 0) {
        return;
    }
    unsigned long long mem_link = sdsalloc(link->sndbuf);
    if (mem_link > server.cluster_link_sndbuf_limit_bytes) {
        serverLog(LL_WARNING, "Freeing cluster link(%s node %.40s, used memory:
        %llu) due to "
            "exceeding send buffer memory limit.", link->inbound ? "from" :
            "to",
            link->node ? link->node->name : "", mem_link);
        freeClusterLink(link);
        server.cluster->stat_cluster_links_buffer_limit_exceeded++;
    }
}

/* Free outbound link to a node if its send buffer size exceeded limit. */
static void clusterNodeCronFreeLinkOnBufferLimitReached(clusterNode *node) {
    freeClusterLinkOnBufferLimitReached(node->link);
    freeClusterLinkOnBufferLimitReached(node->inbound_link);
}

static size_t getClusterLinkMemUsage(clusterLink *link) {
    if (link != NULL) {
        return sizeof(clusterLink) + sdsalloc(link->sndbuf) + link->
        >rcvbuf_alloc;
    } else {
        return 0;
    }
}

/* Update memory usage statistics of all current cluster links */
static void clusterNodeCronUpdateClusterLinksMemUsage(clusterNode *node) {

```



```

server.stat_cluster_links_memory += getClusterLinkMemUsage(node->link);
server.stat_cluster_links_memory += getClusterLinkMemUsage(node->inbound_link);
}

/* This is executed 10 times every second */
void clusterCron(void) {
    dictIterator *di;
    dictEntry *de;
    int update_state = 0;
    int orphaned_masters; /* How many masters there are without ok slaves. */
    int max_slaves; /* Max number of ok slaves for a single master. */
    int this_slaves; /* Number of ok slaves for our master (if we are slave). */
    /*
    mstime_t min_pong = 0, now = mstime();
    clusterNode *min_pong_node = NULL;
    static unsigned long long iteration = 0;
    mstime_t handshake_timeout;

    iteration++; /* Number of times this function was called so far. */

    clusterUpdateMyselfHostname();

    /* The handshake timeout is the time after which a handshake node that was
    * not turned into a normal node is removed from the nodes. Usually it is
    * just the NODE_TIMEOUT value, but when NODE_TIMEOUT is too small we use
    * the value of 1 second. */
    handshake_timeout = server.cluster_node_timeout;
    if (handshake_timeout < 1000) handshake_timeout = 1000;

    /* Clear so clusterNodeCronHandleReconnect can count the number of nodes in
    PFAIL. */
    server.cluster->stats_pfail_nodes = 0;
    /* Clear so clusterNodeCronUpdateClusterLinksMemUsage can count the current
    memory usage of all cluster links. */
    server.stat_cluster_links_memory = 0;
    /* Run through some of the operations we want to do on each cluster node.
    */
    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);
        /* The sequence goes:
        * 1. We try to shrink link buffers if possible.
        * 2. We free the links whose buffers are still oversized after
possible shrinking.
        * 3. We update the latest memory usage of cluster links.
        * 4. We immediately attempt reconnecting after freeing links.
        */
        clusterNodeCronResizeBuffers(node);
        clusterNodeCronFreeLinkOnBufferLimitReached(node);
        clusterNodeCronUpdateClusterLinksMemUsage(node);

```

```

        /* The protocol is that function(s) below return non-zero if the node
was
        * terminated.
        */
        if(clusterNodeCronHandleReconnect(node, handshake_timeout, now))
continue;
    }
    dictReleaseIterator(di);

    /* Ping some random node 1 time every 10 iterations, so that we usually
ping
    * one random node every second. */
    if (!(iteration % 10)) {
        int j;

        /* Check a few random nodes and ping the one with the oldest
        * pong_received time. */
        for (j = 0; j < 5; j++) {
            de = dictGetRandomKey(server.cluster->nodes);
            clusterNode *this = dictGetVal(de);

            /* Don't ping nodes disconnected or with a ping currently active.
*/
            if (this->link == NULL || this->ping_sent != 0) continue;
            if (this->flags & (CLUSTER_NODE_MYSELF|CLUSTER_NODE_HANDSHAKE))
                continue;
            if (min_pong_node == NULL || min_pong > this->pong_received) {
                min_pong_node = this;
                min_pong = this->pong_received;
            }
        }
        if (min_pong_node) {
            serverLog(LL_DEBUG,"Pinging node %.40s", min_pong_node->name);
            clusterSendPing(min_pong_node->link, CLUSTERMSG_TYPE_PING);
        }
    }

    /* Iterate nodes to check if we need to flag something as failing.
    * This loop is also responsible to:
    * 1) Check if there are orphaned masters (masters without non failing
    *    slaves).
    * 2) Count the max number of non failing slaves for a single master.
    * 3) Count the number of slaves for our master, if we are a slave. */
    orphaned_masters = 0;
    max_slaves = 0;
    this_slaves = 0;
    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);
        now = mstime(); /* Use an updated time at every iteration. */

```

```

if (node->flags &
    (CLUSTER_NODE_MYSELF|CLUSTER_NODE_NOADDR|CLUSTER_NODE_HANDSHAKE))
    continue;

/* Orphaned master check, useful only if the current instance
 * is a slave that may migrate to another master. */
if (nodeIsSlave(myself) && nodeIsMaster(node) && !nodeFailed(node)) {
    int okslaves = clusterCountNonFailingSlaves(node);

    /* A master is orphaned if it is serving a non-zero number of
     * slots, have no working slaves, but used to have at least one
     * slave, or failed over a master that used to have slaves. */
    if (okslaves == 0 && node->numslots > 0 &&
        node->flags & CLUSTER_NODE_MIGRATE_TO)
    {
        orphaned_masters++;
    }
    if (okslaves > max_slaves) max_slaves = okslaves;
    if (nodeIsSlave(myself) && myself->slaveof == node)
        this_slaves = okslaves;
}

/* If we are not receiving any data for more than half the cluster
 * timeout, reconnect the link: maybe there is a connection
 * issue even if the node is alive. */
mstime_t ping_delay = now - node->ping_sent;
mstime_t data_delay = now - node->data_received;
if (node->link && /* is connected */
    now - node->link->ctime >
    server.cluster_node_timeout && /* was not already reconnected */
    node->ping_sent && /* we already sent a ping */
    /* and we are waiting for the pong more than timeout/2 */
    ping_delay > server.cluster_node_timeout/2 &&
    /* and in such interval we are not seeing any traffic at all. */
    data_delay > server.cluster_node_timeout/2)
{
    /* Disconnect the link, it will be reconnected automatically. */
    freeClusterLink(node->link);
}

/* If we have currently no active ping in this instance, and the
 * received PONG is older than half the cluster timeout, send
 * a new ping now, to ensure all the nodes are pinged without
 * a too big delay. */
if (node->link &&
    node->ping_sent == 0 &&
    (now - node->pong_received) > server.cluster_node_timeout/2)
{
    clusterSendPing(node->link, CLUSTERMSG_TYPE_PING);
    continue;
}

```

```

/* If we are a master and one of the slaves requested a manual
 * failover, ping it continuously. */
if (server.cluster->mf_end &&
    nodeIsMaster(myself) &&
    server.cluster->mf_slave == node &&
    node->link)
{
    clusterSendPing(node->link, CLUSTERMSG_TYPE_PING);
    continue;
}

/* Check only if we have an active ping for this instance. */
if (node->ping_sent == 0) continue;

/* Check if this node looks unreachable.
 * Note that if we already received the PONG, then node->ping_sent
 * is zero, so can't reach this code at all, so we don't risk of
 * checking for a PONG delay if we didn't sent the PING.
 *
 * We also consider every incoming data as proof of liveness, since
 * our cluster bus link is also used for data: under heavy data
 * load pong delays are possible. */
mstime_t node_delay = (ping_delay < data_delay) ? ping_delay :
                                                                data_delay;

if (node_delay > server.cluster_node_timeout) {
    /* Timeout reached. Set the node as possibly failing if it is
     * not already in this state. */
    if (!(node->flags & (CLUSTER_NODE_PFAIL|CLUSTER_NODE_FAIL))) {
        serverLog(LL_DEBUG,"*** NODE %.40s possibly failing",
            node->name);
        node->flags |= CLUSTER_NODE_PFAIL;
        update_state = 1;
    }
}
}
dictReleaseIterator(di);

/* If we are a slave node but the replication is still turned off,
 * enable it if we know the address of our master and it appears to
 * be up. */
if (nodeIsSlave(myself) &&
    server.masterhost == NULL &&
    myself->slaveof &&
    nodeHasAddr(myself->slaveof))
{
    replicationSetMaster(myself->slaveof->ip, myself->slaveof->port);
}

/* Abort a manual failover if the timeout is reached. */

```

```

manualFailoverCheckTimeout();

if (nodeIsSlave(myself)) {
    clusterHandleManualFailover();
    if (!(server.cluster_module_flags & CLUSTER_MODULE_FLAG_NO_FAILOVER))
        clusterHandleSlaveFailover();
    /* If there are orphaned slaves, and we are a slave among the masters
     * with the max number of non-failing slaves, consider migrating to
     * the orphaned masters. Note that it does not make sense to try
     * a migration if there is no master with at least *two* working
     * slaves. */
    if (orphaned_masters && max_slaves >= 2 && this_slaves == max_slaves &&
        server.cluster_allow_replica_migration)
        clusterHandleSlaveMigration(max_slaves);
}

if (update_state || server.cluster->state == CLUSTER_FAIL)
    clusterUpdateState();
}

/* This function is called before the event handler returns to sleep for
 * events. It is useful to perform operations that must be done ASAP in
 * reaction to events fired but that are not safe to perform inside event
 * handlers, or to perform potentially expansive tasks that we need to do
 * a single time before replying to clients. */
void clusterBeforeSleep(void) {
    int flags = server.cluster->todo_before_sleep;

    /* Reset our flags (not strictly needed since every single function
     * called for flags set should be able to clear its flag). */
    server.cluster->todo_before_sleep = 0;

    if (flags & CLUSTER_TODO_HANDLE_MANUALFAILOVER) {
        /* Handle manual failover as soon as possible so that won't have a
100ms
         * as it was handled only in clusterCron */
        if (nodeIsSlave(myself)) {
            clusterHandleManualFailover();
            if (!(server.cluster_module_flags &
CLUSTER_MODULE_FLAG_NO_FAILOVER))
                clusterHandleSlaveFailover();
        }
    } else if (flags & CLUSTER_TODO_HANDLE_FAILOVER) {
        /* Handle failover, this is needed when it is likely that there is
already
         * the quorum from masters in order to react fast. */
        clusterHandleSlaveFailover();
    }

    /* Update the cluster state. */
    if (flags & CLUSTER_TODO_UPDATE_STATE)

```

```

        clusterUpdateState();

    /* Save the config, possibly using fsync. */
    if (flags & CLUSTER_TODO_SAVE_CONFIG) {
        int fsync = flags & CLUSTER_TODO_FSYNC_CONFIG;
        clusterSaveConfigOrDie(fsync);
    }
}

void clusterDoBeforeSleep(int flags) {
    server.cluster->todo_before_sleep |= flags;
}

/* -----
 * Slots management
 * -----
 */

/* Test bit 'pos' in a generic bitmap. Return 1 if the bit is set,
 * otherwise 0. */
int bitmapTestBit(unsigned char *bitmap, int pos) {
    off_t byte = pos/8;
    int bit = pos&7;
    return (bitmap[byte] & (1<<bit)) != 0;
}

/* Set the bit at position 'pos' in a bitmap. */
void bitmapSetBit(unsigned char *bitmap, int pos) {
    off_t byte = pos/8;
    int bit = pos&7;
    bitmap[byte] |= 1<<bit;
}

/* Clear the bit at position 'pos' in a bitmap. */
void bitmapClearBit(unsigned char *bitmap, int pos) {
    off_t byte = pos/8;
    int bit = pos&7;
    bitmap[byte] &= ~(1<<bit);
}

/* Return non-zero if there is at least one master with slaves in the cluster.
 * Otherwise zero is returned. Used by clusterNodeSetSlotBit() to set the
 * MIGRATE_T0 flag the when a master gets the first slot. */
int clusterMastersHaveSlaves(void) {
    dictIterator *di = dictGetSafeIterator(server.cluster->nodes);
    dictEntry *de;
    int slaves = 0;
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);

```

```

        if (nodeIsSlave(node)) continue;
        slaves += node->numslaves;
    }
    dictReleaseIterator(di);
    return slaves != 0;
}

/* Set the slot bit and return the old value. */
int clusterNodeSetSlotBit(clusterNode *n, int slot) {
    int old = bitmapTestBit(n->slots,slot);
    bitmapSetBit(n->slots,slot);
    if (!old) {
        n->numslots++;
        /* When a master gets its first slot, even if it has no slaves,
         * it gets flagged with MIGRATE_T0, that is, the master is a valid
         * target for replicas migration, if and only if at least one of
         * the other masters has slaves right now.
         *
         * Normally masters are valid targets of replica migration if:
         * 1. The used to have slaves (but no longer have).
         * 2. They are slaves failing over a master that used to have slaves.
         *
         * However new masters with slots assigned are considered valid
         * migration targets if the rest of the cluster is not a slave-less.
         *
         * See https://github.com/redis/redis/issues/3043 for more info. */
        if (n->numslots == 1 && clusterMastersHaveSlaves())
            n->flags |= CLUSTER_NODE_MIGRATE_T0;
    }
    return old;
}

/* Clear the slot bit and return the old value. */
int clusterNodeClearSlotBit(clusterNode *n, int slot) {
    int old = bitmapTestBit(n->slots,slot);
    bitmapClearBit(n->slots,slot);
    if (old) n->numslots--;
    return old;
}

/* Return the slot bit from the cluster node structure. */
int clusterNodeGetSlotBit(clusterNode *n, int slot) {
    return bitmapTestBit(n->slots,slot);
}

/* Add the specified slot to the list of slots that node 'n' will
 * serve. Return C_OK if the operation ended with success.
 * If the slot is already assigned to another instance this is considered
 * an error and C_ERR is returned. */
int clusterAddSlot(clusterNode *n, int slot) {
    if (server.cluster->slots[slot]) return C_ERR;

```

```

    clusterNodeSetSlotBit(n,slot);
    server.cluster->slots[slot] = n;
    return C_OK;
}

/* Delete the specified slot marking it as unassigned.
 * Returns C_OK if the slot was assigned, otherwise if the slot was
 * already unassigned C_ERR is returned. */
int clusterDelSlot(int slot) {
    clusterNode *n = server.cluster->slots[slot];

    if (!n) return C_ERR;

    /* Cleanup the channels in master/replica as part of slot deletion. */
    list *nodes_for_slot = clusterGetNodesServingMySlots(n);
    listNode *ln = listSearchKey(nodes_for_slot, myself);
    if (ln != NULL) {
        removeChannelsInSlot(slot);
    }
    listRelease(nodes_for_slot);
    serverAssert(clusterNodeClearSlotBit(n,slot) == 1);
    server.cluster->slots[slot] = NULL;
    return C_OK;
}

/* Delete all the slots associated with the specified node.
 * The number of deleted slots is returned. */
int clusterDelNodeSlots(clusterNode *node) {
    int deleted = 0, j;

    for (j = 0; j < CLUSTER_SLOTS; j++) {
        if (clusterNodeGetSlotBit(node,j)) {
            clusterDelSlot(j);
            deleted++;
        }
    }
    return deleted;
}

/* Clear the migrating / importing state for all the slots.
 * This is useful at initialization and when turning a master into slave. */
void clusterCloseAllSlots(void) {
    memset(server.cluster->migrating_slots_to,0,
           sizeof(server.cluster->migrating_slots_to));
    memset(server.cluster->importing_slots_from,0,
           sizeof(server.cluster->importing_slots_from));
}

/* -----
 * Cluster state evaluation function

```



```

* -----
*/

/* The following are defines that are only used in the evaluation function
 * and are based on heuristics. Actually the main point about the rejoin and
 * writable delay is that they should be a few orders of magnitude larger
 * than the network latency. */
#define CLUSTER_MAX_REJOIN_DELAY 5000
#define CLUSTER_MIN_REJOIN_DELAY 500
#define CLUSTER_WRITABLE_DELAY 2000

void clusterUpdateState(void) {
    int j, new_state;
    int reachable_masters = 0;
    static mstime_t among_minority_time;
    static mstime_t first_call_time = 0;

    server.cluster->todo_before_sleep &= ~CLUSTER_TODO_UPDATE_STATE;

    /* If this is a master node, wait some time before turning the state
     * into OK, since it is not a good idea to rejoin the cluster as a writable
     * master, after a reboot, without giving the cluster a chance to
     * reconfigure this node. Note that the delay is calculated starting from
     * the first call to this function and not since the server start, in order
     * to not count the DB loading time. */
    if (first_call_time == 0) first_call_time = mstime();
    if (nodeIsMaster(myself) &&
        server.cluster->state == CLUSTER_FAIL &&
        mstime() - first_call_time < CLUSTER_WRITABLE_DELAY) return;

    /* Start assuming the state is OK. We'll turn it into FAIL if there
     * are the right conditions. */
    new_state = CLUSTER_OK;

    /* Check if all the slots are covered. */
    if (server.cluster_require_full_coverage) {
        for (j = 0; j < CLUSTER_SLOTS; j++) {
            if (server.cluster->slots[j] == NULL ||
                server.cluster->slots[j]->flags & (CLUSTER_NODE_FAIL))
            {
                new_state = CLUSTER_FAIL;
                break;
            }
        }
    }

    /* Compute the cluster size, that is the number of master nodes
     * serving at least a single slot.
     *
     * At the same time count the number of reachable masters having
     * at least one slot. */

```

```

{
    dictIterator *di;
    dictEntry *de;

    server.cluster->size = 0;
    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);

        if (nodeIsMaster(node) && node->numslots) {
            server.cluster->size++;
            if ((node->flags & (CLUSTER_NODE_FAIL|CLUSTER_NODE_PFAIL)) ==
0)
                reachable_masters++;
        }
    }
    dictReleaseIterator(di);
}

/* If we are in a minority partition, change the cluster state
 * to FAIL. */
{
    int needed_quorum = (server.cluster->size / 2) + 1;

    if (reachable_masters < needed_quorum) {
        new_state = CLUSTER_FAIL;
        among_minority_time = mstime();
    }
}

/* Log a state change */
if (new_state != server.cluster->state) {
    mstime_t rejoin_delay = server.cluster_node_timeout;

    /* If the instance is a master and was partitioned away with the
     * minority, don't let it accept queries for some time after the
     * partition heals, to make sure there is enough time to receive
     * a configuration update. */
    if (rejoin_delay > CLUSTER_MAX_REJOIN_DELAY)
        rejoin_delay = CLUSTER_MAX_REJOIN_DELAY;
    if (rejoin_delay < CLUSTER_MIN_REJOIN_DELAY)
        rejoin_delay = CLUSTER_MIN_REJOIN_DELAY;

    if (new_state == CLUSTER_OK &&
        nodeIsMaster(myself) &&
        mstime() - among_minority_time < rejoin_delay)
    {
        return;
    }

    /* Change the state and log the event. */

```

```

        serverLog(LL_WARNING,"Cluster state changed: %s",
            new_state == CLUSTER_OK ? "ok" : "fail");
        server.cluster->state = new_state;
    }
}

/* This function is called after the node startup in order to verify that data
 * loaded from disk is in agreement with the cluster configuration:
 *
 * 1) If we find keys about hash slots we have no responsibility for, the
 *    following happens:
 *    A) If no other node is in charge according to the current cluster
 *       configuration, we add these slots to our node.
 *    B) If according to our config other nodes are already in charge for
 *       this slots, we set the slots as IMPORTING from our point of view
 *       in order to justify we have those slots, and in order to make
 *       redis-cli aware of the issue, so that it can try to fix it.
 * 2) If we find data in a DB different than DB0 we return C_ERR to
 *    signal the caller it should quit the server with an error message
 *    or take other actions.
 *
 * The function always returns C_OK even if it will try to correct
 * the error described in "1". However if data is found in DB different
 * from DB0, C_ERR is returned.
 *
 * The function also uses the logging facility in order to warn the user
 * about desynchronizations between the data we have in memory and the
 * cluster configuration. */
int verifyClusterConfigWithData(void) {
    int j;
    int update_config = 0;

    /* Return ASAP if a module disabled cluster redirections. In that case
     * every master can store keys about every possible hash slot. */
    if (server.cluster_module_flags & CLUSTER_MODULE_FLAG_NO_REDIRECTION)
        return C_OK;

    /* If this node is a slave, don't perform the check at all as we
     * completely depend on the replication stream. */
    if (nodeIsSlave(myself)) return C_OK;

    /* Make sure we only have keys in DB0. */
    for (j = 1; j < server.dbnum; j++) {
        if (dictSize(server.db[j].dict)) return C_ERR;
    }

    /* Check that all the slots we see populated memory have a corresponding
     * entry in the cluster table. Otherwise fix the table. */
    for (j = 0; j < CLUSTER_SLOTS; j++) {
        if (!countKeysInSlot(j)) continue; /* No keys in this slot. */
        /* Check if we are assigned to this slot or if we are importing it.

```

```

    * In both cases check the next slot as the configuration makes
    * sense. */
    if (server.cluster->slots[j] == myself ||
        server.cluster->importing_slots_from[j] != NULL) continue;

    /* If we are here data and cluster config don't agree, and we have
    * slot 'j' populated even if we are not importing it, nor we are
    * assigned to this slot. Fix this condition. */

    update_config++;
    /* Case A: slot is unassigned. Take responsibility for it. */
    if (server.cluster->slots[j] == NULL) {
        serverLog(LL_WARNING, "I have keys for unassigned slot %d. "
                        "Taking responsibility for it.",j);
        clusterAddSlot(myself,j);
    } else {
        serverLog(LL_WARNING, "I have keys for slot %d, but the slot is "
                        "assigned to another node. "
                        "Setting it to importing state.",j);
        server.cluster->importing_slots_from[j] = server.cluster->slots[j];
    }
}
if (update_config) clusterSaveConfigOrDie(1);
return C_OK;
}

/* -----
-
* SLAVE nodes handling
* -----
*/

/* Set the specified node 'n' as master for this node.
* If this node is currently a master, it is turned into a slave. */
void clusterSetMaster(clusterNode *n) {
    serverAssert(n != myself);
    serverAssert(myself->numslots == 0);

    if (nodeIsMaster(myself)) {
        myself->flags &= ~(CLUSTER_NODE_MASTER|CLUSTER_NODE_MIGRATE_TO);
        myself->flags |= CLUSTER_NODE_SLAVE;
        clusterCloseAllSlots();
    } else {
        if (myself->slaveof)
            clusterNodeRemoveSlave(myself->slaveof,myself);
    }
    myself->slaveof = n;
    clusterNodeAddSlave(n,myself);
    replicationSetMaster(n->ip, n->port);
    resetManualFailover();
}

```

```

/* -----
-
* Nodes to string representation functions.
* -----
*/

struct redisNodeFlags {
    uint16_t flag;
    char *name;
};

static struct redisNodeFlags redisNodeFlagsTable[] = {
    {CLUSTER_NODE_MYSELF,      "myself,"},
    {CLUSTER_NODE_MASTER,     "master,"},
    {CLUSTER_NODE_SLAVE,      "slave,"},
    {CLUSTER_NODE_PFAIL,      "fail?,"},
    {CLUSTER_NODE_FAIL,       "fail,"},
    {CLUSTER_NODE_HANDSHAKE,   "handshake,"},
    {CLUSTER_NODE_NOADDR,     "noaddr,"},
    {CLUSTER_NODE_NOFAILOVER,  "nofailover,"}
};

/* Concatenate the comma separated list of node flags to the given SDS
* string 'ci'. */
sds representClusterNodeFlags(sds ci, uint16_t flags) {
    size_t orig_len = sdslen(ci);
    int i, size = sizeof(redisNodeFlagsTable)/sizeof(struct redisNodeFlags);
    for (i = 0; i < size; i++) {
        struct redisNodeFlags *nodeflag = redisNodeFlagsTable + i;
        if (flags & nodeflag->flag) ci = sdscat(ci, nodeflag->name);
    }
    /* If no flag was added, add the "noflags" special flag. */
    if (sdslen(ci) == orig_len) ci = sdscat(ci, "noflags,");
    sdsIncrLen(ci, -1); /* Remove trailing comma. */
    return ci;
}

/* Concatenate the slot ownership information to the given SDS string 'ci'.
* If the slot ownership is in a contiguous block, it's represented as start-
end pair,
* else each slot is added separately. */
sds representSlotInfo(sds ci, uint16_t *slot_info_pairs, int
slot_info_pairs_count) {
    for (int i = 0; i < slot_info_pairs_count; i+=2) {
        unsigned long start = slot_info_pairs[i];
        unsigned long end = slot_info_pairs[i+1];
        if (start == end) {
            ci = sdscatfmt(ci, " %i", start);
        } else {
            ci = sdscatfmt(ci, " %i-%i", start, end);
        }
    }
}

```

```

    }
}
return ci;
}

/* Generate a csv-alike representation of the specified cluster node.
 * See clusterGenNodesDescription() top comment for more information.
 *
 * The function returns the string representation as an SDS string. */
sds clusterGenNodeDescription(clusterNode *node, int use_pport) {
    int j, start;
    sds ci;
    int port = use_pport && node->pport ? node->pport : node->port;

    /* Node coordinates */
    ci = sdscatlen(sdsempty(), node->name, CLUSTER_NAMELEN);
    if (sdslens(node->hostname) != 0) {
        ci = sdscatfmt(ci, " %s:%i@%i,%s ",
            node->ip,
            port,
            node->cport,
            node->hostname);
    } else {
        ci = sdscatfmt(ci, " %s:%i@%i ",
            node->ip,
            port,
            node->cport);
    }

    /* Flags */
    ci = representClusterNodeFlags(ci, node->flags);

    /* Slave of... or just "-" */
    ci = sdscatlen(ci, " ", 1);
    if (node->slaveof)
        ci = sdscatlen(ci, node->slaveof->name, CLUSTER_NAMELEN);
    else
        ci = sdscatlen(ci, "-", 1);

    unsigned long long nodeEpoch = node->configEpoch;
    if (nodeIsSlave(node) && node->slaveof) {
        nodeEpoch = node->slaveof->configEpoch;
    }
    /* Latency from the POV of this node, config epoch, link status */
    ci = sdscatfmt(ci, " %I %I %U %s",
        (long long) node->ping_sent,
        (long long) node->pong_received,
        nodeEpoch,
        (node->link || node->flags & CLUSTER_NODE_MYSELF) ?
            "connected" : "disconnected");
}

```

```

/* Slots served by this instance. If we already have slots info,
 * append it directly, otherwise, generate slots only if it has. */
if (node->slot_info_pairs) {
    ci = representSlotInfo(ci, node->slot_info_pairs, node->slot_info_pairs_count);
} else if (node->numslots > 0) {
    start = -1;
    for (j = 0; j < CLUSTER_SLOTS; j++) {
        int bit;

        if ((bit = clusterNodeGetSlotBit(node,j)) != 0) {
            if (start == -1) start = j;
        }
        if (start != -1 && (!bit || j == CLUSTER_SLOTS-1)) {
            if (bit && j == CLUSTER_SLOTS-1) j++;

            if (start == j-1) {
                ci = sdscatfmt(ci," %i",start);
            } else {
                ci = sdscatfmt(ci," %i-%i",start,j-1);
            }
            start = -1;
        }
    }
}

/* Just for MYSELF node we also dump info about slots that
 * we are migrating to other instances or importing from other
 * instances. */
if (node->flags & CLUSTER_NODE_MYSELF) {
    for (j = 0; j < CLUSTER_SLOTS; j++) {
        if (server.cluster->migrating_slots_to[j]) {
            ci = sdscatprintf(ci," [%d->-.40s]",j,
                server.cluster->migrating_slots_to[j]->name);
        } else if (server.cluster->importing_slots_from[j]) {
            ci = sdscatprintf(ci," [%d-<-.40s]",j,
                server.cluster->importing_slots_from[j]->name);
        }
    }
}
return ci;
}

```

```

/* Generate the slot topology for all nodes and store the string representation
 * in the slots_info struct on the node. This is used to improve the efficiency
 * of clusterGenNodesDescription() because it removes looping of the slot space
 * for generating the slot info for each node individually. */

```

```

void clusterGenNodesSlotsInfo(int filter) {
    clusterNode *n = NULL;
    int start = -1;

```

```

for (int i = 0; i <= CLUSTER_SLOTS; i++) {
    /* Find start node and slot id. */
    if (n == NULL) {
        if (i == CLUSTER_SLOTS) break;
        n = server.cluster->slots[i];
        start = i;
        continue;
    }

    /* Generate slots info when occur different node with start
     * or end of slot. */
    if (i == CLUSTER_SLOTS || n != server.cluster->slots[i]) {
        if (!(n->flags & filter)) {
            if (!n->slot_info_pairs) {
                n->slot_info_pairs = zmalloc(2 * n->numslots *
sizeof(uint16_t));
            }
            serverAssert((n->slot_info_pairs_count + 1) < (2 * n-
>numslots));
            n->slot_info_pairs[n->slot_info_pairs_count++] = start;
            n->slot_info_pairs[n->slot_info_pairs_count++] = i-1;
        }
        if (i == CLUSTER_SLOTS) break;
        n = server.cluster->slots[i];
        start = i;
    }
}
}

void clusterFreeNodesSlotsInfo(clusterNode *n) {
    zfree(n->slot_info_pairs);
    n->slot_info_pairs = NULL;
    n->slot_info_pairs_count = 0;
}

/* Generate a csv-alike representation of the nodes we are aware of,
 * including the "myself" node, and return an SDS string containing the
 * representation (it is up to the caller to free it).
 *
 * All the nodes matching at least one of the node flags specified in
 * "filter" are excluded from the output, so using zero as a filter will
 * include all the known nodes in the representation, including nodes in
 * the HANDSHAKE state.
 *
 * Setting use_pport to 1 in a TLS cluster makes the result contain the
 * plaintext client port rather than the TLS client port of each node.
 *
 * The representation obtained using this function is used for the output
 * of the CLUSTER NODES function, and as format for the cluster
 * configuration file (nodes.conf) for a given node. */
sds clusterGenNodesDescription(int filter, int use_pport) {

```



```

sds ci = sdsempty(), ni;
dictIterator *di;
dictEntry *de;

/* Generate all nodes slots info firstly. */
clusterGenNodesSlotsInfo(filter);

di = dictGetSafeIterator(server.cluster->nodes);
while((de = dictNext(di)) != NULL) {
    clusterNode *node = dictGetVal(de);

    if (node->flags & filter) continue;
    ni = clusterGenNodeDescription(node, use_pport);
    ci = sdscatsds(ci,ni);
    sdsfree(ni);
    ci = sdscatlen(ci,"\n",1);

    /* Release slots info. */
    clusterFreeNodesSlotsInfo(node);
}
dictReleaseIterator(di);
return ci;
}

/* Add to the output buffer of the given client the description of the given
cluster link.
 * The description is a map with each entry being an attribute of the link. */
void addReplyClusterLinkDescription(client *c, clusterLink *link) {
    addReplyMapLen(c, 6);

    addReplyBulkCString(c, "direction");
    addReplyBulkCString(c, link->inbound ? "from" : "to");

    /* addReplyClusterLinkDescription is only called for links that have been
     * associated with nodes. The association is always bi-directional, so
     * in addReplyClusterLinkDescription, link->node should never be NULL. */
    serverAssert(link->node);
    sds node_name = sdsnewlen(link->node->name, CLUSTER_NAMELEN);
    addReplyBulkCString(c, "node");
    addReplyBulkCString(c, node_name);
    sdsfree(node_name);

    addReplyBulkCString(c, "create-time");
    addReplyLongLong(c, link->ctime);

    char events[3], *p;
    p = events;
    if (link->conn) {
        if (connHasReadHandler(link->conn)) *p++ = 'r';
        if (connHasWriteHandler(link->conn)) *p++ = 'w';
    }
}

```

```

    *p = '\0';
    addReplyBulkCString(c, "events");
    addReplyBulkCString(c, events);

    addReplyBulkCString(c, "send-buffer-allocated");
    addReplyLongLong(c, sdsalloc(link->sndbuf));

    addReplyBulkCString(c, "send-buffer-used");
    addReplyLongLong(c, sdslen(link->sndbuf));
}

/* Add to the output buffer of the given client an array of cluster link
descriptions,
 * with array entry being a description of a single current cluster link. */
void addReplyClusterLinksDescription(client *c) {
    dictIterator *di;
    dictEntry *de;
    void *arraylen_ptr = NULL;
    int num_links = 0;

    arraylen_ptr = addReplyDeferredLen(c);

    di = dictGetSafeIterator(server.cluster->nodes);
    while((de = dictNext(di)) != NULL) {
        clusterNode *node = dictGetVal(de);
        if (node->link) {
            num_links++;
            addReplyClusterLinkDescription(c, node->link);
        }
        if (node->inbound_link) {
            num_links++;
            addReplyClusterLinkDescription(c, node->inbound_link);
        }
    }
    dictReleaseIterator(di);

    setDeferredArrayLen(c, arraylen_ptr, num_links);
}

/* -----
-
 * CLUSTER command
 * -----
*/

const char *getPreferredEndpoint(clusterNode *n) {
    switch(server.cluster_preferred_endpoint_type) {
        case CLUSTER_ENDPOINT_TYPE_IP: return n->ip;
        case CLUSTER_ENDPOINT_TYPE_HOSTNAME: return (sdslen(n->hostname) != 0) ? n->hostname : "?";
        case CLUSTER_ENDPOINT_TYPE_UNKNOWN_ENDPOINT: return "";
    }
}

```

```

    }
    return "unknown";
}

const char *clusterGetMessageTypeString(int type) {
    switch(type) {
        case CLUSTERMSG_TYPE_PING: return "ping";
        case CLUSTERMSG_TYPE_PONG: return "pong";
        case CLUSTERMSG_TYPE_MEET: return "meet";
        case CLUSTERMSG_TYPE_FAIL: return "fail";
        case CLUSTERMSG_TYPE_PUBLISH: return "publish";
        case CLUSTERMSG_TYPE_PUBLISHSHARD: return "publishshard";
        case CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST: return "auth-req";
        case CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK: return "auth-ack";
        case CLUSTERMSG_TYPE_UPDATE: return "update";
        case CLUSTERMSG_TYPE_MFSTART: return "mfstart";
        case CLUSTERMSG_TYPE_MODULE: return "module";
    }
    return "unknown";
}

int getSlotOrReply(client *c, robj *o) {
    long long slot;

    if (getLongLongFromObject(o,&slot) != C_OK ||
        slot < 0 || slot >= CLUSTER_SLOTS)
    {
        addReplyError(c,"Invalid or out of range slot");
        return -1;
    }
    return (int) slot;
}

/* Returns an indication if the replica node is fully available
 * and should be listed in CLUSTER SLOTS response.
 * Returns 1 for available nodes, 0 for nodes that have
 * not finished their initial sync, in failed state, or are
 * otherwise considered not available to serve read commands. */
static int isReplicaAvailable(clusterNode *node) {
    if (nodeFailed(node)) {
        return 0;
    }
    long long repl_offset = node->repl_offset;
    if (node->flags & CLUSTER_NODE_MYSELF) {
        /* Nodes do not update their own information
         * in the cluster node list. */
        repl_offset = replicationGetSlaveOffset();
    }
    return (repl_offset != 0);
}

```

```

int checkSlotAssignmentsOrReply(client *c, unsigned char *slots, int del, int
start_slot, int end_slot) {
    int slot;
    for (slot = start_slot; slot <= end_slot; slot++) {
        if (del && server.cluster->slots[slot] == NULL) {
            addReplyErrorFormat(c,"Slot %d is already unassigned", slot);
            return C_ERR;
        } else if (!del && server.cluster->slots[slot]) {
            addReplyErrorFormat(c,"Slot %d is already busy", slot);
            return C_ERR;
        }
        if (slots[slot]++ == 1) {
            addReplyErrorFormat(c,"Slot %d specified multiple times",
(int)slot);
            return C_ERR;
        }
    }
    return C_OK;
}

void clusterUpdateSlots(client *c, unsigned char *slots, int del) {
    int j;
    for (j = 0; j < CLUSTER_SLOTS; j++) {
        if (slots[j]) {
            int retval;

            /* If this slot was set as importing we can clear this
             * state as now we are the real owner of the slot. */
            if (server.cluster->importing_slots_from[j])
                server.cluster->importing_slots_from[j] = NULL;

            retval = del ? clusterDelSlot(j) :
                        clusterAddSlot(myself,j);
            serverAssertWithInfo(c,NULL,retval == C_OK);
        }
    }
}

void addNodeToNodeReply(client *c, clusterNode *node) {
    addReplyArrayLen(c, 4);
    if (server.cluster_preferred_endpoint_type == CLUSTER_ENDPOINT_TYPE_IP) {
        addReplyBulkCString(c, node->ip);
    } else if (server.cluster_preferred_endpoint_type ==
CLUSTER_ENDPOINT_TYPE_HOSTNAME) {
        addReplyBulkCString(c, sdslen(node->hostname) != 0 ? node->hostname :
"?");
    } else if (server.cluster_preferred_endpoint_type ==
CLUSTER_ENDPOINT_TYPE_UNKNOWN_ENDPOINT) {
        addReplyNull(c);
    } else {
        serverPanic("Unrecognized preferred endpoint type");
    }
}

```

```

}

/* Report non-TLS ports to non-TLS client in TLS cluster if available. */
int use_pport = (server.tls_cluster &&
                 c->conn && connGetType(c->conn) != CONN_TYPE_TLS);
addReplyLongLong(c, use_pport && node->pport ? node->pport : node->port);
addReplyBulkCBuffer(c, node->name, CLUSTER_NAMELEN);

/* Add the additional endpoint information, this is all the known
networking information
 * that is not the preferred endpoint. */
void *deflen = addReplyDeferredLen(c);
int length = 0;
if (server.cluster_preferred_endpoint_type != CLUSTER_ENDPOINT_TYPE_IP) {
    addReplyBulkCString(c, "ip");
    addReplyBulkCString(c, node->ip);
    length++;
}
if (server.cluster_preferred_endpoint_type !=
CLUSTER_ENDPOINT_TYPE_HOSTNAME
    && sdslen(node->hostname) != 0)
{
    addReplyBulkCString(c, "hostname");
    addReplyBulkCString(c, node->hostname);
    length++;
}
setDeferredMapLen(c, deflen, length);
}

void addNodeReplyForClusterSlot(client *c, clusterNode *node, int start_slot,
int end_slot) {
    int i, nested_elements = 3; /* slots (2) + master addr (1) */
    void *nested_replylen = addReplyDeferredLen(c);
    addReplyLongLong(c, start_slot);
    addReplyLongLong(c, end_slot);
    addNodeToNodeReply(c, node);

    /* Remaining nodes in reply are replicas for slot range */
    for (i = 0; i < node->numslaves; i++) {
        /* This loop is copy/pasted from clusterGenNodeDescription()
        * with modifications for per-slot node aggregation. */
        if (!isReplicaAvailable(node->slaves[i])) continue;
        addNodeToNodeReply(c, node->slaves[i]);
        nested_elements++;
    }
    setDeferredArrayLen(c, nested_replylen, nested_elements);
}

/* Add detailed information of a node to the output buffer of the given client.
*/
void addNodeDetailsToShardReply(client *c, clusterNode *node) {

```

```

int reply_count = 0;
void *node_replylen = addReplyDeferredLen(c);
addReplyBulkCString(c, "id");
addReplyBulkCBuffer(c, node->name, CLUSTER_NAMELEN);
reply_count++;

/* We use server.tls_cluster as a proxy for whether or not
 * the remote port is the tls port or not */
int plaintext_port = server.tls_cluster ? node->pport : node->port;
int tls_port = server.tls_cluster ? node->port : 0;
if (plaintext_port) {
    addReplyBulkCString(c, "port");
    addReplyLongLong(c, plaintext_port);
    reply_count++;
}

if (tls_port) {
    addReplyBulkCString(c, "tls-port");
    addReplyLongLong(c, tls_port);
    reply_count++;
}

addReplyBulkCString(c, "ip");
addReplyBulkCString(c, node->ip);
reply_count++;

addReplyBulkCString(c, "endpoint");
addReplyBulkCString(c, getPreferredEndpoint(node));
reply_count++;

if (node->hostname) {
    addReplyBulkCString(c, "hostname");
    addReplyBulkCString(c, node->hostname);
    reply_count++;
}

long long node_offset;
if (node->flags & CLUSTER_NODE_MYSELF) {
    node_offset = nodeIsSlave(node) ? replicationGetSlaveOffset() :
server.master_repl_offset;
} else {
    node_offset = node->repl_offset;
}

addReplyBulkCString(c, "role");
addReplyBulkCString(c, nodeIsSlave(node) ? "replica" : "master");
reply_count++;

addReplyBulkCString(c, "replication-offset");
addReplyLongLong(c, node_offset);
reply_count++;

```

```

addReplyBulkCString(c, "health");
const char *health_msg = NULL;
if (nodeFailed(node)) {
    health_msg = "fail";
} else if (nodeIsSlave(node) && node_offset == 0) {
    health_msg = "loading";
} else {
    health_msg = "online";
}
addReplyBulkCString(c, health_msg);
reply_count++;

setDeferredMapLen(c, node_replylen, reply_count);
}

```

/* Add the shard reply of a single shard based off the given primary node. */

```

void addShardReplyForClusterShards(client *c, clusterNode *node, uint16_t
*slot_info_pairs, int slot_pairs_count) {
    addReplyMapLen(c, 2);
    addReplyBulkCString(c, "slots");
    if (slot_info_pairs) {
        serverAssert((slot_pairs_count % 2) == 0);
        addReplyArrayLen(c, slot_pairs_count);
        for (int i = 0; i < slot_pairs_count; i++)
            addReplyBulkLongLong(c, (unsigned long)slot_info_pairs[i]);
    } else {
        /* If no slot info pair is provided, the node owns no slots */
        addReplyArrayLen(c, 0);
    }

    addReplyBulkCString(c, "nodes");
    list *nodes_for_slot = clusterGetNodesServingMySlots(node);
    /* At least the provided node should be serving its slots */
    serverAssert(nodes_for_slot);
    addReplyArrayLen(c, listLength(nodes_for_slot));
    if (listLength(nodes_for_slot) != 0) {
        listIter li;
        listNode *ln;
        listRewind(nodes_for_slot, &li);
        while ((ln = listNext(&li))) {
            clusterNode *node = listNodeValue(ln);
            addNodeDetailsToShardReply(c, node);
        }
        listRelease(nodes_for_slot);
    }
}

```

/* Add to the output buffer of the given client, an array of slot (start, end)
* pair owned by the shard, also the primary and set of replica(s) along with
* information about each node. */

```

void clusterReplyShards(client *c) {
    void *shard_replylen = addReplyDeferredLen(c);
    int shard_count = 0;
    /* This call will add slot_info_pairs to all nodes */
    clusterGenNodesSlotsInfo(0);
    dictIterator *di = dictGetSafeIterator(server.cluster->nodes);
    dictEntry *de;
    /* Iterate over all the available nodes in the cluster, for each primary
     * node return generate the cluster shards response. if the primary node
     * doesn't own any slot, cluster shard response contains the node related
     * information and an empty slots array. */
    while((de = dictNext(di)) != NULL) {
        clusterNode *n = dictGetVal(de);
        if (nodeIsSlave(n)) {
            /* You can force a replica to own slots, even though it'll get
            reverted,
             * so freeing the slot pair here just in case. */
            clusterFreeNodesSlotsInfo(n);
            continue;
        }
        shard_count++;
        /* n->slot_info_pairs is set to NULL when the the node owns no slots.
    */
        addShardReplyForClusterShards(c, n, n->slot_info_pairs, n->slot_info_pairs_count);
        clusterFreeNodesSlotsInfo(n);
    }
    dictReleaseIterator(di);
    setDeferredArrayLen(c, shard_replylen, shard_count);
}

void clusterReplyMultiBulkSlots(client * c) {
    /* Format: 1) 1) start slot
     *           2) end slot
     *           3) 1) master IP
     *              2) master port
     *              3) node ID
     *           4) 1) replica IP
     *              2) replica port
     *              3) node ID
     *           ... continued until done
    */
    clusterNode *n = NULL;
    int num_masters = 0, start = -1;
    void *slot_replylen = addReplyDeferredLen(c);

    for (int i = 0; i <= CLUSTER_SLOTS; i++) {
        /* Find start node and slot id. */
        if (n == NULL) {
            if (i == CLUSTER_SLOTS) break;
            n = server.cluster->slots[i];

```



```

        start = i;
        continue;
    }

    /* Add cluster slots info when occur different node with start
    * or end of slot. */
    if (i == CLUSTER_SLOTS || n != server.cluster->slots[i]) {
        addNodeReplyForClusterSlot(c, n, start, i-1);
        num_masters++;
        if (i == CLUSTER_SLOTS) break;
        n = server.cluster->slots[i];
        start = i;
    }
}
setDeferredArrayLen(c, slot_replylen, num_masters);
}

void clusterCommand(client *c) {
    if (server.cluster_enabled == 0) {
        addReplyError(c, "This instance has cluster support disabled");
        return;
    }

    if (c->argc == 2 && !strcasecmp(c->argv[1]->ptr, "help")) {
        const char *help[] = {
"ADDSLOTS <slot> [<slot> ...]",
"    Assign slots to current node.",
"ADDSLOTSRANGE <start slot> <end slot> [<start slot> <end slot> ...]",
"    Assign slots which are between <start-slot> and <end-slot> to current
node.",
"BUMPEPOCH",
"    Advance the cluster config epoch.",
"COUNT-FAILURE-REPORTS <node-id>",
"    Return number of failure reports for <node-id>.",
"COUNTKEYSINSLOT <slot>",
"    Return the number of keys in <slot>.",
"DELSLOTS <slot> [<slot> ...]",
"    Delete slots information from current node.",
"DELSLOTSRANGE <start slot> <end slot> [<start slot> <end slot> ...]",
"    Delete slots information which are between <start-slot> and <end-slot>
from current node.",
"FAILOVER [FORCE|TAKEOVER]",
"    Promote current replica node to being a master.",
"FORGET <node-id>",
"    Remove a node from the cluster.",
"GETKEYSINSLOT <slot> <count>",
"    Return key names stored by current node in a slot.",
"FLUSHSLOTS",
"    Delete current node own slots information.",
"INFO",
"    Return information about the cluster.",

```

```

"KEYSLOT <key>",
"    Return the hash slot for <key>.",
"MEET <ip> <port> [<bus-port>]",
"    Connect nodes into a working cluster.",
"MYID",
"    Return the node id.",
"NODES",
"    Return cluster configuration seen by node. Output format:",
"    <id> <ip:port> <flags> <master> <pings> <pongs> <epoch> <link> <slot>
...",
"REPLICATE <node-id>",
"    Configure current node as replica to <node-id>.",
"RESET [HARD|SOFT]",
"    Reset current node (default: soft).",
"SET-CONFIG-EPOCH <epoch>",
"    Set config epoch of current node.",
"SETSLOT <slot> (IMPORTING <node-id>|MIGRATING <node-id>|STABLE|NODE <node-
id>)",
"    Set slot state.",
"REPLICAS <node-id>",
"    Return <node-id> replicas.",
"SAVECONFIG",
"    Force saving cluster configuration on disk.",
"SLOTS",
"    Return information about slots range mappings. Each range is made of:",
"    start, end, master and replicas IP addresses, ports and ids",
"SHARDS",
"    Return information about slot range mappings and the nodes associated with
them.",
"LINKS",
"    Return information about all network links between this node and its
peers.",
"    Output format is an array where each array element is a map containing
attributes of a link",
NULL
};
addReplyHelp(c, help);
} else if (!strcasecmp(c->argv[1]->ptr,"meet") && (c->argc == 4 || c->argc
== 5)) {
    /* CLUSTER MEET <ip> <port> [cport] */
    long long port, cport;

    if (getLongLongFromObject(c->argv[3], &port) != C_OK) {
        addReplyErrorFormat(c,"Invalid TCP base port specified: %s",
                           (char*)c->argv[3]->ptr);
        return;
    }

    if (c->argc == 5) {
        if (getLongLongFromObject(c->argv[4], &cport) != C_OK) {
            addReplyErrorFormat(c,"Invalid TCP bus port specified: %s",

```

```

        (char*)c->argv[4]->ptr);

    return;
}
} else {
    cport = port + CLUSTER_PORT_INCR;
}

if (clusterStartHandshake(c->argv[2]->ptr,port,cport) == 0 &&
    errno == EINVAL)
{
    addReplyErrorFormat(c,"Invalid node address specified: %s:%s",
        (char*)c->argv[2]->ptr, (char*)c->argv[3]->ptr);
} else {
    addReply(c,shared.ok);
}
} else if (!strcasecmp(c->argv[1]->ptr,"nodes") && c->argc == 2) {
    /* CLUSTER NODES */
    /* Report plaintext ports, only if cluster is TLS but client is known
to
    * be non-TLS). */
    int use_pport = (server.tls_cluster &&
        c->conn && connGetType(c->conn) != CONN_TYPE_TLS);
    sds nodes = clusterGenNodesDescription(0, use_pport);
    addReplyVerbatim(c,nodes,sdslen(nodes),"txt");
    sdsfree(nodes);
} else if (!strcasecmp(c->argv[1]->ptr,"myid") && c->argc == 2) {
    /* CLUSTER MYID */
    addReplyBulkCBuffer(c,myself->name, CLUSTER_NAMELEN);
} else if (!strcasecmp(c->argv[1]->ptr,"slots") && c->argc == 2) {
    /* CLUSTER SLOTS */
    clusterReplyMultiBulkSlots(c);
} else if (!strcasecmp(c->argv[1]->ptr,"shards") && c->argc == 2) {
    /* CLUSTER SHARDS */
    clusterReplyShards(c);
} else if (!strcasecmp(c->argv[1]->ptr,"flushslots") && c->argc == 2) {
    /* CLUSTER FLUSHSLOTS */
    if (dictSize(server.db[0].dict) != 0) {
        addReplyError(c,"DB must be empty to perform CLUSTER FLUSHSLOTS.");
        return;
    }
    clusterDelNodeSlots(myself);

clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|CLUSTER_TODO_SAVE_CONFIG);
    addReply(c,shared.ok);
} else if ((!strcasecmp(c->argv[1]->ptr,"addslots") ||
    !strcasecmp(c->argv[1]->ptr,"delslots")) && c->argc >= 3)
{
    /* CLUSTER ADDSLOTS <slot> [slot] ... */
    /* CLUSTER DELSLOTS <slot> [slot] ... */
    int j, slot;
    unsigned char *slots = zmalloc(CLUSTER_SLOTS);

```

```

    int del = !strcasecmp(c->argv[1]->ptr,"delslots");

    memset(slots,0,CLUSTER_SLOTS);
    /* Check that all the arguments are parseable.*/
    for (j = 2; j < c->argc; j++) {
        if ((slot = getSlotOrReply(c,c->argv[j])) == C_ERR) {
            zfree(slots);
            return;
        }
    }
    /* Check that the slots are not already busy. */
    for (j = 2; j < c->argc; j++) {
        slot = getSlotOrReply(c,c->argv[j]);
        if (checkSlotAssignmentsOrReply(c, slots, del, slot, slot) ==
C_ERR) {
            zfree(slots);
            return;
        }
    }
    clusterUpdateSlots(c, slots, del);
    zfree(slots);

clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|CLUSTER_TODO_SAVE_CONFIG);
    addReply(c,shared.ok);
} else if ((!strcasecmp(c->argv[1]->ptr,"addslotsrange") ||
    !strcasecmp(c->argv[1]->ptr,"delslotsrange")) && c->argc >= 4) {
    if (c->argc % 2 == 1) {
        addReplyErrorArity(c);
        return;
    }
    /* CLUSTER ADDSLOTSRANGE <start slot> <end slot> [<start slot> <end
slot> ...] */
    /* CLUSTER DELSLOTSRANGE <start slot> <end slot> [<start slot> <end
slot> ...] */
    int j, startslot, endslot;
    unsigned char *slots = zmalloc(CLUSTER_SLOTS);
    int del = !strcasecmp(c->argv[1]->ptr,"delslotsrange");

    memset(slots,0,CLUSTER_SLOTS);
    /* Check that all the arguments are parseable and that all the
    * slots are not already busy. */
    for (j = 2; j < c->argc; j += 2) {
        if ((startslot = getSlotOrReply(c,c->argv[j])) == C_ERR) {
            zfree(slots);
            return;
        }
        if ((endslot = getSlotOrReply(c,c->argv[j+1])) == C_ERR) {
            zfree(slots);
            return;
        }
        if (startslot > endslot) {

```

```

        addReplyErrorFormat(c,"start slot number %d is greater than end
slot number %d", startslot, endslot);
        zfree(slots);
        return;
    }

    if (checkSlotAssignmentsOrReply(c, slots, del, startslot, endslot)
== C_ERR) {
        zfree(slots);
        return;
    }
}
clusterUpdateSlots(c, slots, del);
zfree(slots);

clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|CLUSTER_TODO_SAVE_CONFIG);
addReply(c,shared.ok);
} else if (!strcasecmp(c->argv[1]->ptr,"setslot") && c->argc >= 4) {
    /* SETSLOT 10 MIGRATING <node ID> */
    /* SETSLOT 10 IMPORTING <node ID> */
    /* SETSLOT 10 STABLE */
    /* SETSLOT 10 NODE <node ID> */
    int slot;
    clusterNode *n;

    if (nodeIsSlave(myself)) {
        addReplyError(c,"Please use SETSLOT only with masters.");
        return;
    }

    if ((slot = getSlotOrReply(c,c->argv[2])) == -1) return;

    if (!strcasecmp(c->argv[3]->ptr,"migrating") && c->argc == 5) {
        if (server.cluster->slots[slot] != myself) {
            addReplyErrorFormat(c,"I'm not the owner of hash slot
%u",slot);
            return;
        }
        n = clusterLookupNode(c->argv[4]->ptr, sdslen(c->argv[4]->ptr));
        if (n == NULL) {
            addReplyErrorFormat(c,"I don't know about node %s",
(char*)c->argv[4]->ptr);
            return;
        }
        if (nodeIsSlave(n)) {
            addReplyError(c,"Target node is not a master");
            return;
        }
        server.cluster->migrating_slots_to[slot] = n;
    } else if (!strcasecmp(c->argv[3]->ptr,"importing") && c->argc == 5) {
        if (server.cluster->slots[slot] == myself) {

```

```

        addReplyErrorFormat(c,
            "I'm already the owner of hash slot %u",slot);
        return;
    }
    n = clusterLookupNode(c->argv[4]->ptr, sdslen(c->argv[4]->ptr));
    if (n == NULL) {
        addReplyErrorFormat(c,"I don't know about node %s",
            (char*)c->argv[4]->ptr);
        return;
    }
    if (nodeIsSlave(n)) {
        addReplyError(c,"Target node is not a master");
        return;
    }
    server.cluster->importing_slots_from[slot] = n;
} else if (!strcasecmp(c->argv[3]->ptr,"stable") && c->argc == 4) {
    /* CLUSTER SETSLOT <SLOT> STABLE */
    server.cluster->importing_slots_from[slot] = NULL;
    server.cluster->migrating_slots_to[slot] = NULL;
} else if (!strcasecmp(c->argv[3]->ptr,"node") && c->argc == 5) {
    /* CLUSTER SETSLOT <SLOT> NODE <NODE ID> */
    n = clusterLookupNode(c->argv[4]->ptr, sdslen(c->argv[4]->ptr));
    if (!n) {
        addReplyErrorFormat(c,"Unknown node %s",
            (char*)c->argv[4]->ptr);
        return;
    }
    if (nodeIsSlave(n)) {
        addReplyError(c,"Target node is not a master");
        return;
    }
}
/* If this hash slot was served by 'myself' before to switch
 * make sure there are no longer local keys for this hash slot. */
if (server.cluster->slots[slot] == myself && n != myself) {
    if (countKeysInSlot(slot) != 0) {
        addReplyErrorFormat(c,
            "Can't assign hashslot %d to a different node "
            "while I still hold keys for this hash slot.", slot);
        return;
    }
}

/* If this slot is in migrating status but we have no keys
 * for it assigning the slot to another node will clear
 * the migrating status. */
if (countKeysInSlot(slot) == 0 &&
    server.cluster->migrating_slots_to[slot])
    server.cluster->migrating_slots_to[slot] = NULL;

int slot_was_mine = server.cluster->slots[slot] == myself;
clusterDelSlot(slot);
clusterAddSlot(n,slot);

```

```

/* If we are a master left without slots, we should turn into a
 * replica of the new master. */
if (slot_was_mine &&
    n != myself &&
    myself->numslots == 0 &&
    server.cluster_allow_replica_migration)
{
    serverLog(LL_WARNING,
        "Configuration change detected. Reconfiguring myself
"
        "as a replica of %.40s", n->name);
    clusterSetMaster(n);
    clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG |
                        CLUSTER_TODO_UPDATE_STATE |
                        CLUSTER_TODO_FSYNC_CONFIG);
}

/* If this node was importing this slot, assigning the slot to
 * itself also clears the importing status. */
if (n == myself &&
    server.cluster->importing_slots_from[slot])
{
    /* This slot was manually migrated, set this node configEpoch
     * to a new epoch so that the new version can be propagated
     * by the cluster.
     *
     * Note that if this ever results in a collision with another
     * node getting the same configEpoch, for example because a
     * failover happens at the same time we close the slot, the
     * configEpoch collision resolution will fix it assigning
     * a different epoch to each node. */
    if (clusterBumpConfigEpochWithoutConsensus() == C_OK) {
        serverLog(LL_WARNING,
            "configEpoch updated after importing slot %d", slot);
    }
    server.cluster->importing_slots_from[slot] = NULL;
    /* After importing this slot, let the other nodes know as
     * soon as possible. */
    clusterBroadcastPong(CLUSTER_BROADCAST_ALL);
}
} else {
    addReplyError(c,
        "Invalid CLUSTER SETSLOT action or number of arguments. Try
CLUSTER HELP");
    return;
}

clusterDoBeforeSleep(CLUSTER_TODO_SAVE_CONFIG|CLUSTER_TODO_UPDATE_STATE);
addReply(c,shared.ok);
} else if (!strcasecmp(c->argv[1]->ptr,"bumpepoch") && c->argc == 2) {

```

```

/* CLUSTER BUMPEPOCH */
int retval = clusterBumpConfigEpochWithoutConsensus();
sds reply = sdscatprintf(sdsempty(), "+%s %llu\r\n",
    (retval == C_OK) ? "BUMPED" : "STILL",
    (unsigned long long) myself->configEpoch);
addReplySds(c, reply);
} else if (!strcasecmp(c->argv[1]->ptr, "info") && c->argc == 2) {
/* CLUSTER INFO */
char *statestr[] = {"ok", "fail"};
int slots_assigned = 0, slots_ok = 0, slots_pfail = 0, slots_fail = 0;
uint64_t myepoch;
int j;

for (j = 0; j < CLUSTER_SLOTS; j++) {
    clusterNode *n = server.cluster->slots[j];

    if (n == NULL) continue;
    slots_assigned++;
    if (nodeFailed(n)) {
        slots_fail++;
    } else if (nodeTimedOut(n)) {
        slots_pfail++;
    } else {
        slots_ok++;
    }
}

myepoch = (nodeIsSlave(myself) && myself->slaveof) ?
    myself->slaveof->configEpoch : myself->configEpoch;

sds info = sdscatprintf(sdsempty(),
    "cluster_state:%s\r\n"
    "cluster_slots_assigned:%d\r\n"
    "cluster_slots_ok:%d\r\n"
    "cluster_slots_pfail:%d\r\n"
    "cluster_slots_fail:%d\r\n"
    "cluster_known_nodes:%lu\r\n"
    "cluster_size:%d\r\n"
    "cluster_current_epoch:%llu\r\n"
    "cluster_my_epoch:%llu\r\n"
    , statestr[server.cluster->state],
    slots_assigned,
    slots_ok,
    slots_pfail,
    slots_fail,
    dictSize(server.cluster->nodes),
    server.cluster->size,
    (unsigned long long) server.cluster->currentEpoch,
    (unsigned long long) myepoch
);

```



```

/* Show stats about messages sent and received. */
long long tot_msg_sent = 0;
long long tot_msg_received = 0;

for (int i = 0; i < CLUSTERMSG_TYPE_COUNT; i++) {
    if (server.cluster->stats_bus_messages_sent[i] == 0) continue;
    tot_msg_sent += server.cluster->stats_bus_messages_sent[i];
    info = sdscatprintf(info,
        "cluster_stats_messages_%s_sent:%lld\r\n",
        clusterGetMessageTypeString(i),
        server.cluster->stats_bus_messages_sent[i]);
}
info = sdscatprintf(info,
    "cluster_stats_messages_sent:%lld\r\n", tot_msg_sent);

for (int i = 0; i < CLUSTERMSG_TYPE_COUNT; i++) {
    if (server.cluster->stats_bus_messages_received[i] == 0) continue;
    tot_msg_received += server.cluster->stats_bus_messages_received[i];
    info = sdscatprintf(info,
        "cluster_stats_messages_%s_received:%lld\r\n",
        clusterGetMessageTypeString(i),
        server.cluster->stats_bus_messages_received[i]);
}
info = sdscatprintf(info,
    "cluster_stats_messages_received:%lld\r\n", tot_msg_received);

info = sdscatprintf(info,
    "total_cluster_links_buffer_limit_exceeded:%llu\r\n",
    server.cluster->stat_cluster_links_buffer_limit_exceeded);

/* Produce the reply protocol. */
addReplyVerbatim(c,info,sdslen(info),"txt");
sdsfree(info);
} else if (!strcasecmp(c->argv[1]->ptr,"saveconfig") && c->argc == 2) {
    int retval = clusterSaveConfig(1);

    if (retval == 0)
        addReply(c,shared.ok);
    else
        addReplyErrorFormat(c,"error saving the cluster node config: %s",
            strerror(errno));
} else if (!strcasecmp(c->argv[1]->ptr,"keyslot") && c->argc == 3) {
    /* CLUSTER KEYSLOT <key> */
    sds key = c->argv[2]->ptr;

    addReplyLongLong(c,keyHashSlot(key,sdslen(key)));
} else if (!strcasecmp(c->argv[1]->ptr,"countkeysinslot") && c->argc == 3)
{
    /* CLUSTER COUNTKEYSINSLOT <slot> */
    long long slot;

```

```

    if (getLongLongFromObjectOrReply(c,c->argv[2],&slot,NULL) != C_OK)
        return;
    if (slot < 0 || slot >= CLUSTER_SLOTS) {
        addReplyError(c,"Invalid slot");
        return;
    }
    addReplyLongLong(c,countKeysInSlot(slot));
} else if (!strcasecmp(c->argv[1]->ptr,"getkeysinslot") && c->argc == 4) {
    /* CLUSTER GETKEYSINSLOT <slot> <count> */
    long long maxkeys, slot;

    if (getLongLongFromObjectOrReply(c,c->argv[2],&slot,NULL) != C_OK)
        return;
    if (getLongLongFromObjectOrReply(c,c->argv[3],&maxkeys,NULL)
        != C_OK)
        return;
    if (slot < 0 || slot >= CLUSTER_SLOTS || maxkeys < 0) {
        addReplyError(c,"Invalid slot or number of keys");
        return;
    }

    unsigned int keys_in_slot = countKeysInSlot(slot);
    unsigned int numkeys = maxkeys > keys_in_slot ? keys_in_slot : maxkeys;
    addReplyArrayLen(c,numkeys);
    dictEntry *de = (*server.db->slots_to_keys).by_slot[slot].head;
    for (unsigned int j = 0; j < numkeys; j++) {
        serverAssert(de != NULL);
        sds sdskey = dictGetKey(de);
        addReplyBulkCBuffer(c, sdskey, sdslen(sdskey));
        de = dictEntryNextInSlot(de);
    }
} else if (!strcasecmp(c->argv[1]->ptr,"forget") && c->argc == 3) {
    /* CLUSTER FORGET <NODE ID> */
    clusterNode *n = clusterLookupNode(c->argv[2]->ptr, sdslen(c->argv[2]-
>ptr));
    if (!n) {
        addReplyErrorFormat(c,"Unknown node %s", (char*)c->argv[2]->ptr);
        return;
    } else if (n == myself) {
        addReplyError(c,"I tried hard but I can't forget myself...");
        return;
    } else if (nodeIsSlave(myself) && myself->slaveof == n) {
        addReplyError(c,"Can't forget my master!");
        return;
    }
    clusterBlacklistAddNode(n);
    clusterDelNode(n);
    clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|
        CLUSTER_TODO_SAVE_CONFIG);
    addReply(c,shared.ok);
} else if (!strcasecmp(c->argv[1]->ptr,"replicate") && c->argc == 3) {

```

```

/* CLUSTER REPLICATE <NODE ID> */
/* Lookup the specified node in our table. */
clusterNode *n = clusterLookupNode(c->argv[2]->ptr, sdslen(c->argv[2]-
>ptr));
if (!n) {
    addReplyErrorFormat(c,"Unknown node %s", (char*)c->argv[2]->ptr);
    return;
}

/* I can't replicate myself. */
if (n == myself) {
    addReplyError(c,"Can't replicate myself");
    return;
}

/* Can't replicate a slave. */
if (nodeIsSlave(n)) {
    addReplyError(c,"I can only replicate a master, not a replica.");
    return;
}

/* If the instance is currently a master, it should have no assigned
 * slots nor keys to accept to replicate some other node.
 * Slaves can switch to another master without issues. */
if (nodeIsMaster(myself) &&
    (myself->numslots != 0 || dictSize(server.db[0].dict) != 0)) {
    addReplyError(c,
        "To set a master the node must be empty and "
        "without assigned slots.");
    return;
}

/* Set the master. */
clusterSetMaster(n);

clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|CLUSTER_TODO_SAVE_CONFIG);
addReply(c,shared.ok);
} else if ((!strcasecmp(c->argv[1]->ptr,"slaves") ||
    !strcasecmp(c->argv[1]->ptr,"replicas")) && c->argc == 3) {
    /* CLUSTER SLAVES <NODE ID> */
    clusterNode *n = clusterLookupNode(c->argv[2]->ptr, sdslen(c->argv[2]-
>ptr));
    int j;

    /* Lookup the specified node in our table. */
    if (!n) {
        addReplyErrorFormat(c,"Unknown node %s", (char*)c->argv[2]->ptr);
        return;
    }

    if (nodeIsSlave(n)) {

```

```

        addReplyError(c,"The specified node is not a master");
        return;
    }

    /* Use plaintext port if cluster is TLS but client is non-TLS. */
    int use_pport = (server.tls_cluster &&
                     c->conn && connGetType(c->conn) != CONN_TYPE_TLS);
    addReplyArrayLen(c,n->numslaves);
    for (j = 0; j < n->numslaves; j++) {
        sds ni = clusterGenNodeDescription(n->slaves[j], use_pport);
        addReplyBulkCString(c,ni);
        sdsfree(ni);
    }
} else if (!strcasecmp(c->argv[1]->ptr,"count-failure-reports") &&
           c->argc == 3)
{
    /* CLUSTER COUNT-FAILURE-REPORTS <NODE ID> */
    clusterNode *n = clusterLookupNode(c->argv[2]->ptr, sdslen(c->argv[2]-
>ptr));

    if (!n) {
        addReplyErrorFormat(c,"Unknown node %s", (char*)c->argv[2]->ptr);
        return;
    } else {
        addReplyLongLong(c,clusterNodeFailureReportsCount(n));
    }
} else if (!strcasecmp(c->argv[1]->ptr,"failover") &&
           (c->argc == 2 || c->argc == 3))
{
    /* CLUSTER FAILOVER [FORCE|TAKEOVER] */
    int force = 0, takeover = 0;

    if (c->argc == 3) {
        if (!strcasecmp(c->argv[2]->ptr,"force")) {
            force = 1;
        } else if (!strcasecmp(c->argv[2]->ptr,"takeover")) {
            takeover = 1;
            force = 1; /* Takeover also implies force. */
        } else {
            addReplyErrorObject(c,shared.syntaxerr);
            return;
        }
    }

    /* Check preconditions. */
    if (nodeIsMaster(myself)) {
        addReplyError(c,"You should send CLUSTER FAILOVER to a replica");
        return;
    } else if (myself->slaveof == NULL) {
        addReplyError(c,"I'm a replica but my master is unknown to me");
        return;
    }
}

```

```

} else if (!force &&
           (nodeFailed(myself->slaveof) ||
            myself->slaveof->link == NULL))
{
    addReplyError(c,"Master is down or failed, "
                  "please use CLUSTER FAILOVER FORCE");
    return;
}
resetManualFailover();
server.cluster->mf_end = mstime() + CLUSTER_MF_TIMEOUT;

if (takeover) {
    /* A takeover does not perform any initial check. It just
     * generates a new configuration epoch for this node without
     * consensus, claims the master's slots, and broadcast the new
     * configuration. */
    serverLog(LL_WARNING,"Taking over the master (user request).");
    clusterBumpConfigEpochWithoutConsensus();
    clusterFailoverReplaceYourMaster();
} else if (force) {
    /* If this is a forced failover, we don't need to talk with our
     * master to agree about the offset. We just failover taking over
     * it without coordination. */
    serverLog(LL_WARNING,"Forced failover user request accepted.");
    server.cluster->mf_can_start = 1;
} else {
    serverLog(LL_WARNING,"Manual failover user request accepted.");
    clusterSendMFStart(myself->slaveof);
}
addReply(c,shared.ok);
} else if (!strcasecmp(c->argv[1]->ptr,"set-config-epoch") && c->argc == 3)
{
    /* CLUSTER SET-CONFIG-EPOCH <epoch>
     *
     * The user is allowed to set the config epoch only when a node is
     * totally fresh: no config epoch, no other known node, and so forth.
     * This happens at cluster creation time to start with a cluster where
     * every node has a different node ID, without to rely on the conflicts
     * resolution system which is too slow when a big cluster is created.
    */
    long long epoch;

    if (getLongLongFromObjectOrReply(c,c->argv[2],&epoch,NULL) != C_OK)
        return;

    if (epoch < 0) {
        addReplyErrorFormat(c,"Invalid config epoch specified:
%lld",epoch);
    } else if (dictSize(server.cluster->nodes) > 1) {
        addReplyError(c,"The user can assign a config epoch only when the "
                        "node does not know any other node.");
    }
}

```

```

    } else if (myself->configEpoch != 0) {
        addReplyError(c,"Node config epoch is already non-zero");
    } else {
        myself->configEpoch = epoch;
        serverLog(LL_WARNING,
            "configEpoch set to %llu via CLUSTER SET-CONFIG-EPOCH",
            (unsigned long long) myself->configEpoch);

        if (server.cluster->currentEpoch < (uint64_t)epoch)
            server.cluster->currentEpoch = epoch;
        /* No need to fsync the config here since in the unlucky event
         * of a failure to persist the config, the conflict resolution code
         * will assign a unique config to this node. */
        clusterDoBeforeSleep(CLUSTER_TODO_UPDATE_STATE|
            CLUSTER_TODO_SAVE_CONFIG);
        addReply(c,shared.ok);
    }
} else if (!strcasecmp(c->argv[1]->ptr,"reset") &&
    (c->argc == 2 || c->argc == 3))
{
    /* CLUSTER RESET [SOFT|HARD] */
    int hard = 0;

    /* Parse soft/hard argument. Default is soft. */
    if (c->argc == 3) {
        if (!strcasecmp(c->argv[2]->ptr,"hard")) {
            hard = 1;
        } else if (!strcasecmp(c->argv[2]->ptr,"soft")) {
            hard = 0;
        } else {
            addReplyErrorObject(c,shared.syntaxerr);
            return;
        }
    }

    /* Slaves can be reset while containing data, but not master nodes
     * that must be empty. */
    if (nodeIsMaster(myself) && dictSize(c->db->dict) != 0) {
        addReplyError(c,"CLUSTER RESET can't be called with "
            "master nodes containing keys");
        return;
    }
    clusterReset(hard);
    addReply(c,shared.ok);
} else if (!strcasecmp(c->argv[1]->ptr,"links") && c->argc == 2) {
    /* CLUSTER LINKS */
    addReplyClusterLinksDescription(c);
} else {
    addReplySubcommandSyntaxError(c);
    return;
}

```

```

}

void removeChannelsInSlot(unsigned int slot) {
    unsigned int channelcount = countChannelsInSlot(slot);
    if (channelcount == 0) return;

    /* Retrieve all the channels for the slot. */
    robj **channels = zmalloc(sizeof(robj*)*channelcount);
    raxIterator iter;
    int j = 0;
    unsigned char indexed[2];

    indexed[0] = (slot >> 8) & 0xff;
    indexed[1] = slot & 0xff;
    raxStart(&iter, server.cluster->slots_to_channels);
    raxSeek(&iter, ">=", indexed, 2);
    while(raxNext(&iter)) {
        if (iter.key[0] != indexed[0] || iter.key[1] != indexed[1]) break;
        channels[j++] = createStringObject((char*)iter.key + 2, iter.key_len -
2);
    }
    raxStop(&iter);

    pubsubUnsubscribeShardChannels(channels, channelcount);
    zfree(channels);
}

/* -----
-
* DUMP, RESTORE and MIGRATE commands
* -----
*/

/* Generates a DUMP-format representation of the object 'o', adding it to the
 * io stream pointed by 'rio'. This function can't fail. */
void createDumpPayload(rio *payload, robj *o, robj *key, int dbid) {
    unsigned char buf[2];
    uint64_t crc;

    /* Serialize the object in an RDB-like format. It consist of an object type
     * byte followed by the serialized object. This is understood by RESTORE.
    */
    rioInitWithBuffer(payload, sdsempty());
    serverAssert(rdbSaveObjectType(payload, o));
    serverAssert(rdbSaveObject(payload, o, key, dbid));

    /* Write the footer, this is how it looks like:
     * -----+-----+-----+
     * ... RDB payload | 2 bytes RDB version | 8 bytes CRC64 |
     * -----+-----+-----+
     * RDB version and CRC are both in little endian.
    */

```

```

    */

    /* RDB version */
    buf[0] = RDB_VERSION & 0xff;
    buf[1] = (RDB_VERSION >> 8) & 0xff;
    payload->io.buffer.ptr = sdscatlen(payload->io.buffer.ptr,buf,2);

    /* CRC64 */
    crc = crc64(0,(unsigned char*)payload->io.buffer.ptr,
                sdslen(payload->io.buffer.ptr));
    memrev64ifbe(&crc);
    payload->io.buffer.ptr = sdscatlen(payload->io.buffer.ptr,&crc,8);
}

/* Verify that the RDB version of the dump payload matches the one of this
Redis
* instance and that the checksum is ok.
* If the DUMP payload looks valid C_OK is returned, otherwise C_ERR
* is returned. If rdbver_ptr is not NULL, its populated with the value read
* from the input buffer. */
int verifyDumpPayload(unsigned char *p, size_t len, uint16_t *rdbver_ptr) {
    unsigned char *footer;
    uint16_t rdbver;
    uint64_t crc;

    /* At least 2 bytes of RDB version and 8 of CRC64 should be present. */
    if (len < 10) return C_ERR;
    footer = p+(len-10);

    /* Set and verify RDB version. */
    rdbver = (footer[1] << 8) | footer[0];
    if (rdbver_ptr) {
        *rdbver_ptr = rdbver;
    }
    if (rdbver > RDB_VERSION) return C_ERR;

    if (server.skip_checksum_validation)
        return C_OK;

    /* Verify CRC64 */
    crc = crc64(0,p,len-8);
    memrev64ifbe(&crc);
    return (memcmp(&crc,footer+2,8) == 0) ? C_OK : C_ERR;
}

/* DUMP keyname
* DUMP is actually not used by Redis Cluster but it is the obvious
* complement of RESTORE and can be useful for different applications. */
void dumpCommand(client *c) {
    robj *o;
    rio payload;

```



```

/* Check if the key is here. */
if ((o = lookupKeyRead(c->db,c->argv[1])) == NULL) {
    addReplyNull(c);
    return;
}

/* Create the DUMP encoded representation. */
createDumpPayload(&payload,o,c->argv[1],c->db->id);

/* Transfer to the client */
addReplyBulkSds(c,payload.io.buffer.ptr);
return;
}

/* RESTORE key ttl serialized-value [REPLACE] [ABSTTL] [IDLETIME seconds] [FREQ
frequency] */
void restoreCommand(client *c) {
    long long ttl, lfu_freq = -1, lru_idle = -1, lru_clock = -1;
    rio payload;
    int j, type, replace = 0, absttl = 0;
    robj *obj;

    /* Parse additional options */
    for (j = 4; j < c->argc; j++) {
        int additional = c->argc-j-1;
        if (!strcasecmp(c->argv[j]->ptr,"replace")) {
            replace = 1;
        } else if (!strcasecmp(c->argv[j]->ptr,"absttl")) {
            absttl = 1;
        } else if (!strcasecmp(c->argv[j]->ptr,"idletime") && additional >= 1
&&
                lfu_freq == -1)
        {
            if (getLongLongFromObjectOrReply(c,c->argv[j+1],&lru_idle,NULL)
                != C_OK) return;
            if (lru_idle < 0) {
                addReplyError(c,"Invalid IDLETIME value, must be >= 0");
                return;
            }
            lru_clock = LRU_CLOCK();
            j++; /* Consume additional arg. */
        } else if (!strcasecmp(c->argv[j]->ptr,"freq") && additional >= 1 &&
                lru_idle == -1)
        {
            if (getLongLongFromObjectOrReply(c,c->argv[j+1],&lfu_freq,NULL)
                != C_OK) return;
            if (lfu_freq < 0 || lfu_freq > 255) {
                addReplyError(c,"Invalid FREQ value, must be >= 0 and <= 255");
                return;
            }
        }
    }
}

```

```

        j++; /* Consume additional arg. */
    } else {
        addReplyErrorObject(c,shared.syntaxerr);
        return;
    }
}

/* Make sure this key does not already exist here... */
robject *key = c->argv[1];
if (!replace && lookupKeyWrite(c->db,key) != NULL) {
    addReplyErrorObject(c,shared.busykeyerr);
    return;
}

/* Check if the TTL value makes sense */
if (getLongLongFromObjectOrReply(c,c->argv[2],&ttl,NULL) != C_OK) {
    return;
} else if (ttl < 0) {
    addReplyError(c,"Invalid TTL value, must be >= 0");
    return;
}

/* Verify RDB version and data checksum. */
if (verifyDumpPayload(c->argv[3]->ptr,sdslen(c->argv[3]->ptr),NULL) ==
C_ERR)
{
    addReplyError(c,"DUMP payload version or checksum are wrong");
    return;
}

rioInitWithBuffer(&payload,c->argv[3]->ptr);
if (((type = rdbLoadObjectType(&payload)) == -1) ||
    ((obj = rdbLoadObject(type,&payload,key->ptr,c->db->id,NULL)) == NULL))
{
    addReplyError(c,"Bad data format");
    return;
}

/* Remove the old key if needed. */
int deleted = 0;
if (replace)
    deleted = dbDelete(c->db,key);

if (ttl && !absttl) ttl+=mstime();
if (ttl && checkAlreadyExpired(ttl)) {
    if (deleted) {
        rewriteClientCommandVector(c,2,shared.del,key);
        signalModifiedKey(c,c->db,key);
        notifyKeyspaceEvent(NOTIFY_GENERIC,"del",key,c->db->id);
        server.dirty++;
    }
}

```

```

        decrRefCount(obj);
        addReply(c, shared.ok);
        return;
    }

    /* Create the key and set the TTL if any */
    dbAdd(c->db, key, obj);
    if (ttl) {
        setExpire(c, c->db, key, ttl);
        if (!absttl) {
            /* Propagate TTL as absolute timestamp */
            robj *ttl_obj = createStringObjectFromLongLong(ttl);
            rewriteClientCommandArgument(c, 2, ttl_obj);
            decrRefCount(ttl_obj);
            rewriteClientCommandArgument(c, c->argc, shared.absttl);
        }
    }
    objectSetLRUorLFU(obj, lfu_freq, lru_idle, lru_clock, 1000);
    signalModifiedKey(c, c->db, key);
    notifyKeyspaceEvent(NOTIFY_GENERIC, "restore", key, c->db->id);
    addReply(c, shared.ok);
    server.dirty++;
}

/* MIGRATE socket cache implementation.
 *
 * We take a map between host:ip and a TCP socket that we used to connect
 * to this instance in recent time.
 * This sockets are closed when the max number we cache is reached, and also
 * in serverCron() when they are around for more than a few seconds. */
#define MIGRATE_SOCKET_CACHE_ITEMS 64 /* max num of items in the cache. */
#define MIGRATE_SOCKET_CACHE_TTL 10 /* close cached sockets after 10 sec. */

typedef struct migrateCachedSocket {
    connection *conn;
    long last_dbid;
    time_t last_use_time;
} migrateCachedSocket;

/* Return a migrateCachedSocket containing a TCP socket connected with the
 * target instance, possibly returning a cached one.
 *
 * This function is responsible of sending errors to the client if a
 * connection can't be established. In this case -1 is returned.
 * Otherwise on success the socket is returned, and the caller should not
 * attempt to free it after usage.
 *
 * If the caller detects an error while using the socket, migrateCloseSocket()
 * should be called so that the connection will be created from scratch
 * the next time. */
migrateCachedSocket* migrateGetSocket(client *c, robj *host, robj *port, long

```

```

timeout) {
    connection *conn;
    sds name = sdsempty();
    migrateCachedSocket *cs;

    /* Check if we have an already cached socket for this ip:port pair. */
    name = sdscatlen(name,host->ptr,sdslen(host->ptr));
    name = sdscatlen(name,":",1);
    name = sdscatlen(name,port->ptr,sdslen(port->ptr));
    cs = dictFetchValue(server.migrate_cached_sockets,name);
    if (cs) {
        sdsfree(name);
        cs->last_use_time = server.unixtime;
        return cs;
    }

    /* No cached socket, create one. */
    if (dictSize(server.migrate_cached_sockets) == MIGRATE_SOCKET_CACHE_ITEMS)
    {
        /* Too many items, drop one at random. */
        dictEntry *de = dictGetRandomKey(server.migrate_cached_sockets);
        cs = dictGetVal(de);
        connClose(cs->conn);
        zfree(cs);
        dictDelete(server.migrate_cached_sockets,dictGetKey(de));
    }

    /* Create the socket */
    conn = server.tls_cluster ? connCreateTLS() : connCreateSocket();
    if (connBlockingConnect(conn, c->argv[1]->ptr, atoi(c->argv[2]->ptr),
timeout)
        != C_OK) {
        addReplyError(c,"-IOERR error or timeout connecting to the client");
        connClose(conn);
        sdsfree(name);
        return NULL;
    }
    connEnableTcpNoDelay(conn);

    /* Add to the cache and return it to the caller. */
    cs = zmalloc(sizeof(*cs));
    cs->conn = conn;

    cs->last_dbid = -1;
    cs->last_use_time = server.unixtime;
    dictAdd(server.migrate_cached_sockets,name,cs);
    return cs;
}

/* Free a migrate cached connection. */
void migrateCloseSocket(robj *host, robj *port) {

```

```

sds name = sdsempty();
migrateCachedSocket *cs;

name = sdscatlen(name, host->ptr, sdslen(host->ptr));
name = sdscatlen(name, ":", 1);
name = sdscatlen(name, port->ptr, sdslen(port->ptr));
cs = dictFetchValue(server.migrate_cached_sockets, name);
if (!cs) {
    sdsfree(name);
    return;
}

connClose(cs->conn);
zfree(cs);
dictDelete(server.migrate_cached_sockets, name);
sdsfree(name);
}

void migrateCloseTimedoutSockets(void) {
    dictIterator *di = dictGetSafeIterator(server.migrate_cached_sockets);
    dictEntry *de;

    while((de = dictNext(di)) != NULL) {
        migrateCachedSocket *cs = dictGetVal(de);

        if ((server.unixtime - cs->last_use_time) > MIGRATE_SOCKET_CACHE_TTL) {
            connClose(cs->conn);
            zfree(cs);
            dictDelete(server.migrate_cached_sockets, dictGetKey(de));
        }
    }
    dictReleaseIterator(di);
}

/* MIGRATE host port key dbid timeout [COPY | REPLACE | AUTH password |
 *      AUTH2 username password]
 *
 * On in the multiple keys form:
 *
 * MIGRATE host port "" dbid timeout [COPY | REPLACE | AUTH password |
 *      AUTH2 username password] KEYS key1 key2 ... keyN */
void migrateCommand(client *c) {
    migrateCachedSocket *cs;
    int copy = 0, replace = 0, j;
    char *username = NULL;
    char *password = NULL;
    long timeout;
    long dbid;
    robj **ov = NULL; /* Objects to migrate. */
    robj **kv = NULL; /* Key names. */
    robj **newargv = NULL; /* Used to rewrite the command as DEL ... keys ...

```

```

*/
    rio cmd, payload;
    int may_retry = 1;
    int write_error = 0;
    int argv_rewritten = 0;

    /* To support the KEYS option we need the following additional state. */
    int first_key = 3; /* Argument index of the first key. */
    int num_keys = 1; /* By default only migrate the 'key' argument. */

    /* Parse additional options */
    for (j = 6; j < c->argc; j++) {
        int moreargs = (c->argc-1) - j;
        if (!strcasecmp(c->argv[j]->ptr,"copy")) {
            copy = 1;
        } else if (!strcasecmp(c->argv[j]->ptr,"replace")) {
            replace = 1;
        } else if (!strcasecmp(c->argv[j]->ptr,"auth")) {
            if (!moreargs) {
                addReplyErrorObject(c,shared.syntaxerr);
                return;
            }
            j++;
            password = c->argv[j]->ptr;
            redactClientCommandArgument(c,j);
        } else if (!strcasecmp(c->argv[j]->ptr,"auth2")) {
            if (moreargs < 2) {
                addReplyErrorObject(c,shared.syntaxerr);
                return;
            }
            username = c->argv[++j]->ptr;
            redactClientCommandArgument(c,j);
            password = c->argv[++j]->ptr;
            redactClientCommandArgument(c,j);
        } else if (!strcasecmp(c->argv[j]->ptr,"keys")) {
            if (sdslen(c->argv[3]->ptr) != 0) {
                addReplyError(c,
                    "When using MIGRATE KEYS option, the key argument"
                    " must be set to the empty string");
                return;
            }
            first_key = j+1;
            num_keys = c->argc - j - 1;
            break; /* All the remaining args are keys. */
        } else {
            addReplyErrorObject(c,shared.syntaxerr);
            return;
        }
    }

    /* Sanity check */

```

```

if (getLongFromObjectOrReply(c,c->argv[5],&timeout,NULL) != C_OK ||
    getLongFromObjectOrReply(c,c->argv[4],&dbid,NULL) != C_OK)
{
    return;
}
if (timeout <= 0) timeout = 1000;

/* Check if the keys are here. If at least one key is to migrate, do it
 * otherwise if all the keys are missing reply with "NOKEY" to signal
 * the caller there was nothing to migrate. We don't return an error in
 * this case, since often this is due to a normal condition like the key
 * expiring in the meantime. */
ov = zrealloc(ov,sizeof(robj*)*num_keys);
kv = zrealloc(kv,sizeof(robj*)*num_keys);
int oi = 0;

for (j = 0; j < num_keys; j++) {
    if ((ov[oi] = lookupKeyRead(c->db,c->argv[first_key+j])) != NULL) {
        kv[oi] = c->argv[first_key+j];
        oi++;
    }
}
num_keys = oi;
if (num_keys == 0) {
    zfree(ov); zfree(kv);
    addReplySds(c,sdsnew("+NOKEY\r\n"));
    return;
}

try_again:
write_error = 0;

/* Connect */
cs = migrateGetSocket(c,c->argv[1],c->argv[2],timeout);
if (cs == NULL) {
    zfree(ov); zfree(kv);
    return; /* error sent to the client by migrateGetSocket() */
}

rioInitWithBuffer(&cmd,sdsempty());

/* Authentication */
if (password) {
    int arity = username ? 3 : 2;
    serverAssertWithInfo(c,NULL,rioWriteBulkCount(&cmd,'*',arity));
    serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,"AUTH",4));
    if (username) {
        serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,username,
            sdslen(username)));
    }
    serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,password,

```

```

        sdslen(password)));
    }

    /* Send the SELECT command if the current DB is not already selected. */
    int select = cs->last_dbid != dbid; /* Should we emit SELECT? */
    if (select) {
        serverAssertWithInfo(c,NULL,rioWriteBulkCount(&cmd,'*',2));
        serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,"SELECT",6));
        serverAssertWithInfo(c,NULL,rioWriteBulkLongLong(&cmd,dbid));
    }

    int non_expired = 0; /* Number of keys that we'll find non expired.
                           Note that serializing large keys may take some time
                           so certain keys that were found non expired by the
                           lookupKey() function, may be expired later. */

    /* Create RESTORE payload and generate the protocol to call the command. */
    for (j = 0; j < num_keys; j++) {
        long long ttl = 0;
        long long expireat = getExpire(c->db,kv[j]);

        if (expireat != -1) {
            ttl = expireat-mstime();
            if (ttl < 0) {
                continue;
            }
            if (ttl < 1) ttl = 1;
        }

        /* Relocate valid (non expired) keys and values into the array in
        successive
        * positions to remove holes created by the keys that were present
        * in the first lookup but are now expired after the second lookup. */
        ov[non_expired] = ov[j];
        kv[non_expired++] = kv[j];

        serverAssertWithInfo(c,NULL,
            rioWriteBulkCount(&cmd,'*',replace ? 5 : 4));

        if (server.cluster_enabled)
            serverAssertWithInfo(c,NULL,
                rioWriteBulkString(&cmd,"RESTORE-ASKING",14));
        else
            serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,"RESTORE",7));
        serverAssertWithInfo(c,NULL,sdsEncodedObject(kv[j]));
        serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,kv[j]->ptr,
            sdslen(kv[j]->ptr)));
        serverAssertWithInfo(c,NULL,rioWriteBulkLongLong(&cmd,ttl));

        /* Emit the payload argument, that is the serialized object using
        * the DUMP format. */
    }

```



```

        createDumpPayload(&payload,ov[j],kv[j],dbid);
        serverAssertWithInfo(c,NULL,
            rioWriteBulkString(&cmd,payload.io.buffer.ptr,
                sdslen(payload.io.buffer.ptr)));
        sdsfree(payload.io.buffer.ptr);

        /* Add the REPLACE option to the RESTORE command if it was specified
         * as a MIGRATE option. */
        if (replace)
            serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,"REPLACE",7));
    }

    /* Fix the actual number of keys we are migrating. */
    num_keys = non_expired;

    /* Transfer the query to the other node in 64K chunks. */
    errno = 0;
    {
        sds buf = cmd.io.buffer.ptr;
        size_t pos = 0, towrite;
        int nwritten = 0;

        while ((towrite = sdslen(buf)-pos) > 0) {
            towrite = (towrite > (64*1024) ? (64*1024) : towrite);
            nwritten = connSyncWrite(cs->conn,buf+pos,towrite,timeout);
            if (nwritten != (signed)towrite) {
                write_error = 1;
                goto socket_err;
            }
            pos += nwritten;
        }
    }

    char buf0[1024]; /* Auth reply. */
    char buf1[1024]; /* Select reply. */
    char buf2[1024]; /* Restore reply. */

    /* Read the AUTH reply if needed. */
    if (password && connSyncReadLine(cs->conn, buf0, sizeof(buf0), timeout) <=
0)
        goto socket_err;

    /* Read the SELECT reply if needed. */
    if (select && connSyncReadLine(cs->conn, buf1, sizeof(buf1), timeout) <= 0)
        goto socket_err;

    /* Read the RESTORE replies. */
    int error_from_target = 0;
    int socket_error = 0;
    int del_idx = 1; /* Index of the key argument for the replicated DEL op. */

```

```

/* Allocate the new argument vector that will replace the current command,
 * to propagate the MIGRATE as a DEL command (if no COPY option was given).
 * We allocate num_keys+1 because the additional argument is for "DEL"
 * command name itself. */
if (!copy) newargv = zmalloc(sizeof(robj)*(num_keys+1));

for (j = 0; j < num_keys; j++) {
    if (connSyncReadLine(cs->conn, buf2, sizeof(buf2), timeout) <= 0) {
        socket_error = 1;
        break;
    }
    if ((password && buf0[0] == '-') ||
        (select && buf1[0] == '-') ||
        buf2[0] == '-')
    {
        /* On error assume that last_dbid is no longer valid. */
        if (!error_from_target) {
            cs->last_dbid = -1;
            char *errbuf;
            if (password && buf0[0] == '-') errbuf = buf0;
            else if (select && buf1[0] == '-') errbuf = buf1;
            else errbuf = buf2;

            error_from_target = 1;
            addReplyErrorFormat(c,"Target instance replied with error: %s",
                               errbuf+1);
        }
    } else {
        if (!copy) {
            /* No COPY option: remove the local key, signal the change. */
            dbDelete(c->db,kv[j]);
            signalModifiedKey(c,c->db,kv[j]);
            notifyKeyspaceEvent(NOTIFY_GENERIC,"del",kv[j],c->db->id);
            server.dirty++;

            /* Populate the argument vector to replace the old one. */
            newargv[del_idx++] = kv[j];
            incrRefCount(kv[j]);
        }
    }
}

/* On socket error, if we want to retry, do it now before rewriting the
 * command vector. We only retry if we are sure nothing was processed
 * and we failed to read the first reply (j == 0 test). */
if (!error_from_target && socket_error && j == 0 && may_retry &&
    errno != ETIMEDOUT)
{
    goto socket_err; /* A retry is guaranteed because of tested
conditions.*/
}

```

```

/* On socket errors, close the migration socket now that we still have
 * the original host/port in the ARGV. Later the original command may be
 * rewritten to DEL and will be too later. */
if (socket_error) migrateCloseSocket(c->argv[1],c->argv[2]);

if (!copy) {
    /* Translate MIGRATE as DEL for replication/AOF. Note that we do
     * this only for the keys for which we received an acknowledgement
     * from the receiving Redis server, by using the del_idx index. */
    if (del_idx > 1) {
        newargv[0] = createStringObject("DEL",3);
        /* Note that the following call takes ownership of newargv. */
        replaceClientCommandVector(c,del_idx,newargv);
        argv_rewritten = 1;
    } else {
        /* No key transfer acknowledged, no need to rewrite as DEL. */
        zfree(newargv);
    }
    newargv = NULL; /* Make it safe to call zfree() on it in the future. */
}

/* If we are here and a socket error happened, we don't want to retry.
 * Just signal the problem to the client, but only do it if we did not
 * already queue a different error reported by the destination server. */
if (!error_from_target && socket_error) {
    may_retry = 0;
    goto socket_err;
}

if (!error_from_target) {
    /* Success! Update the last_dbid in migrateCachedSocket, so that we can
     * avoid SELECT the next time if the target DB is the same. Reply +OK.
     *
     * Note: If we reached this point, even if socket_error is true
     * still the SELECT command succeeded (otherwise the code jumps to
     * socket_err label. */
    cs->last_dbid = dbid;
    addReply(c,shared.ok);
} else {
    /* On error we already sent it in the for loop above, and set
     * the currently selected socket to -1 to force SELECT the next time.
 */
}

sdsfree(cmd.io.buffer.ptr);
zfree(ov); zfree(kv); zfree(newargv);
return;

/* On socket errors we try to close the cached socket and try again.
 * It is very common for the cached socket to get closed, if just reopening

```

```

    * it works it's a shame to notify the error to the caller. */
socket_err:
    /* Cleanup we want to perform in both the retry and no retry case.
     * Note: Closing the migrate socket will also force SELECT next time. */
    sdsfree(cmd.io.buffer.ptr);

    /* If the command was rewritten as DEL and there was a socket error,
     * we already closed the socket earlier. While migrateCloseSocket()
     * is idempotent, the host/port arguments are now gone, so don't do it
     * again. */
    if (!argv_rewritten) migrateCloseSocket(c->argv[1],c->argv[2]);
    zfree(newargv);
    newargv = NULL; /* This will get reallocated on retry. */

    /* Retry only if it's not a timeout and we never attempted a retry
     * (or the code jumping here did not set may_retry to zero). */
    if (errno != ETIMEDOUT && may_retry) {
        may_retry = 0;
        goto try_again;
    }

    /* Cleanup we want to do if no retry is attempted. */
    zfree(ov); zfree(kv);
    addReplyErrorSds(c, sdscatprintf(sdsempy(),
                                     "-IOERR error or timeout %s to target
instance",
                                     write_error ? "writing" : "reading"));
    return;
}

/* -----
-
* Cluster functions related to serving / redirecting clients
* -----
*/

/* The ASKING command is required after a -ASK redirection.
 * The client should issue ASKING before to actually send the command to
 * the target instance. See the Redis Cluster specification for more
 * information. */
void askingCommand(client *c) {
    if (server.cluster_enabled == 0) {
        addReplyError(c,"This instance has cluster support disabled");
        return;
    }
    c->flags |= CLIENT_ASKING;
    addReply(c,shared.ok);
}

/* The READONLY command is used by clients to enter the read-only mode.
 * In this mode slaves will not redirect clients as long as clients access

```

```

    * with read-only commands to keys that are served by the slave's master. */
void readonlyCommand(client *c) {
    if (server.cluster_enabled == 0) {
        addReplyError(c, "This instance has cluster support disabled");
        return;
    }
    c->flags |= CLIENT_READONLY;
    addReply(c, shared.ok);
}

/* The READWRITE command just clears the READONLY command state. */
void readwriteCommand(client *c) {
    if (server.cluster_enabled == 0) {
        addReplyError(c, "This instance has cluster support disabled");
        return;
    }
    c->flags &= ~CLIENT_READONLY;
    addReply(c, shared.ok);
}

/* Return the pointer to the cluster node that is able to serve the command.
 * For the function to succeed the command should only target either:
 *
 * 1) A single key (even multiple times like LPOPRUSH mylist mylist).
 * 2) Multiple keys in the same hash slot, while the slot is stable (no
 *    resharding in progress).
 *
 * On success the function returns the node that is able to serve the request.
 * If the node is not 'myself' a redirection must be performed. The kind of
 * redirection is specified setting the integer passed by reference
 * 'error_code', which will be set to CLUSTER_REDIR_ASK or
 * CLUSTER_REDIR_MOVED.
 *
 * When the node is 'myself' 'error_code' is set to CLUSTER_REDIR_NONE.
 *
 * If the command fails NULL is returned, and the reason of the failure is
 * provided via 'error_code', which will be set to:
 *
 * CLUSTER_REDIR_CROSS_SLOT if the request contains multiple keys that
 * don't belong to the same hash slot.
 *
 * CLUSTER_REDIR_UNSTABLE if the request contains multiple keys
 * belonging to the same slot, but the slot is not stable (in migration or
 * importing state, likely because a resharding is in progress).
 *
 * CLUSTER_REDIR_DOWN_UNBOUND if the request addresses a slot which is
 * not bound to any node. In this case the cluster global state should be
 * already "down" but it is fragile to rely on the update of the global state,
 * so we also handle it here.
 *
 * CLUSTER_REDIR_DOWN_STATE and CLUSTER_REDIR_DOWN_RO_STATE if the cluster is

```

```

    * down but the user attempts to execute a command that addresses one or more
    keys. */
clusterNode *getNodeByQuery(client *c, struct redisCommand *cmd, robj **argv,
int argc, int *hashslot, int *error_code) {
    clusterNode *n = NULL;
    robj *firstkey = NULL;
    int multiple_keys = 0;
    multiState *ms, _ms;
    multiCmd mc;
    int i, slot = 0, migrating_slot = 0, importing_slot = 0, missing_keys = 0;

    /* Allow any key to be set if a module disabled cluster redirections. */
    if (server.cluster_module_flags & CLUSTER_MODULE_FLAG_NO_REDIRECTION)
        return myself;

    /* Set error code optimistically for the base case. */
    if (error_code) *error_code = CLUSTER_REDIR_NONE;

    /* Modules can turn off Redis Cluster redirection: this is useful
     * when writing a module that implements a completely different
     * distributed system. */

    /* We handle all the cases as if they were EXEC commands, so we have
     * a common code path for everything */
    if (cmd->proc == execCommand) {
        /* If CLIENT_MULTI flag is not set EXEC is just going to return an
         * error. */
        if (!(c->flags & CLIENT_MULTI)) return myself;
        ms = &c->mstate;
    } else {
        /* In order to have a single codepath create a fake Multi State
         * structure if the client is not in MULTI/EXEC state, this way
         * we have a single codepath below. */
        ms = &_ms;
        _ms.commands = &mc;
        _ms.count = 1;
        mc.argv = argv;
        mc argc = argc;
        mc.cmd = cmd;
    }

    int is_pubsubshard = cmd->proc == ssubscribeCommand ||
        cmd->proc == sunsubscribeCommand ||
        cmd->proc == spublishCommand;

    /* Check that all the keys are in the same hash slot, and obtain this
     * slot and the node associated. */
    for (i = 0; i < ms->count; i++) {
        struct redisCommand *mcmd;
        robj **margv;
        int margc, numkeys, j;

```

```

keyReference *keyindex;

mcmd = ms->commands[i].cmd;
margc = ms->commands[i].argc;
margv = ms->commands[i].argv;

getKeyResult result = GETKEYS_RESULT_INIT;
numkeys = getKeyFromCommand(mcmd,margv,margc,&result);
keyindex = result.keys;

for (j = 0; j < numkeys; j++) {
    robj *thiskey = margv[keyindex[j].pos];
    int thisslot = keyHashSlot((char*)thiskey->ptr,
                               sdslen(thiskey->ptr));

    if (firstkey == NULL) {
        /* This is the first key we see. Check what is the slot
         * and node. */
        firstkey = thiskey;
        slot = thisslot;
        n = server.cluster->slots[slot];

        /* Error: If a slot is not served, we are in "cluster down"
         * state. However the state is yet to be updated, so this was
         * not trapped earlier in processCommand(). Report the same
         * error to the client. */
        if (n == NULL) {
            getKeyFreeResult(&result);
            if (error_code)
                *error_code = CLUSTER_REDIR_DOWN_UNBOUND;
            return NULL;
        }

        /* If we are migrating or importing this slot, we need to check
         * if we have all the keys in the request (the only way we
         * can safely serve the request, otherwise we return a TRYAGAIN
         * error). To do so we set the importing/migrating state and
         * increment a counter for every missing key. */
        if (n == myself &&
            server.cluster->migrating_slots_to[slot] != NULL)
        {
            migrating_slot = 1;
        } else if (server.cluster->importing_slots_from[slot] != NULL)
        {
            importing_slot = 1;
        }
    } else {
        /* If it is not the first key/channel, make sure it is exactly
         * the same key/channel as the first we saw. */
        if (!equalStringObjects(firstkey,thiskey)) {
            if (slot != thisslot) {

```

```

        /* Error: multiple keys from different slots. */
        getKeysFreeResult(&result);
        if (error_code)
            *error_code = CLUSTER_REDIR_CROSS_SLOT;
        return NULL;
    } else {
        /* Flag this request as one with multiple different
         * keys/channels. */
        multiple_keys = 1;
    }
}

/* Migrating / Importing slot? Count keys we don't have.
 * If it is pubsubshard command, it isn't required to check
 * the channel being present or not in the node during the
 * slot migration, the channel will be served from the source
 * node until the migration completes with CLUSTER SETSLOT <slot>
 * NODE <node-id>. */
int flags = LOOKUP_NOTOUCH | LOOKUP_NOSTATS | LOOKUP_NONOTIFY;
if ((migrating_slot || importing_slot) && !is_pubsubshard &&
    lookupKeyReadWithFlags(&server.db[0], thiskey, flags) == NULL)
{
    missing_keys++;
}

}
getKeysFreeResult(&result);
}

/* No key at all in command? then we can serve the request
 * without redirections or errors in all the cases. */
if (n == NULL) return myself;

/* Cluster is globally down but we got keys? We only serve the request
 * if it is a read command and when allow_reads_when_down is enabled. */
if (server.cluster->state != CLUSTER_OK) {
    if (is_pubsubshard) {
        if (!server.cluster_allow_pubsubshard_when_down) {
            if (error_code) *error_code = CLUSTER_REDIR_DOWN_STATE;
            return NULL;
        }
    } else if (!server.cluster_allow_reads_when_down) {
        /* The cluster is configured to block commands when the
         * cluster is down. */
        if (error_code) *error_code = CLUSTER_REDIR_DOWN_STATE;
        return NULL;
    } else if (cmd->flags & CMD_WRITE) {
        /* The cluster is configured to allow read only commands */
        if (error_code) *error_code = CLUSTER_REDIR_DOWN_RO_STATE;
        return NULL;
    } else {

```



```

        /* Fall through and allow the command to be executed:
         * this happens when server.cluster_allow_reads_when_down is
         * true and the command is not a write command */
    }
}

/* Return the hashslot by reference. */
if (hashslot) *hashslot = slot;

/* MIGRATE always works in the context of the local node if the slot
 * is open (migrating or importing state). We need to be able to freely
 * move keys among instances in this case. */
if ((migrating_slot || importing_slot) && cmd->proc == migrateCommand)
    return myself;

/* If we don't have all the keys and we are migrating the slot, send
 * an ASK redirection. */
if (migrating_slot && missing_keys) {
    if (error_code) *error_code = CLUSTER_REDIR_ASK;
    return server.cluster->migrating_slots_to[slot];
}

/* If we are receiving the slot, and the client correctly flagged the
 * request as "ASKING", we can serve the request. However if the request
 * involves multiple keys and we don't have them all, the only option is
 * to send a TRYAGAIN error. */
if (importing_slot &&
    (c->flags & CLIENT_ASKING || cmd->flags & CMD_ASKING))
{
    if (multiple_keys && missing_keys) {
        if (error_code) *error_code = CLUSTER_REDIR_UNSTABLE;
        return NULL;
    } else {
        return myself;
    }
}

/* Handle the read-only client case reading from a slave: if this
 * node is a slave and the request is about a hash slot our master
 * is serving, we can reply without redirection. */
int is_write_command = (c->cmd->flags & CMD_WRITE) ||
    (c->cmd->proc == execCommand && (c->mstate.cmd_flags
& CMD_WRITE));
if (((c->flags & CLIENT_READONLY) || is_pubsubshard) &&
    !is_write_command &&
    nodeIsSlave(myself) &&
    myself->slaveof == n)
{
    return myself;
}

```

```

    /* Base case: just return the right node. However if this node is not
    * myself, set error_code to MOVED since we need to issue a redirection. */
    if (n != myself && error_code) *error_code = CLUSTER_REDIR_MOVED;
    return n;
}

/* Send the client the right redirection code, according to error_code
 * that should be set to one of CLUSTER_REDIR_* macros.
 *
 * If CLUSTER_REDIR_ASK or CLUSTER_REDIR_MOVED error codes
 * are used, then the node 'n' should not be NULL, but should be the
 * node we want to mention in the redirection. Moreover hashslot should
 * be set to the hash slot that caused the redirection. */
void clusterRedirectClient(client *c, clusterNode *n, int hashslot, int
error_code) {
    if (error_code == CLUSTER_REDIR_CROSS_SLOT) {
        addReplyError(c, "-CROSSSLOT Keys in request don't hash to the same
slot");
    } else if (error_code == CLUSTER_REDIR_UNSTABLE) {
        /* The request spawns multiple keys in the same slot,
        * but the slot is not "stable" currently as there is
        * a migration or import in progress. */
        addReplyError(c, "-TRYAGAIN Multiple keys request during rehashing of
slot");
    } else if (error_code == CLUSTER_REDIR_DOWN_STATE) {
        addReplyError(c, "-CLUSTERDOWN The cluster is down");
    } else if (error_code == CLUSTER_REDIR_DOWN_RO_STATE) {
        addReplyError(c, "-CLUSTERDOWN The cluster is down and only accepts read
commands");
    } else if (error_code == CLUSTER_REDIR_DOWN_UNBOUND) {
        addReplyError(c, "-CLUSTERDOWN Hash slot not served");
    } else if (error_code == CLUSTER_REDIR_MOVED ||
        error_code == CLUSTER_REDIR_ASK)
    {
        /* Redirect to IP:port. Include plaintext port if cluster is TLS but
        * client is non-TLS. */
        int use_pport = (server.tls_cluster &&
            c->conn && connGetType(c->conn) != CONN_TYPE_TLS);
        int port = use_pport && n->pport ? n->pport : n->port;
        addReplyErrorSds(c, sdscatprintf(sdsempty(),
            "-%s %d %s:%d",
            (error_code == CLUSTER_REDIR_ASK) ? "ASK" : "MOVED",
            hashslot, getPreferredEndpoint(n), port));
    } else {
        serverPanic("getNodeByQuery() unknown error.");
    }
}

/* This function is called by the function processing clients incrementally
 * to detect timeouts, in order to handle the following case:
 *

```

```

* 1) A client blocks with BLOPOP or similar blocking operation.
* 2) The master migrates the hash slot elsewhere or turns into a slave.
* 3) The client may remain blocked forever (or up to the max timeout time)
*   waiting for a key change that will never happen.
*
* If the client is found to be blocked into a hash slot this node no
* longer handles, the client is sent a redirection error, and the function
* returns 1. Otherwise 0 is returned and no operation is performed. */
int clusterRedirectBlockedClientIfNeeded(client *c) {
    if (c->flags & CLIENT_BLOCKED &&
        (c->btype == BLOCKED_LIST ||
         c->btype == BLOCKED_ZSET ||
         c->btype == BLOCKED_STREAM ||
         c->btype == BLOCKED_MODULE))
    {
        dictEntry *de;
        dictIterator *di;

        /* If the cluster is down, unblock the client with the right error.
         * If the cluster is configured to allow reads on cluster down, we
         * still want to emit this error since a write will be required
         * to unblock them which may never come. */
        if (server.cluster->state == CLUSTER_FAIL) {
            clusterRedirectClient(c, NULL, 0, CLUSTER_REDIR_DOWN_STATE);
            return 1;
        }

        /* If the client is blocked on module, but not on a specific key,
         * don't unblock it (except for the CLUSTER_FAIL case above). */
        if (c->btype == BLOCKED_MODULE && !moduleClientIsBlockedOnKeys(c))
            return 0;

        /* All keys must belong to the same slot, so check first key only. */
        di = dictGetIterator(c->bpop.keys);
        if ((de = dictNext(di)) != NULL) {
            robj *key = dictGetKey(de);
            int slot = keyHashSlot((char*)key->ptr, sdslen(key->ptr));
            clusterNode *node = server.cluster->slots[slot];

            /* if the client is read-only and attempting to access key that our
             * replica can handle, allow it. */
            if ((c->flags & CLIENT_READONLY) &&
                !(c->lastcmd->flags & CMD_WRITE) &&
                nodeIsSlave(myself) && myself->slaveof == node)
            {
                node = myself;
            }

            /* We send an error and unblock the client if:
             * 1) The slot is unassigned, emitting a cluster down error.
             * 2) The slot is not handled by this node, nor being imported. */

```

```

        if (node != myself &&
            server.cluster->importing_slots_from[slot] == NULL)
        {
            if (node == NULL) {
                clusterRedirectClient(c,NULL,0,
                    CLUSTER_REDIR_DOWN_UNBOUND);
            } else {
                clusterRedirectClient(c,node,slot,
                    CLUSTER_REDIR_MOVED);
            }
            dictReleaseIterator(di);
            return 1;
        }
    }
    dictReleaseIterator(di);
}
return 0;
}

```

/* Slot to Key API. This is used by Redis Cluster in order to obtain in
 * a fast way a key that belongs to a specified hash slot. This is useful
 * while rehashing the cluster and in other conditions when we need to
 * understand if we have keys for a given hash slot. */

```

void slotToKeyAddEntry(dictEntry *entry, redisDb *db) {
    sds key = entry->key;
    unsigned int hashslot = keyHashSlot(key, sdslen(key));
    slotToKeys *slot_to_keys = &(*db->slots_to_keys).by_slot[hashslot];
    slot_to_keys->count++;

    /* Insert entry before the first element in the list. */
    dictEntry *first = slot_to_keys->head;
    dictEntryNextInSlot(entry) = first;
    if (first != NULL) {
        serverAssert(dictEntryPrevInSlot(first) == NULL);
        dictEntryPrevInSlot(first) = entry;
    }
    serverAssert(dictEntryPrevInSlot(entry) == NULL);
    slot_to_keys->head = entry;
}

```

```

void slotToKeyDelEntry(dictEntry *entry, redisDb *db) {
    sds key = entry->key;
    unsigned int hashslot = keyHashSlot(key, sdslen(key));
    slotToKeys *slot_to_keys = &(*db->slots_to_keys).by_slot[hashslot];
    slot_to_keys->count--;

    /* Connect previous and next entries to each other. */
    dictEntry *next = dictEntryNextInSlot(entry);
    dictEntry *prev = dictEntryPrevInSlot(entry);
    if (next != NULL) {

```

```

        dictEntryPrevInSlot(next) = prev;
    }
    if (prev != NULL) {
        dictEntryNextInSlot(prev) = next;
    } else {
        /* The removed entry was the first in the list. */
        serverAssert(slot_to_keys->head == entry);
        slot_to_keys->head = next;
    }
}

/* Updates neighbour entries when an entry has been replaced (e.g. reallocated
 * during active defrag). */
void slotToKeyReplaceEntry(dictEntry *entry, redisDb *db) {
    dictEntry *next = dictEntryNextInSlot(entry);
    dictEntry *prev = dictEntryPrevInSlot(entry);
    if (next != NULL) {
        dictEntryPrevInSlot(next) = entry;
    }
    if (prev != NULL) {
        dictEntryNextInSlot(prev) = entry;
    } else {
        /* The replaced entry was the first in the list. */
        sds key = entry->key;
        unsigned int hashslot = keyHashSlot(key, sdslen(key));
        slotToKeys *slot_to_keys = &(*db->slots_to_keys).by_slot[hashslot];
        slot_to_keys->head = entry;
    }
}

/* Initialize slots-keys map of given db. */
void slotToKeyInit(redisDb *db) {
    db->slots_to_keys = zcalloc(sizeof(clusterSlotToKeyMapping));
}

/* Empty slots-keys map of given db. */
void slotToKeyFlush(redisDb *db) {
    memset(db->slots_to_keys, 0,
        sizeof(clusterSlotToKeyMapping));
}

/* Free slots-keys map of given db. */
void slotToKeyDestroy(redisDb *db) {
    zfree(db->slots_to_keys);
    db->slots_to_keys = NULL;
}

/* Remove all the keys in the specified hash slot.
 * The number of removed items is returned. */
unsigned int delKeysInSlot(unsigned int hashslot) {
    unsigned int j = 0;

```

```

dictEntry *de = (*server.db->slots_to_keys).by_slot[hashslot].head;
while (de != NULL) {
    sds sdskey = dictGetKey(de);
    de = dictEntryNextInSlot(de);
    robj *key = createStringObject(sdskey, sdslen(sdskey));
    dbDelete(&server.db[0], key);
    decrRefCount(key);
    j++;
}
return j;
}

unsigned int countKeysInSlot(unsigned int hashslot) {
    return (*server.db->slots_to_keys).by_slot[hashslot].count;
}

/* -----
-
* Operation(s) on channel rax tree.
* -----
*/

void slotToChannelUpdate(sds channel, int add) {
    size_t keylen = sdslen(channel);
    unsigned int hashslot = keyHashSlot(channel, keylen);
    unsigned char buf[64];
    unsigned char *indexed = buf;

    if (keylen+2 > 64) indexed = zmalloc(keylen+2);
    indexed[0] = (hashslot >> 8) & 0xff;
    indexed[1] = hashslot & 0xff;
    memcpy(indexed+2, channel, keylen);
    if (add) {
        raxInsert(server.cluster->slots_to_channels, indexed, keylen+2, NULL, NULL);
    } else {
        raxRemove(server.cluster->slots_to_channels, indexed, keylen+2, NULL);
    }
    if (indexed != buf) zfree(indexed);
}

void slotToChannelAdd(sds channel) {
    slotToChannelUpdate(channel, 1);
}

void slotToChannelDel(sds channel) {
    slotToChannelUpdate(channel, 0);
}

/* Get the count of the channels for a given slot. */
unsigned int countChannelsInSlot(unsigned int hashslot) {

```

```
    raxIterator iter;
    int j = 0;
    unsigned char indexed[2];

    indexed[0] = (hashslot >> 8) & 0xff;
    indexed[1] = hashslot & 0xff;
    raxStart(&iter, server.cluster->slots_to_channels);
    raxSeek(&iter, ">=", indexed, 2);
    while(raxNext(&iter)) {
        if (iter.key[0] != indexed[0] || iter.key[1] != indexed[1]) break;
        j++;
    }
    raxStop(&iter);
    return j;
}
```

/cluster.h

[to top](#)

```

#ifndef __CLUSTER_H
#define __CLUSTER_H

/*-----
 * Redis cluster data structures, defines, exported API.
 *-----
*/

#define CLUSTER_SLOTS 16384
#define CLUSTER_OK 0 /* Everything looks ok */
#define CLUSTER_FAIL 1 /* The cluster can't work */
#define CLUSTER_NAMELEN 40 /* sha1 hex length */
#define CLUSTER_PORT_INCR 10000 /* Cluster port = baseport + PORT_INCR */

/* The following defines are amount of time, sometimes expressed as
 * multipliers of the node timeout value (when ending with MULT). */
#define CLUSTER_FAIL_REPORT_VALIDITY_MULT 2 /* Fail report validity. */
#define CLUSTER_FAIL_UNDO_TIME_MULT 2 /* Undo fail if master is back. */
#define CLUSTER_MF_TIMEOUT 5000 /* Milliseconds to do a manual failover. */
#define CLUSTER_MF_PAUSE_MULT 2 /* Master pause manual failover mult. */
#define CLUSTER_SLAVE_MIGRATION_DELAY 5000 /* Delay for slave migration. */

/* Redirection errors returned by getNodeByQuery(). */
#define CLUSTER_REDIR_NONE 0 /* Node can serve the request. */
#define CLUSTER_REDIR_CROSS_SLOT 1 /* -CROSSSLOT request. */
#define CLUSTER_REDIR_UNSTABLE 2 /* -TRYAGAIN redirection required */
#define CLUSTER_REDIR_ASK 3 /* -ASK redirection required. */
#define CLUSTER_REDIR_MOVED 4 /* -MOVED redirection required. */
#define CLUSTER_REDIR_DOWN_STATE 5 /* -CLUSTERDOWN, global state. */
#define CLUSTER_REDIR_DOWN_UNBOUND 6 /* -CLUSTERDOWN, unbound slot. */
#define CLUSTER_REDIR_DOWN_RO_STATE 7 /* -CLUSTERDOWN, allow reads. */

struct clusterNode;

/* clusterLink encapsulates everything needed to talk with a remote node. */
typedef struct clusterLink {
    mstime_t ctime; /* Link creation time */
    connection *conn; /* Connection to remote node */
    sds sndbuf; /* Packet send buffer */
    char *rcvbuf; /* Packet reception buffer */
    size_t rcvbuf_len; /* Used size of rcvbuf */
    size_t rcvbuf_alloc; /* Allocated size of rcvbuf */
    struct clusterNode *node; /* Node related to this link. Initialized to
NULL when unknown */
    int inbound; /* 1 if this link is an inbound link accepted
from the related node */
} clusterLink;

/* Cluster node flags and macros. */
#define CLUSTER_NODE_MASTER 1 /* The node is a master */

```



```
#define CLUSTER_NODE_SLAVE 2      /* The node is a slave */
#define CLUSTER_NODE_PFAIL 4     /* Failure? Need acknowledge */
#define CLUSTER_NODE_FAIL 8     /* The node is believed to be malfunctioning */
/*
#define CLUSTER_NODE_MYSELF 16   /* This node is myself */
#define CLUSTER_NODE_HANDSHAKE 32 /* We have still to exchange the first ping */
*/
#define CLUSTER_NODE_NOADDR 64  /* We don't know the address of this node */
#define CLUSTER_NODE_MEET 128   /* Send a MEET message to this node */
#define CLUSTER_NODE_MIGRATE_TO 256 /* Master eligible for replica migration. */
/*
#define CLUSTER_NODE_NOFAILOVER 512 /* Slave will not try to failover. */
#define CLUSTER_NODE_NULL_NAME
"\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\"

#define nodeIsMaster(n) ((n)->flags & CLUSTER_NODE_MASTER)
#define nodeIsSlave(n) ((n)->flags & CLUSTER_NODE_SLAVE)
#define nodeInHandshake(n) ((n)->flags & CLUSTER_NODE_HANDSHAKE)
#define nodeHasAddr(n) (!((n)->flags & CLUSTER_NODE_NOADDR))
#define nodeWithoutAddr(n) ((n)->flags & CLUSTER_NODE_NOADDR)
#define nodeTimedOut(n) ((n)->flags & CLUSTER_NODE_PFAIL)
#define nodeFailed(n) ((n)->flags & CLUSTER_NODE_FAIL)
#define nodeCantFailover(n) ((n)->flags & CLUSTER_NODE_NOFAILOVER)

/* Reasons why a slave is not able to failover. */
#define CLUSTER_CANT_FAILOVER_NONE 0
#define CLUSTER_CANT_FAILOVER_DATA_AGE 1
#define CLUSTER_CANT_FAILOVER_WAITING_DELAY 2
#define CLUSTER_CANT_FAILOVER_EXPIRED 3
#define CLUSTER_CANT_FAILOVER_WAITING_VOTES 4
#define CLUSTER_CANT_FAILOVER_RELOG_PERIOD (60*5) /* seconds. */

/* clusterState todo_before_sleep flags. */
#define CLUSTER_TODO_HANDLE_FAILOVER (1<<0)
#define CLUSTER_TODO_UPDATE_STATE (1<<1)
#define CLUSTER_TODO_SAVE_CONFIG (1<<2)
#define CLUSTER_TODO_FSYNC_CONFIG (1<<3)
#define CLUSTER_TODO_HANDLE_MANUAL_FAILOVER (1<<4)

/* Message types.
 *
 * Note that the PING, PONG and MEET messages are actually the same exact
 * kind of packet. PONG is the reply to ping, in the exact format as a PING,
 * while MEET is a special PING that forces the receiver to add the sender
 * as a node (if it is not already in the list). */
#define CLUSTERMSG_TYPE_PING 0    /* Ping */
#define CLUSTERMSG_TYPE_PONG 1    /* Pong (reply to Ping) */
#define CLUSTERMSG_TYPE_MEET 2    /* Meet "let's join" message */
#define CLUSTERMSG_TYPE_FAIL 3    /* Mark node xxx as failing */
#define CLUSTERMSG_TYPE_PUBLISH 4 /* Pub/Sub Publish propagation */
#define CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST 5 /* May I failover? */
```

```

#define CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK 6      /* Yes, you have my vote */
#define CLUSTERMSG_TYPE_UPDATE 7                /* Another node slots configuration */
#define CLUSTERMSG_TYPE_MFSTART 8               /* Pause clients for manual failover */
#define CLUSTERMSG_TYPE_MODULE 9               /* Module cluster API message. */
#define CLUSTERMSG_TYPE_PUBLISHSHARD 10        /* Pub/Sub Publish shard propagation */
#define CLUSTERMSG_TYPE_COUNT 11               /* Total number of message types. */

/* Flags that a module can set in order to prevent certain Redis Cluster
 * features to be enabled. Useful when implementing a different distributed
 * system on top of Redis Cluster message bus, using modules. */
#define CLUSTER_MODULE_FLAG_NONE 0
#define CLUSTER_MODULE_FLAG_NO_FAILOVER (1<<1)
#define CLUSTER_MODULE_FLAG_NO_REDIRECTION (1<<2)

/* This structure represent elements of node->fail_reports. */
typedef struct clusterNodeFailReport {
    struct clusterNode *node; /* Node reporting the failure condition. */
    mstime_t time;           /* Time of the last report from this node. */
} clusterNodeFailReport;

typedef struct clusterNode {
    mstime_t ctime; /* Node object creation time. */
    char name[CLUSTER_NAMELEN]; /* Node name, hex string, sha1-size */
    int flags; /* CLUSTER_NODE_... */
    uint64_t configEpoch; /* Last configEpoch observed for this node */
    unsigned char slots[CLUSTER_SLOTS/8]; /* slots handled by this node */
    uint16_t *slot_info_pairs; /* Slots info represented as (start/end) pair
(consecutive index). */
    int slot_info_pairs_count; /* Used number of slots in slot_info_pairs */
    int numslots; /* Number of slots handled by this node */
    int numslaves; /* Number of slave nodes, if this is a master */
    struct clusterNode **slaves; /* pointers to slave nodes */
    struct clusterNode *slaveof; /* pointer to the master node. Note that it
may be NULL even if the node is a slave
if we don't have the master node in our
tables. */

    mstime_t ping_sent; /* Unix time we sent latest ping */
    mstime_t pong_received; /* Unix time we received the pong */
    mstime_t data_received; /* Unix time we received any data */
    mstime_t fail_time; /* Unix time when FAIL flag was set */
    mstime_t voted_time; /* Last time we voted for a slave of this master
*/
    mstime_t repl_offset_time; /* Unix time we received offset for this node
*/
    mstime_t orphaned_time; /* Starting time of orphaned master condition
*/
    long long repl_offset; /* Last known repl offset for this node. */
    char ip[NET_IP_STR_LEN]; /* Latest known IP address of this node */
    sds hostname; /* The known hostname for this node */
    int port; /* Latest known clients port (TLS or plain). */
    int pport; /* Latest known clients plaintext port. Only

```

```

used

                                if the main clients port is for TLS. */
    int cport;                    /* Latest known cluster port of this node. */
    clusterLink *link;            /* TCP/IP link established toward this node */
    clusterLink *inbound_link;    /* TCP/IP link accepted from this node */
    list *fail_reports;           /* List of nodes signaling this as failing */
} clusterNode;

/* Slot to keys for a single slot. The keys in the same slot are linked
together
 * using dictEntry metadata. */
typedef struct slotToKeys {
    uint64_t count;               /* Number of keys in the slot. */
    dictEntry *head;              /* The first key-value entry in the slot. */
} slotToKeys;

/* Slot to keys mapping for all slots, opaque outside this file. */
struct clusterSlotToKeyMapping {
    slotToKeys by_slot[CLUSTER_SLOTS];
};

/* Dict entry metadata for cluster mode, used for the Slot to Key API to form a
 * linked list of the entries belonging to the same slot. */
typedef struct clusterDictEntryMetadata {
    dictEntry *prev;              /* Prev entry with key in the same slot */
    dictEntry *next;              /* Next entry with key in the same slot */
} clusterDictEntryMetadata;

typedef struct clusterState {
    clusterNode *myself;          /* This node */
    uint64_t currentEpoch;
    int state;                    /* CLUSTER_OK, CLUSTER_FAIL, ... */
    int size;                     /* Num of master nodes with at least one slot */
    dict *nodes;                  /* Hash table of name -> clusterNode structures */
    dict *nodes_black_list;        /* Nodes we don't re-add for a few seconds. */
    clusterNode *migrating_slots_to[CLUSTER_SLOTS];
    clusterNode *importing_slots_from[CLUSTER_SLOTS];
    clusterNode *slots[CLUSTER_SLOTS];
    rax *slots_to_channels;
    /* The following fields are used to take the slave state on elections. */
    mstime_t failover_auth_time; /* Time of previous or next election. */
    int failover_auth_count;      /* Number of votes received so far. */
    int failover_auth_sent;       /* True if we already asked for votes. */
    int failover_auth_rank;       /* This slave rank for current auth request. */
    uint64_t failover_auth_epoch; /* Epoch of the current election. */
    int cant_failover_reason;     /* Why a slave is currently not able to
                                failover. See the CANT_FAILOVER_* macros. */
    /* Manual failover state in common. */
    mstime_t mf_end;              /* Manual failover time limit (ms unixtime).
                                It is zero if there is no MF in progress. */
}

```

```

/* Manual failover state of master. */
clusterNode *mf_slave; /* Slave performing the manual failover. */
/* Manual failover state of slave. */
long long mf_master_offset; /* Master offset the slave needs to start MF
                             or -1 if still not received. */

int mf_can_start; /* If non-zero signal that the manual failover
                  can start requesting masters vote. */

/* The following fields are used by masters to take state on elections. */
uint64_t lastVoteEpoch; /* Epoch of the last vote granted. */
int todo_before_sleep; /* Things to do in clusterBeforeSleep(). */
/* Stats */
/* Messages received and sent by type. */
long long stats_bus_messages_sent[CLUSTERMSG_TYPE_COUNT];
long long stats_bus_messages_received[CLUSTERMSG_TYPE_COUNT];
long long stats_pfail_nodes; /* Number of nodes in PFAIL status,
                             excluding nodes without address. */
unsigned long long stat_cluster_links_buffer_limit_exceeded; /* Total
number of cluster links freed due to exceeding buffer limit */
} clusterState;

/* Redis cluster messages header */

/* Initially we don't know our "name", but we'll find it once we connect
 * to the first node, using the getsockname() function. Then we'll use this
 * address for all the next messages. */
typedef struct {
    char nodename[CLUSTER_NAMELEN];
    uint32_t ping_sent;
    uint32_t pong_received;
    char ip[NET_IP_STR_LEN]; /* IP address last time it was seen */
    uint16_t port; /* base port last time it was seen */
    uint16_t cport; /* cluster port last time it was seen */
    uint16_t flags; /* node->flags copy */
    uint16_t pport; /* plaintext-port, when base port is TLS */
    uint16_t notused1;
} clusterMsgDataGossip;

typedef struct {
    char nodename[CLUSTER_NAMELEN];
} clusterMsgDataFail;

typedef struct {
    uint32_t channel_len;
    uint32_t message_len;
    unsigned char bulk_data[8]; /* 8 bytes just as placeholder. */
} clusterMsgDataPublish;

typedef struct {
    uint64_t configEpoch; /* Config epoch of the specified instance. */
    char nodename[CLUSTER_NAMELEN]; /* Name of the slots owner. */
    unsigned char slots[CLUSTER_SLOTS/8]; /* Slots bitmap. */

```

```

} clusterMsgDataUpdate;

typedef struct {
    uint64_t module_id;    /* ID of the sender module. */
    uint32_t len;          /* ID of the sender module. */
    uint8_t type;          /* Type from 0 to 255. */
    unsigned char bulk_data[3]; /* 3 bytes just as placeholder. */
} clusterMsgModule;

/* The cluster supports optional extension messages that can be sent
 * along with ping/pong/meet messages to give additional info in a
 * consistent manner. */
typedef enum {
    CLUSTERMSG_EXT_TYPE_HOSTNAME,
} clusterMsgPingtypes;

/* Helper function for making sure extensions are eight byte aligned. */
#define EIGHT_BYTE_ALIGN(size) (((size) + 7) / 8) * 8

typedef struct {
    char hostname[1]; /* The announced hostname, ends with \0. */
} clusterMsgPingExtHostname;

typedef struct {
    uint32_t length; /* Total length of this extension message (including this
header) */
    uint16_t type; /* Type of this extension message (see
clusterMsgPingExtTypes) */
    uint16_t unused; /* 16 bits of padding to make this structure 8 byte
aligned. */
    union {
        clusterMsgPingExtHostname hostname;
    } ext[]; /* Actual extension information, formatted so that the data is 8
* byte aligned, regardless of its content. */
} clusterMsgPingExt;

union clusterMsgData {
    /* PING, MEET and PONG */
    struct {
        /* Array of N clusterMsgDataGossip structures */
        clusterMsgDataGossip gossip[1];
        /* Extension data that can optionally be sent for ping/meet/pong
* messages. We can't explicitly define them here though, since
* the gossip array isn't the real length of the gossip data. */
    } ping;

    /* FAIL */
    struct {
        clusterMsgDataFail about;
    } fail;

```

```

/* PUBLISH */
struct {
    clusterMsgDataPublish msg;
} publish;

/* UPDATE */
struct {
    clusterMsgDataUpdate nodecfg;
} update;

/* MODULE */
struct {
    clusterMsgModule msg;
} module;
};

#define CLUSTER_PROTO_VER 1 /* Cluster bus protocol version. */

typedef struct {
    char sig[4]; /* Signature "RCmb" (Redis Cluster message bus). */
    uint32_t totlen; /* Total length of this message */
    uint16_t ver; /* Protocol version, currently set to 1. */
    uint16_t port; /* TCP base port number. */
    uint16_t type; /* Message type */
    uint16_t count; /* Only used for some kind of messages. */
    uint64_t currentEpoch; /* The epoch accordingly to the sending node. */
    uint64_t configEpoch; /* The config epoch if it's a master, or the last
                           epoch advertised by its master if it is a
                           slave. */
    uint64_t offset; /* Master replication offset if node is a master or
                     processed replication offset if node is a slave. */
    char sender[CLUSTER_NAMELEN]; /* Name of the sender node */
    unsigned char myslots[CLUSTER_SLOTS/8];
    char slaveof[CLUSTER_NAMELEN];
    char myip[NET_IP_STR_LEN]; /* Sender IP, if not all zeroed. */
    uint16_t extensions; /* Number of extensions sent along with this packet.
    */
    char notused1[30]; /* 30 bytes reserved for future usage. */
    uint16_t pport; /* Sender TCP plaintext port, if base port is TLS */
    uint16_t cport; /* Sender TCP cluster bus port */
    uint16_t flags; /* Sender node flags */
    unsigned char state; /* Cluster state from the POV of the sender */
    unsigned char mflags[3]; /* Message flags: CLUSTERMSG_FLAG[012]_... */
    union clusterMsgData data;
} clusterMsg;

/* clusterMsg defines the gossip wire protocol exchanged among Redis cluster
 * members, which can be running different versions of redis-server bits,
 * especially during cluster rolling upgrades.
 *
 * Therefore, fields in this struct should remain at the same offset from

```

```

* release to release. The static asserts below ensures that incompatible
* changes in clusterMsg be caught at compile time.
*/

static_assert(offsetof(clusterMsg, sig) == 0, "unexpected field offset");
static_assert(offsetof(clusterMsg, totlen) == 4, "unexpected field offset");
static_assert(offsetof(clusterMsg, ver) == 8, "unexpected field offset");
static_assert(offsetof(clusterMsg, port) == 10, "unexpected field offset");
static_assert(offsetof(clusterMsg, type) == 12, "unexpected field offset");
static_assert(offsetof(clusterMsg, count) == 14, "unexpected field offset");
static_assert(offsetof(clusterMsg, currentEpoch) == 16, "unexpected field
offset");
static_assert(offsetof(clusterMsg, configEpoch) == 24, "unexpected field
offset");
static_assert(offsetof(clusterMsg, offset) == 32, "unexpected field offset");
static_assert(offsetof(clusterMsg, sender) == 40, "unexpected field offset");
static_assert(offsetof(clusterMsg, myslots) == 80, "unexpected field offset");
static_assert(offsetof(clusterMsg, slaveof) == 2128, "unexpected field
offset");
static_assert(offsetof(clusterMsg, myip) == 2168, "unexpected field offset");
static_assert(offsetof(clusterMsg, extensions) == 2214, "unexpected field
offset");
static_assert(offsetof(clusterMsg, notused1) == 2216, "unexpected field
offset");
static_assert(offsetof(clusterMsg, pport) == 2246, "unexpected field offset");
static_assert(offsetof(clusterMsg, cport) == 2248, "unexpected field offset");
static_assert(offsetof(clusterMsg, flags) == 2250, "unexpected field offset");
static_assert(offsetof(clusterMsg, state) == 2252, "unexpected field offset");
static_assert(offsetof(clusterMsg, mflags) == 2253, "unexpected field offset");
static_assert(offsetof(clusterMsg, data) == 2256, "unexpected field offset");

#define CLUSTERMSG_MIN_LEN (sizeof(clusterMsg)-sizeof(union clusterMsgData))

/* Message flags better specify the packet content or are used to
 * provide some information about the node state. */
#define CLUSTERMSG_FLAG0_PAUSED (1<<0) /* Master paused for manual failover. */
#define CLUSTERMSG_FLAG0_FORCEACK (1<<1) /* Give ACK to AUTH_REQUEST even if
master is up. */
#define CLUSTERMSG_FLAG0_EXT_DATA (1<<2) /* Message contains extension data */

/* ----- API exported outside cluster.c -----
*/
void clusterInit(void);
void clusterCron(void);
void clusterBeforeSleep(void);
clusterNode *getNodeByQuery(client *c, struct redisCommand *cmd, robj **argv,
int argc, int *hashslot, int *ask);
int verifyClusterNodeId(const char *name, int length);
clusterNode *clusterLookupNode(const char *name, int length);
int clusterRedirectBlockedClientIfNeeded(client *c);
void clusterRedirectClient(client *c, clusterNode *n, int hashslot, int

```



```
error_code);
void migrateCloseTimedoutSockets(void);
int verifyClusterConfigWithData(void);
unsigned long getClusterConnectionsCount(void);
int clusterSendModuleMessageToTarget(const char *target, uint64_t module_id,
uint8_t type, const char *payload, uint32_t len);
void clusterPropagatePublish(robj *channel, robj *message, int sharded);
unsigned int keyHashSlot(char *key, int keylen);
void slotToKeyAddEntry(dictEntry *entry, redisDb *db);
void slotToKeyDelEntry(dictEntry *entry, redisDb *db);
void slotToKeyReplaceEntry(dictEntry *entry, redisDb *db);
void slotToKeyInit(redisDb *db);
void slotToKeyFlush(redisDb *db);
void slotToKeyDestroy(redisDb *db);
void clusterUpdateMyselfFlags(void);
void clusterUpdateMyselfIp(void);
void slotToChannelAdd(sds channel);
void slotToChannelDel(sds channel);
void clusterUpdateMyselfHostname(void);

#endif /* __CLUSTER_H */
```

/commands.c

[to top](#)


```

/* Automatically generated by generate-command-code.py, do not edit. */

#include "server.h"

/* We have fabulous commands from
 * the fantastic
 * Redis Command Table! */

/***** BITCOUNT *****/

/* BITCOUNT history */
commandHistory BITCOUNT_History[] = {
{"7.0.0", "Added the `BYTE|BIT` option."},
{0}
};

/* BITCOUNT tips */
#define BITCOUNT_tips NULL

/* BITCOUNT index index_unit argument table */
struct redisCommandArg BITCOUNT_index_index_unit_Subargs[] = {
{"byte", ARG_TYPE_PURE_TOKEN, -1, "BYTE", NULL, NULL, CMD_ARG_NONE},
{"bit", ARG_TYPE_PURE_TOKEN, -1, "BIT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BITCOUNT index argument table */
struct redisCommandArg BITCOUNT_index_Subargs[] = {
{"start", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"end", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"index_unit", ARG_TYPE_ONEOF, -1, NULL, NULL, "7.0.0", CMD_ARG_OPTIONAL, .subargs=BITCOUNT_index_index_unit_Subargs},
{0}
};

/* BITCOUNT argument table */
struct redisCommandArg BITCOUNT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"index", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=BITCOUNT_index_Subargs},
{0}
};

/***** BITFIELD *****/

/* BITFIELD history */
#define BITFIELD_History NULL

/* BITFIELD tips */
#define BITFIELD_tips NULL

/* BITFIELD operation encoding_offset argument table */

```

```

struct redisCommandArg BITFIELD_operation_encoding_offset_Subargs[] = {
{"encoding", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BITFIELD operation write wrap_sat_fail argument table */
struct redisCommandArg BITFIELD_operation_write_wrap_sat_fail_Subargs[] = {
{"wrap", ARG_TYPE_PURE_TOKEN, -1, "WRAP", NULL, NULL, CMD_ARG_NONE},
{"sat", ARG_TYPE_PURE_TOKEN, -1, "SAT", NULL, NULL, CMD_ARG_NONE},
{"fail", ARG_TYPE_PURE_TOKEN, -1, "FAIL", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BITFIELD operation write write_operation encoding_offset_value argument
table */
struct redisCommandArg
BITFIELD_operation_write_write_operation_encoding_offset_value_Subargs[] = {
{"encoding", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BITFIELD operation write write_operation encoding_offset_increment argument
table */
struct redisCommandArg
BITFIELD_operation_write_write_operation_encoding_offset_increment_Subargs[] =
{
{"encoding", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"increment", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BITFIELD operation write write_operation argument table */
struct redisCommandArg BITFIELD_operation_write_write_operation_Subargs[] = {
{"encoding_offset_value", ARG_TYPE_BLOCK, -1, "SET", NULL, NULL, CMD_ARG_NONE, .subargs=B
{"encoding_offset_increment", ARG_TYPE_BLOCK, -1, "INCRBY", NULL, NULL, CMD_ARG_NONE, .su
{0}
};

/* BITFIELD operation write argument table */
struct redisCommandArg BITFIELD_operation_write_Subargs[] = {
{"wrap_sat_fail", ARG_TYPE_ONEOF, -1, "OVERFLOW", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=
{"write_operation", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=BITFIELD
{0}
};

/* BITFIELD operation argument table */
struct redisCommandArg BITFIELD_operation_Subargs[] = {

```

```

{"encoding_offset", ARG_TYPE_BLOCK, -1, "GET", NULL, NULL, CMD_ARG_NONE, .subargs=BITFIEL
{"write", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=BITFIELD_operation
{0}
};

/* BITFIELD argument table */
struct redisCommandArg BITFIELD_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"operation", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=BITFIELD_o
{0}
};

/***** BITFIELD_R0 *****/

/* BITFIELD_R0 history */
#define BITFIELD_R0_History NULL

/* BITFIELD_R0 tips */
#define BITFIELD_R0_tips NULL

/* BITFIELD_R0 encoding_offset argument table */
struct redisCommandArg BITFIELD_R0_encoding_offset_Subargs[] = {
{"encoding", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BITFIELD_R0 argument table */
struct redisCommandArg BITFIELD_R0_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"encoding_offset", ARG_TYPE_BLOCK, -1, "GET", NULL, NULL, CMD_ARG_MULTIPLE, .subargs=BIT
{0}
};

/***** BITOP *****/

/* BITOP history */
#define BITOP_History NULL

/* BITOP tips */
#define BITOP_tips NULL

/* BITOP argument table */
struct redisCommandArg BITOP_Args[] = {
{"operation", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"destkey", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** BITPOS *****/

```

```

/* BITPOS history */
commandHistory BITPOS_History[] = {
{"7.0.0","Added the `BYTE|BIT` option."},
{0}
};

/* BITPOS tips */
#define BITPOS_tips NULL

/* BITPOS index end_index index_unit argument table */
struct redisCommandArg BITPOS_index_end_index_index_unit_Subargs[] = {
{"byte",ARG_TYPE_PURE_TOKEN,-1,"BYTE",NULL,NULL,CMD_ARG_NONE},
{"bit",ARG_TYPE_PURE_TOKEN,-1,"BIT",NULL,NULL,CMD_ARG_NONE},
{0}
};

/* BITPOS index end_index argument table */
struct redisCommandArg BITPOS_index_end_index_Subargs[] = {
{"end",ARG_TYPE_INTEGER,-1,NULL,NULL,NULL,CMD_ARG_NONE},
{"index_unit",ARG_TYPE_ONEOF,-1,NULL,NULL,"7.0.0",CMD_ARG_OPTIONAL,.subargs=BITPOS_index_end_index_index_unit_Subargs},
{0}
};

/* BITPOS index argument table */
struct redisCommandArg BITPOS_index_Subargs[] = {
{"start",ARG_TYPE_INTEGER,-1,NULL,NULL,NULL,CMD_ARG_NONE},
{"end_index",ARG_TYPE_BLOCK,-1,NULL,NULL,NULL,CMD_ARG_OPTIONAL,.subargs=BITPOS_index_end_index_Subargs},
{0}
};

/* BITPOS argument table */
struct redisCommandArg BITPOS_Args[] = {
{"key",ARG_TYPE_KEY,0,NULL,NULL,NULL,CMD_ARG_NONE},
{"bit",ARG_TYPE_INTEGER,-1,NULL,NULL,NULL,CMD_ARG_NONE},
{"index",ARG_TYPE_BLOCK,-1,NULL,NULL,NULL,CMD_ARG_OPTIONAL,.subargs=BITPOS_index_Subargs},
{0}
};

/***** GETBIT *****/

/* GETBIT history */
#define GETBIT_History NULL

/* GETBIT tips */
#define GETBIT_tips NULL

/* GETBIT argument table */
struct redisCommandArg GETBIT_Args[] = {
{"key",ARG_TYPE_KEY,0,NULL,NULL,NULL,CMD_ARG_NONE},
{"offset",ARG_TYPE_INTEGER,-1,NULL,NULL,NULL,CMD_ARG_NONE},

```

```

{0}
};

/***** SETBIT *****/

/* SETBIT history */
#define SETBIT_History NULL

/* SETBIT tips */
#define SETBIT_tips NULL

/* SETBIT argument table */
struct redisCommandArg SETBIT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ASKING *****/

/* ASKING history */
#define ASKING_History NULL

/* ASKING tips */
#define ASKING_tips NULL

/***** CLUSTER ADDSLOTS *****/

/* CLUSTER ADDSLOTS history */
#define CLUSTER_ADDSLOTS_History NULL

/* CLUSTER ADDSLOTS tips */
const char *CLUSTER_ADDSLOTS_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER ADDSLOTS argument table */
struct redisCommandArg CLUSTER_ADDSLOTS_Args[] = {
{"slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** CLUSTER ADDSLOTSRANGE *****/

/* CLUSTER ADDSLOTSRANGE history */
#define CLUSTER_ADDSLOTSRANGE_History NULL

/* CLUSTER ADDSLOTSRANGE tips */
const char *CLUSTER_ADDSLOTSRANGE_tips[] = {

```

```

"nondeterministic_output",
NULL
};

/* CLUSTER ADDSLOTSRANGE start_slot_end_slot argument table */
struct redisCommandArg CLUSTER_ADDSLOTSRANGE_start_slot_end_slot_Subargs[] = {
{"start-slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"end-slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLUSTER ADDSLOTSRANGE argument table */
struct redisCommandArg CLUSTER_ADDSLOTSRANGE_Args[] = {
{"start-slot-end-
slot", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=CLUSTER_ADDSLOTSR
{0}
};

/***** CLUSTER BUMPEPOCH *****/

/* CLUSTER BUMPEPOCH history */
#define CLUSTER BUMPEPOCH_History NULL

/* CLUSTER BUMPEPOCH tips */
const char *CLUSTER BUMPEPOCH_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLUSTER COUNT_FAILURE_REPORTS *****/

/* CLUSTER COUNT_FAILURE_REPORTS history */
#define CLUSTER COUNT_FAILURE_REPORTS_History NULL

/* CLUSTER COUNT_FAILURE_REPORTS tips */
const char *CLUSTER COUNT_FAILURE_REPORTS_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER COUNT_FAILURE_REPORTS argument table */
struct redisCommandArg CLUSTER_COUNT_FAILURE_REPORTS_Args[] = {
{"node-id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER COUNTKEYSINSLOT *****/

/* CLUSTER COUNTKEYSINSLOT history */
#define CLUSTER COUNTKEYSINSLOT_History NULL

```

```

/* CLUSTER COUNTKEYSINSLOT tips */
const char *CLUSTER_COUNTKEYSINSLOT_tips[] = {
    "nondeterministic_output",
    NULL
};

/* CLUSTER COUNTKEYSINSLOT argument table */
struct redisCommandArg CLUSTER_COUNTKEYSINSLOT_Args[] = {
    {"slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/***** CLUSTER DELSLOTS *****/

/* CLUSTER DELSLOTS history */
#define CLUSTER_DELSLOTS_History NULL

/* CLUSTER DELSLOTS tips */
const char *CLUSTER_DELSLOTS_tips[] = {
    "nondeterministic_output",
    NULL
};

/* CLUSTER DELSLOTS argument table */
struct redisCommandArg CLUSTER_DELSLOTS_Args[] = {
    {"slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
    {0}
};

/***** CLUSTER DELSLOTSRANGE *****/

/* CLUSTER DELSLOTSRANGE history */
#define CLUSTER_DELSLOTSRANGE_History NULL

/* CLUSTER DELSLOTSRANGE tips */
const char *CLUSTER_DELSLOTSRANGE_tips[] = {
    "nondeterministic_output",
    NULL
};

/* CLUSTER DELSLOTSRANGE start_slot_end_slot argument table */
struct redisCommandArg CLUSTER_DELSLOTSRANGE_start_slot_end_slot_Subargs[] = {
    {"start-slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"end-slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* CLUSTER DELSLOTSRANGE argument table */
struct redisCommandArg CLUSTER_DELSLOTSRANGE_Args[] = {
    {"start-slot-end-
slot", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=CLUSTER_DELSLOTSR

```

```

{0}
};

/***** CLUSTER FAILOVER *****/

/* CLUSTER FAILOVER history */
#define CLUSTER_FAILOVER_History NULL

/* CLUSTER FAILOVER tips */
const char *CLUSTER_FAILOVER_tips[] = {
    "nondeterministic_output",
    NULL
};

/* CLUSTER FAILOVER options argument table */
struct redisCommandArg CLUSTER_FAILOVER_options_Subargs[] = {
    {"force", ARG_TYPE_PURE_TOKEN, -1, "FORCE", NULL, NULL, CMD_ARG_NONE},
    {"takeover", ARG_TYPE_PURE_TOKEN, -1, "TAKEOVER", NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* CLUSTER FAILOVER argument table */
struct redisCommandArg CLUSTER_FAILOVER_Args[] = {
    {"options", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=CLUSTER_FAIL
    {0}
};

/***** CLUSTER FLUSHSLOTS *****/

/* CLUSTER FLUSHSLOTS history */
#define CLUSTER_FLUSHSLOTS_History NULL

/* CLUSTER FLUSHSLOTS tips */
const char *CLUSTER_FLUSHSLOTS_tips[] = {
    "nondeterministic_output",
    NULL
};

/***** CLUSTER FORGET *****/

/* CLUSTER FORGET history */
#define CLUSTER_FORGET_History NULL

/* CLUSTER FORGET tips */
const char *CLUSTER_FORGET_tips[] = {
    "nondeterministic_output",
    NULL
};

/* CLUSTER FORGET argument table */
struct redisCommandArg CLUSTER_FORGET_Args[] = {

```



```

{"node-id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER GETKEYSINSLOT *****/

/* CLUSTER GETKEYSINSLOT history */
#define CLUSTER_GETKEYSINSLOT_History NULL

/* CLUSTER GETKEYSINSLOT tips */
const char *CLUSTER_GETKEYSINSLOT_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER GETKEYSINSLOT argument table */
struct redisCommandArg CLUSTER_GETKEYSINSLOT_Args[] = {
{"slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER HELP *****/

/* CLUSTER HELP history */
#define CLUSTER_HELP_History NULL

/* CLUSTER HELP tips */
#define CLUSTER_HELP_tips NULL

/***** CLUSTER INFO *****/

/* CLUSTER INFO history */
#define CLUSTER_INFO_History NULL

/* CLUSTER INFO tips */
const char *CLUSTER_INFO_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLUSTER KEYSLOT *****/

/* CLUSTER KEYSLOT history */
#define CLUSTER_KEYSLOT_History NULL

/* CLUSTER KEYSLOT tips */
const char *CLUSTER_KEYSLOT_tips[] = {
"nondeterministic_output",
NULL
};

```

```

/* CLUSTER KEYSLOT argument table */
struct redisCommandArg CLUSTER_KEYSLOT_Args[] = {
{"key", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER LINKS *****/

/* CLUSTER LINKS history */
#define CLUSTER_LINKS_History NULL

/* CLUSTER LINKS tips */
const char *CLUSTER_LINKS_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLUSTER MEET *****/

/* CLUSTER MEET history */
#define CLUSTER_MEET_History NULL

/* CLUSTER MEET tips */
const char *CLUSTER_MEET_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER MEET argument table */
struct redisCommandArg CLUSTER_MEET_Args[] = {
{"ip", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"port", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER MYID *****/

/* CLUSTER MYID history */
#define CLUSTER_MYID_History NULL

/* CLUSTER MYID tips */
const char *CLUSTER_MYID_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLUSTER NODES *****/

/* CLUSTER NODES history */
#define CLUSTER_NODES_History NULL

```

```

/* CLUSTER NODES tips */
const char *CLUSTER_NODES_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLUSTER REPLICAS *****/

/* CLUSTER REPLICAS history */
#define CLUSTER_REPLICAS_History NULL

/* CLUSTER REPLICAS tips */
const char *CLUSTER_REPLICAS_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER REPLICAS argument table */
struct redisCommandArg CLUSTER_REPLICAS_Args[] = {
{"node-id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER REPLICATE *****/

/* CLUSTER REPLICATE history */
#define CLUSTER_REPLICATE_History NULL

/* CLUSTER REPLICATE tips */
const char *CLUSTER_REPLICATE_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER REPLICATE argument table */
struct redisCommandArg CLUSTER_REPLICATE_Args[] = {
{"node-id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER RESET *****/

/* CLUSTER RESET history */
#define CLUSTER_RESET_History NULL

/* CLUSTER RESET tips */
const char *CLUSTER_RESET_tips[] = {
"nondeterministic_output",
NULL
};

```

```

/* CLUSTER RESET hard_soft argument table */
struct redisCommandArg CLUSTER_RESET_hard_soft_Subargs[] = {
{"hard", ARG_TYPE_PURE_TOKEN, -1, "HARD", NULL, NULL, CMD_ARG_NONE},
{"soft", ARG_TYPE_PURE_TOKEN, -1, "SOFT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLUSTER RESET argument table */
struct redisCommandArg CLUSTER_RESET_Args[] = {
{"hard_soft", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=CLUSTER_RE
{0}
};

/***** CLUSTER SAVECONFIG *****/

/* CLUSTER SAVECONFIG history */
#define CLUSTER_SAVECONFIG_History NULL

/* CLUSTER SAVECONFIG tips */
const char *CLUSTER_SAVECONFIG_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLUSTER SET_CONFIG_EPOCH *****/

/* CLUSTER SET_CONFIG_EPOCH history */
#define CLUSTER_SET_CONFIG_EPOCH_History NULL

/* CLUSTER SET_CONFIG_EPOCH tips */
const char *CLUSTER_SET_CONFIG_EPOCH_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER SET_CONFIG_EPOCH argument table */
struct redisCommandArg CLUSTER_SET_CONFIG_EPOCH_Args[] = {
{"config-epoch", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER SETSLOT *****/

/* CLUSTER SETSLOT history */
#define CLUSTER_SETSLOT_History NULL

/* CLUSTER SETSLOT tips */
const char *CLUSTER_SETSLOT_tips[] = {
"nondeterministic_output",
NULL
};

```

```

};

/* CLUSTER SETSLOT subcommand argument table */
struct redisCommandArg CLUSTER_SETSLOT_subcommand_Subargs[] = {
{"node-id", ARG_TYPE_STRING, -1, "IMPORTING", NULL, NULL, CMD_ARG_NONE},
{"node-id", ARG_TYPE_STRING, -1, "MIGRATING", NULL, NULL, CMD_ARG_NONE},
{"node-id", ARG_TYPE_STRING, -1, "NODE", NULL, NULL, CMD_ARG_NONE},
{"stable", ARG_TYPE_PURE_TOKEN, -1, "STABLE", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLUSTER SETSLOT argument table */
struct redisCommandArg CLUSTER_SETSLOT_Args[] = {
{"slot", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"subcommand", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=CLUSTER_SETSLOT_subcommand_Subargs},
{0}
};

/***** CLUSTER SHARDS *****/

/* CLUSTER SHARDS history */
#define CLUSTER_SHARDS_History NULL

/* CLUSTER SHARDS tips */
const char *CLUSTER_SHARDS_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLUSTER SLAVES *****/

/* CLUSTER SLAVES history */
#define CLUSTER_SLAVES_History NULL

/* CLUSTER SLAVES tips */
const char *CLUSTER_SLAVES_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER SLAVES argument table */
struct redisCommandArg CLUSTER_SLAVES_Args[] = {
{"node-id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLUSTER SLOTS *****/

/* CLUSTER SLOTS history */
commandHistory CLUSTER_SLOTS_History[] = {
{"4.0.0", "Added node IDs."},

```

```

{"7.0.0","Added additional networking metadata field."},
{0}
};

/* CLUSTER SLOTS tips */
const char *CLUSTER_SLOTS_tips[] = {
"nondeterministic_output",
NULL
};

/* CLUSTER command table */
struct redisCommand CLUSTER_Subcommands[] = {
{"addslots","Assign new hash slots to receiving node","0(N) where N is the
total number of hash slot
arguments","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_ADDSLOTS_H
{"addslotsrange","Assign new hash slots to receiving node","0(N) where N is the
total number of the slots between the start slot and end slot
arguments.", "7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_ADDSLOTSR
{"bumpepoch","Advance the cluster config
epoch","0(1)","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER BUMPEPO
{"count-failure-reports","Return the number of failure reports active for a
given node","0(N) where N is the number of failure
reports","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_COUNT_FAILUR
{"countkeysinslot","Return the number of local keys in the specified hash
slot","0(1)","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_COUNTKEY
{"delslots","Set hash slots as unbound in receiving node","0(N) where N is the
total number of hash slot
arguments","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_DELSLOTS_H
{"delslotsrange","Set hash slots as unbound in receiving node","0(N) where N is
the total number of the slots between the start slot and end slot
arguments.", "7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_DELSLOTSR
{"failover","Forces a replica to perform a manual failover of its
master.", "0(1)","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_FAILO
{"flushslots","Delete a node's own slots
information","0(1)","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_F
{"forget","Remove a node from the nodes
table","0(1)","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_FORGET_
{"getkeysinslot","Return local key names in the specified hash slot","0(log(N))
where N is the number of requested
keys","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_GETKEYSINSLOT_H
{"help","Show helpful text about the different
subcommands","0(1)","5.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_H
{"info","Provides info about Redis Cluster node
state","0(1)","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_INFO_Hi
{"keyslot","Returns the hash slot of the specified key","0(N) where N is the
number of bytes in the
key","3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_KEYSLOT_History,
{"links","Returns a list of all TCP links to and from peer nodes in
cluster","0(N) where N is the total number of Cluster
nodes","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CLUSTER,CLUSTER_LINKS_History,
{"meet","Force a node cluster to handshake with another

```

```

node", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_MEET_His
{"myid", "Return the node
id", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_MYID_Histo
{"nodes", "Get Cluster config for the node", "0(N) where N is the total number of
Cluster
nodes", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_NODES_History,
{"replicas", "List replica nodes of the specified master
node", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_REPLICAS
{"replicate", "Reconfigure a node as a replica of the specified master
node", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_REPLICAT
{"reset", "Reset a Redis Cluster node", "0(N) where N is the number of known
nodes. The command may execute a FLUSHALL as a side
effect.", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_RESET_Histor
{"saveconfig", "Forces the node to save cluster state on
disk", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_SAVECONF
{"set-config-epoch", "Set the configuration epoch in a new
node", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_SET_CONF
{"setslot", "Bind a hash slot to a specific
node", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_SETSLOT_
{"shards", "Get array of cluster slots to node mappings", "0(N) where N is the
total number of cluster
nodes", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_SHARDS_History
{"slaves", "List replica nodes of the specified master
node", "0(1)", "3.0.0", CMD_DOC_DEPRECATED, "`CLUSTER
REPLICAS`", "5.0.0", COMMAND_GROUP_CLUSTER, CLUSTER_SLAVES_History, CLUSTER_SLAVES_tip
{"slots", "Get array of Cluster slot to node mappings", "0(N) where N is the
total number of Cluster nodes", "3.0.0", CMD_DOC_DEPRECATED, "`CLUSTER
SHARDS`", "7.0.0", COMMAND_GROUP_CLUSTER, CLUSTER_SLOTS_History, CLUSTER_SLOTS_tips, cl
{0}
};

/***** CLUSTER *****/

/* CLUSTER history */
#define CLUSTER_History NULL

/* CLUSTER tips */
#define CLUSTER_tips NULL

/***** READONLY *****/

/* READONLY history */
#define READONLY_History NULL

/* READONLY tips */
#define READONLY_tips NULL

/***** READWRITE *****/

/* READWRITE history */
#define READWRITE_History NULL

```

```

/* READWRITE tips */
#define READWRITE_tips NULL

/***** AUTH *****/

/* AUTH history */
commandHistory AUTH_History[] = {
{"6.0.0", "Added ACL style (username and password)."},
{0}
};

/* AUTH tips */
#define AUTH_tips NULL

/* AUTH argument table */
struct redisCommandArg AUTH_Args[] = {
{"username", ARG_TYPE_STRING, -1, NULL, NULL, "6.0.0", CMD_ARG_OPTIONAL},
{"password", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLIENT CACHING *****/

/* CLIENT CACHING history */
#define CLIENT_CACHING_History NULL

/* CLIENT CACHING tips */
#define CLIENT_CACHING_tips NULL

/* CLIENT CACHING mode argument table */
struct redisCommandArg CLIENT_CACHING_mode_Subargs[] = {
{"yes", ARG_TYPE_PURE_TOKEN, -1, "YES", NULL, NULL, CMD_ARG_NONE},
{"no", ARG_TYPE_PURE_TOKEN, -1, "NO", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT CACHING argument table */
struct redisCommandArg CLIENT_CACHING_Args[] = {
{"mode", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=CLIENT_CACHING_mode_Subargs},
{0}
};

/***** CLIENT GETNAME *****/

/* CLIENT GETNAME history */
#define CLIENT_GETNAME_History NULL

/* CLIENT GETNAME tips */
#define CLIENT_GETNAME_tips NULL

```



```

/***** CLIENT GETREDIR *****/

/* CLIENT GETREDIR history */
#define CLIENT_GETREDIR_History NULL

/* CLIENT GETREDIR tips */
#define CLIENT_GETREDIR_tips NULL

/***** CLIENT HELP *****/

/* CLIENT HELP history */
#define CLIENT_HELP_History NULL

/* CLIENT HELP tips */
#define CLIENT_HELP_tips NULL

/***** CLIENT ID *****/

/* CLIENT ID history */
#define CLIENT_ID_History NULL

/* CLIENT ID tips */
#define CLIENT_ID_tips NULL

/***** CLIENT INFO *****/

/* CLIENT INFO history */
#define CLIENT_INFO_History NULL

/* CLIENT INFO tips */
const char *CLIENT_INFO_tips[] = {
"nondeterministic_output",
NULL
};

/***** CLIENT KILL *****/

/* CLIENT KILL history */
commandHistory CLIENT_KILL_History[] = {
{"2.8.12", "Added new filter format."},
{"2.8.12", "`ID` option."},
{"3.2.0", "Added `master` type in for `TYPE` option."},
{"5.0.0", "Replaced `slave` `TYPE` with `replica`. `slave` still supported for backward compatibility."},
{"6.2.0", "`LADDR` option."},
{0}
};

/* CLIENT KILL tips */
#define CLIENT_KILL_tips NULL

```

```

/* CLIENT KILL normal_master_slave_pubsub argument table */
struct redisCommandArg CLIENT_KILL_normal_master_slave_pubsub_Subargs[] = {
{"normal", ARG_TYPE_PURE_TOKEN, -1, "NORMAL", NULL, NULL, CMD_ARG_NONE},
{"master", ARG_TYPE_PURE_TOKEN, -1, "MASTER", NULL, "3.2.0", CMD_ARG_NONE},
{"slave", ARG_TYPE_PURE_TOKEN, -1, "SLAVE", NULL, NULL, CMD_ARG_NONE},
{"replica", ARG_TYPE_PURE_TOKEN, -1, "REPLICA", NULL, "5.0.0", CMD_ARG_NONE},
{"pubsub", ARG_TYPE_PURE_TOKEN, -1, "PUBSUB", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT KILL argument table */
struct redisCommandArg CLIENT_KILL_Args[] = {
{"ip:port", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{"client-id", ARG_TYPE_INTEGER, -1, "ID", NULL, "2.8.12", CMD_ARG_OPTIONAL},
{"normal_master_slave_pubsub", ARG_TYPE_ONEOF, -1, "TYPE", NULL, "2.8.12", CMD_ARG_OPTIO
{"username", ARG_TYPE_STRING, -1, "USER", NULL, NULL, CMD_ARG_OPTIONAL},
{"ip:port", ARG_TYPE_STRING, -1, "ADDR", NULL, NULL, CMD_ARG_OPTIONAL},
{"ip:port", ARG_TYPE_STRING, -1, "LADDR", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{"yes/no", ARG_TYPE_STRING, -1, "SKIPME", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** CLIENT LIST *****/

/* CLIENT LIST history */
commandHistory CLIENT_LIST_History[] = {
{"2.8.12", "Added unique client `id` field."},
{"5.0.0", "Added optional `TYPE` filter."},
{"6.2.0", "Added `laddr` field and the optional `ID` filter."},
{0}
};

/* CLIENT LIST tips */
const char *CLIENT_LIST_tips[] = {
"nondeterministic_output",
NULL
};

/* CLIENT LIST normal_master_replica_pubsub argument table */
struct redisCommandArg CLIENT_LIST_normal_master_replica_pubsub_Subargs[] = {
{"normal", ARG_TYPE_PURE_TOKEN, -1, "NORMAL", NULL, NULL, CMD_ARG_NONE},
{"master", ARG_TYPE_PURE_TOKEN, -1, "MASTER", NULL, NULL, CMD_ARG_NONE},
{"replica", ARG_TYPE_PURE_TOKEN, -1, "REPLICA", NULL, NULL, CMD_ARG_NONE},
{"pubsub", ARG_TYPE_PURE_TOKEN, -1, "PUBSUB", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT LIST id argument table */
struct redisCommandArg CLIENT_LIST_id_Subargs[] = {
{"client-id", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

```

```

};

/* CLIENT LIST argument table */
struct redisCommandArg CLIENT_LIST_Args[] = {
{"normal_master_replica_pubsub", ARG_TYPE_ONEOF, -1, "TYPE", NULL, "5.0.0", CMD_ARG_OPTI
{"id", ARG_TYPE_BLOCK, -1, "ID", NULL, "6.2.0", CMD_ARG_OPTIONAL, .subargs=CLIENT_LIST_id
{0}
};

/***** CLIENT NO_EVICT *****/

/* CLIENT NO_EVICT history */
#define CLIENT_NO_EVICT_History NULL

/* CLIENT NO_EVICT tips */
#define CLIENT_NO_EVICT_tips NULL

/* CLIENT NO_EVICT enabled argument table */
struct redisCommandArg CLIENT_NO_EVICT_enabled_Subargs[] = {
{"on", ARG_TYPE_PURE_TOKEN, -1, "ON", NULL, NULL, CMD_ARG_NONE},
{"off", ARG_TYPE_PURE_TOKEN, -1, "OFF", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT NO_EVICT argument table */
struct redisCommandArg CLIENT_NO_EVICT_Args[] = {
{"enabled", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=CLIENT_NO_EVICT_
{0}
};

/***** CLIENT PAUSE *****/

/* CLIENT PAUSE history */
commandHistory CLIENT_PAUSE_History[] = {
{"6.2.0", "`CLIENT PAUSE WRITE` mode added along with the `mode` option."},
{0}
};

/* CLIENT PAUSE tips */
#define CLIENT_PAUSE_tips NULL

/* CLIENT PAUSE mode argument table */
struct redisCommandArg CLIENT_PAUSE_mode_Subargs[] = {
{"write", ARG_TYPE_PURE_TOKEN, -1, "WRITE", NULL, NULL, CMD_ARG_NONE},
{"all", ARG_TYPE_PURE_TOKEN, -1, "ALL", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT PAUSE argument table */
struct redisCommandArg CLIENT_PAUSE_Args[] = {
{"timeout", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{"mode", ARG_TYPE_ONEOF, -1, NULL, NULL, "6.2.0", CMD_ARG_OPTIONAL, .subargs=CLIENT_PAUSE
{0}
};

/***** CLIENT REPLY *****/

/* CLIENT REPLY history */
#define CLIENT_REPLY_History NULL

/* CLIENT REPLY tips */
#define CLIENT_REPLY_tips NULL

/* CLIENT REPLY on_off_skip argument table */
struct redisCommandArg CLIENT_REPLY_on_off_skip_Subargs[] = {
{"on", ARG_TYPE_PURE_TOKEN, -1, "ON", NULL, NULL, CMD_ARG_NONE},
{"off", ARG_TYPE_PURE_TOKEN, -1, "OFF", NULL, NULL, CMD_ARG_NONE},
{"skip", ARG_TYPE_PURE_TOKEN, -1, "SKIP", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT REPLY argument table */
struct redisCommandArg CLIENT_REPLY_Args[] = {
{"on_off_skip", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=CLIENT_REPLY
{0}
};

/***** CLIENT SETNAME *****/

/* CLIENT SETNAME history */
#define CLIENT_SETNAME_History NULL

/* CLIENT SETNAME tips */
#define CLIENT_SETNAME_tips NULL

/* CLIENT SETNAME argument table */
struct redisCommandArg CLIENT_SETNAME_Args[] = {
{"connection-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** CLIENT TRACKING *****/

/* CLIENT TRACKING history */
#define CLIENT_TRACKING_History NULL

/* CLIENT TRACKING tips */
#define CLIENT_TRACKING_tips NULL

/* CLIENT TRACKING status argument table */
struct redisCommandArg CLIENT_TRACKING_status_Subargs[] = {
{"on", ARG_TYPE_PURE_TOKEN, -1, "ON", NULL, NULL, CMD_ARG_NONE},

```

```

{"off", ARG_TYPE_PURE_TOKEN, -1, "OFF", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT TRACKING argument table */
struct redisCommandArg CLIENT_TRACKING_Args[] = {
{"status", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=CLIENT_TRACKING_s
{"client-id", ARG_TYPE_INTEGER, -1, "REDIRECT", NULL, NULL, CMD_ARG_OPTIONAL},
{"prefix", ARG_TYPE_STRING, -1, "PREFIX", NULL, NULL, CMD_ARG_OPTIONAL|CMD_ARG_MULTIPLE|
{"bcast", ARG_TYPE_PURE_TOKEN, -1, "BCAST", NULL, NULL, CMD_ARG_OPTIONAL},
{"optin", ARG_TYPE_PURE_TOKEN, -1, "OPTIN", NULL, NULL, CMD_ARG_OPTIONAL},
{"optout", ARG_TYPE_PURE_TOKEN, -1, "OPTOUT", NULL, NULL, CMD_ARG_OPTIONAL},
{"nolop", ARG_TYPE_PURE_TOKEN, -1, "NOLOOP", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** CLIENT TRACKINGINFO *****/

/* CLIENT TRACKINGINFO history */
#define CLIENT_TRACKINGINFO_History NULL

/* CLIENT TRACKINGINFO tips */
#define CLIENT_TRACKINGINFO_tips NULL

/***** CLIENT UNBLOCK *****/

/* CLIENT UNBLOCK history */
#define CLIENT_UNBLOCK_History NULL

/* CLIENT UNBLOCK tips */
#define CLIENT_UNBLOCK_tips NULL

/* CLIENT UNBLOCK timeout_error argument table */
struct redisCommandArg CLIENT_UNBLOCK_timeout_error_Subargs[] = {
{"timeout", ARG_TYPE_PURE_TOKEN, -1, "TIMEOUT", NULL, NULL, CMD_ARG_NONE},
{"error", ARG_TYPE_PURE_TOKEN, -1, "ERROR", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CLIENT UNBLOCK argument table */
struct redisCommandArg CLIENT_UNBLOCK_Args[] = {
{"client-id", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"timeout_error", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=CLIENT
{0}
};

/***** CLIENT UNPAUSE *****/

/* CLIENT UNPAUSE history */
#define CLIENT_UNPAUSE_History NULL

```

```

/* CLIENT UNPAUSE tips */
#define CLIENT_UNPAUSE_tips NULL

/* CLIENT command table */
struct redisCommand CLIENT_Subcommands[] = {
{"caching","Instruct the server about tracking or not keys in the next
request","0(1)","6.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_CAC
{"getname","Get the current connection
name","0(1)","2.6.9",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_GETNAM
{"getredir","Get tracking notifications redirection client ID if
any","0(1)","6.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_GETREDI
{"help","Show helpful text about the different
subcommands","0(1)","5.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT
{"id","Returns the client ID for the current
connection","0(1)","5.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_
{"info","Returns information about the current client
connection.","0(1)","6.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT
{"kill","Kill the connection of a client","0(N) where N is the number of client
connections","2.4.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_KILL_H
{"list","Get the list of client connections","0(N) where N is the number of
client
connections","2.4.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_LIST_H
{"no-evict","Set client eviction mode for the current
connection","0(1)","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_
{"pause","Stop processing commands from clients for some
time","0(1)","2.9.50",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_PAUSE
{"reply","Instruct the server whether to reply to
commands","0(1)","3.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_RE
{"setname","Set the current connection
name","0(1)","2.6.9",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_SETNAM
{"tracking","Enable or disable server assisted client side caching
support","0(1). Some options may introduce additional
complexity.","6.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_TRACKI
{"trackinginfo","Return information about server assisted client side caching
for the current
connection","0(1)","6.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_
{"unblock","Unblock a client blocked in a blocking command from a different
connection","0(log N) where N is the number of client
connections","5.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_UNBLOC
{"unpause","Resume processing of clients that were paused","0(N) Where N is the
number of paused
clients","6.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_CONNECTION,CLIENT_UNPAUSE_Hi
{0}
};

/***** CLIENT *****/

/* CLIENT history */
#define CLIENT_History NULL

/* CLIENT tips */

```

```

#define CLIENT_tips NULL

/***** ECHO *****/

/* ECHO history */
#define ECHO_History NULL

/* ECHO tips */
#define ECHO_tips NULL

/* ECHO argument table */
struct redisCommandArg ECHO_Args[] = {
{"message", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HELLO *****/

/* HELLO history */
commandHistory HELLO_History[] = {
{"6.2.0", "`protover` made optional; when called without arguments the command reports the current connection's context."},
{0}
};

/* HELLO tips */
#define HELLO_tips NULL

/* HELLO arguments username_password argument table */
struct redisCommandArg HELLO_arguments_username_password_Subargs[] = {
{"username", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"password", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* HELLO arguments argument table */
struct redisCommandArg HELLO_arguments_Subargs[] = {
{"protover", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"username_password", ARG_TYPE_BLOCK, -1, "AUTH", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=
{"clientname", ARG_TYPE_STRING, -1, "SETNAME", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* HELLO argument table */
struct redisCommandArg HELLO_Args[] = {
{"arguments", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=HELLO_argu
{0}
};

/***** PING *****/

```

```

/* PING history */
#define PING_History NULL

/* PING tips */
const char *PING_tips[] = {
    "request_policy:all_shards",
    "response_policy:all_succeeded",
    NULL
};

/* PING argument table */
struct redisCommandArg PING_Args[] = {
    {"message", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
    {0}
};

/***** QUIT *****/

/* QUIT history */
#define QUIT_History NULL

/* QUIT tips */
#define QUIT_tips NULL

/***** RESET *****/

/* RESET history */
#define RESET_History NULL

/* RESET tips */
#define RESET_tips NULL

/***** SELECT *****/

/* SELECT history */
#define SELECT_History NULL

/* SELECT tips */
#define SELECT_tips NULL

/* SELECT argument table */
struct redisCommandArg SELECT_Args[] = {
    {"index", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/***** COPY *****/

/* COPY history */
#define COPY_History NULL

```



```

/* COPY tips */
#define COPY_tips NULL

/* COPY argument table */
struct redisCommandArg COPY_Args[] = {
{"source", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"destination", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"destination-db", ARG_TYPE_INTEGER, -1, "DB", NULL, NULL, CMD_ARG_OPTIONAL},
{"replace", ARG_TYPE_PURE_TOKEN, -1, "REPLACE", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** DEL *****/

/* DEL history */
#define DEL_History NULL

/* DEL tips */
const char *DEL_tips[] = {
"request_policy:multi_shard",
"response_policy:agg_sum",
NULL
};

/* DEL argument table */
struct redisCommandArg DEL_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** DUMP *****/

/* DUMP history */
#define DUMP_History NULL

/* DUMP tips */
const char *DUMP_tips[] = {
"nondeterministic_output",
NULL
};

/* DUMP argument table */
struct redisCommandArg DUMP_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** EXISTS *****/

/* EXISTS history */
commandHistory EXISTS_History[] = {

```

```

{"3.0.3","Accepts multiple `key` arguments."},
{0}
};

/* EXISTS tips */
const char *EXISTS_tips[] = {
"request_policy:multi_shard",
"response_policy:agg_sum",
NULL
};

/* EXISTS argument table */
struct redisCommandArg EXISTS_Args[] = {
{"key",ARG_TYPE_KEY,0,NULL,NULL,NULL,CMD_ARG_MULTIPLE},
{0}
};

/***** EXPIRE *****/

/* EXPIRE history */
commandHistory EXPIRE_History[] = {
{"7.0.0","Added options: `NX`, `XX`, `GT` and `LT`."},
{0}
};

/* EXPIRE tips */
#define EXPIRE_tips NULL

/* EXPIRE condition argument table */
struct redisCommandArg EXPIRE_condition_Subargs[] = {
{"nx",ARG_TYPE_PURE_TOKEN,-1,"NX",NULL,NULL,CMD_ARG_NONE},
{"xx",ARG_TYPE_PURE_TOKEN,-1,"XX",NULL,NULL,CMD_ARG_NONE},
{"gt",ARG_TYPE_PURE_TOKEN,-1,"GT",NULL,NULL,CMD_ARG_NONE},
{"lt",ARG_TYPE_PURE_TOKEN,-1,"LT",NULL,NULL,CMD_ARG_NONE},
{0}
};

/* EXPIRE argument table */
struct redisCommandArg EXPIRE_Args[] = {
{"key",ARG_TYPE_KEY,0,NULL,NULL,NULL,CMD_ARG_NONE},
{"seconds",ARG_TYPE_INTEGER,-1,NULL,NULL,NULL,CMD_ARG_NONE},
{"condition",ARG_TYPE_ONEOF,-1,NULL,NULL,"7.0.0",CMD_ARG_OPTIONAL,.subargs=EXPIRE_
{0}
};

/***** EXPIREAT *****/

/* EXPIREAT history */
commandHistory EXPIREAT_History[] = {
{"7.0.0","Added options: `NX`, `XX`, `GT` and `LT`."},
{0}
};

```

```

};

/* EXPIREAT tips */
#define EXPIREAT_tips NULL

/* EXPIREAT condition argument table */
struct redisCommandArg EXPIREAT_condition_Subargs[] = {
{"nx", ARG_TYPE_PURE_TOKEN, -1, "NX", NULL, NULL, CMD_ARG_NONE},
{"xx", ARG_TYPE_PURE_TOKEN, -1, "XX", NULL, NULL, CMD_ARG_NONE},
{"gt", ARG_TYPE_PURE_TOKEN, -1, "GT", NULL, NULL, CMD_ARG_NONE},
{"lt", ARG_TYPE_PURE_TOKEN, -1, "LT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* EXPIREAT argument table */
struct redisCommandArg EXPIREAT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"unix-time-seconds", ARG_TYPE_UNIX_TIME, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"condition", ARG_TYPE_ONEOF, -1, NULL, NULL, "7.0.0", CMD_ARG_OPTIONAL, .subargs=EXPIREA
{0}
};

/***** EXPIRETIME *****/

/* EXPIRETIME history */
#define EXPIRETIME_History NULL

/* EXPIRETIME tips */
#define EXPIRETIME_tips NULL

/* EXPIRETIME argument table */
struct redisCommandArg EXPIRETIME_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** KEYS *****/

/* KEYS history */
#define KEYS_History NULL

/* KEYS tips */
const char *KEYS_tips[] = {
"request_policy:all_shards",
"nondeterministic_output_order",
NULL
};

/* KEYS argument table */
struct redisCommandArg KEYS_Args[] = {
{"pattern", ARG_TYPE_PATTERN, -1, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{0}
};

/***** MIGRATE *****/

/* MIGRATE history */
commandHistory MIGRATE_History[] = {
{"3.0.0", "Added the `COPY` and `REPLACE` options."},
{"3.0.6", "Added the `KEYS` option."},
{"4.0.7", "Added the `AUTH` option."},
{"6.0.0", "Added the `AUTH2` option."},
{0}
};

/* MIGRATE tips */
const char *MIGRATE_tips[] = {
"nondeterministic_output",
NULL
};

/* MIGRATE key_or_empty_string argument table */
struct redisCommandArg MIGRATE_key_or_empty_string_Subargs[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"empty_string", ARG_TYPE_PURE_TOKEN, -1, "", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* MIGRATE authentication username_password argument table */
struct redisCommandArg MIGRATE_authentication_username_password_Subargs[] = {
{"username", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"password", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* MIGRATE authentication argument table */
struct redisCommandArg MIGRATE_authentication_Subargs[] = {
{"password", ARG_TYPE_STRING, -1, "AUTH", NULL, "4.0.7", CMD_ARG_OPTIONAL},
{"username_password", ARG_TYPE_BLOCK, -1, "AUTH2", NULL, "6.0.0", CMD_ARG_OPTIONAL, .subargs=MIGRATE_authentication_username_password_Subargs},
{0}
};

/* MIGRATE argument table */
struct redisCommandArg MIGRATE_Args[] = {
{"host", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"port", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key_or_empty_string", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=MIGRATE_key_or_empty_string_Subargs},
{"destination-db", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"timeout", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"copy", ARG_TYPE_PURE_TOKEN, -1, "COPY", NULL, "3.0.0", CMD_ARG_OPTIONAL},
{"replace", ARG_TYPE_PURE_TOKEN, -1, "REPLACE", NULL, "3.0.0", CMD_ARG_OPTIONAL},
{"authentication", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=MIGRATE_authentication_Subargs},
{0}
};

```

```

{"key", ARG_TYPE_KEY, 1, "KEYS", NULL, "3.0.6", CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** MOVE *****/

/* MOVE history */
#define MOVE_History NULL

/* MOVE tips */
#define MOVE_tips NULL

/* MOVE argument table */
struct redisCommandArg MOVE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"db", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** OBJECT ENCODING *****/

/* OBJECT ENCODING history */
#define OBJECT_ENCODING_History NULL

/* OBJECT ENCODING tips */
const char *OBJECT_ENCODING_tips[] = {
"nondeterministic_output",
NULL
};

/* OBJECT ENCODING argument table */
struct redisCommandArg OBJECT_ENCODING_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** OBJECT FREQ *****/

/* OBJECT FREQ history */
#define OBJECT_FREQ_History NULL

/* OBJECT FREQ tips */
const char *OBJECT_FREQ_tips[] = {
"nondeterministic_output",
NULL
};

/* OBJECT FREQ argument table */
struct redisCommandArg OBJECT_FREQ_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

};

/***** OBJECT HELP *****/

/* OBJECT HELP history */
#define OBJECT_HELP_History NULL

/* OBJECT HELP tips */
#define OBJECT_HELP_tips NULL

/***** OBJECT IDLETIME *****/

/* OBJECT IDLETIME history */
#define OBJECT_IDLETIME_History NULL

/* OBJECT IDLETIME tips */
const char *OBJECT_IDLETIME_tips[] = {
    "nondeterministic_output",
    NULL
};

/* OBJECT IDLETIME argument table */
struct redisCommandArg OBJECT_IDLETIME_Args[] = {
    {"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/***** OBJECT REFCOUNT *****/

/* OBJECT REFCOUNT history */
#define OBJECT_REFCOUNT_History NULL

/* OBJECT REFCOUNT tips */
const char *OBJECT_REFCOUNT_tips[] = {
    "nondeterministic_output",
    NULL
};

/* OBJECT REFCOUNT argument table */
struct redisCommandArg OBJECT_REFCOUNT_Args[] = {
    {"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* OBJECT command table */
struct redisCommand OBJECT_Subcommands[] = {
    {"encoding", "Inspect the internal encoding of a Redis
object", "0(1)", "2.2.3", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, OBJECT_ENCODING,
{{NULL, CMD_KEY_RO, KSPEC_BS_INDEX, .bs.index={2}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, .args=OBJECT_ENCODING_Args},
    {"freq", "Get the logarithmic access frequency counter of a Redis

```

```

object", "0(1)", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, OBJECT_FREQ_Hi
{{NULL, CMD_KEY_R0, KSPEC_BS_INDEX, .bs.index={2}, KSPEC_FK_RANGE, .fk.range=
{0, 1, 0}}}, .args=OBJECT_FREQ_Args},
{"help", "Show helpful text about the different
subcommands", "0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, OBJECT_HE
{"idletime", "Get the time since a Redis object was last
accessed", "0(1)", "2.2.3", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, OBJECT_IDLET
{{NULL, CMD_KEY_R0, KSPEC_BS_INDEX, .bs.index={2}, KSPEC_FK_RANGE, .fk.range=
{0, 1, 0}}}, .args=OBJECT_IDLETIME_Args},
{"refcount", "Get the number of references to the value of the
key", "0(1)", "2.2.3", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, OBJECT_REFCOUNT_H
{{NULL, CMD_KEY_R0, KSPEC_BS_INDEX, .bs.index={2}, KSPEC_FK_RANGE, .fk.range=
{0, 1, 0}}}, .args=OBJECT_REFCOUNT_Args},
{0}
};

/***** OBJECT *****/

/* OBJECT history */
#define OBJECT_History NULL

/* OBJECT tips */
#define OBJECT_tips NULL

/***** PERSIST *****/

/* PERSIST history */
#define PERSIST_History NULL

/* PERSIST tips */
#define PERSIST_tips NULL

/* PERSIST argument table */
struct redisCommandArg PERSIST_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** PEXPIRE *****/

/* PEXPIRE history */
commandHistory PEXPIRE_History[] = {
{"7.0.0", "Added options: `NX`, `XX`, `GT` and `LT`."},
{0}
};

/* PEXPIRE tips */
#define PEXPIRE_tips NULL

/* PEXPIRE condition argument table */
struct redisCommandArg PEXPIRE_condition_Subargs[] = {

```

```

{"nx", ARG_TYPE_PURE_TOKEN, -1, "NX", NULL, NULL, CMD_ARG_NONE},
{"xx", ARG_TYPE_PURE_TOKEN, -1, "XX", NULL, NULL, CMD_ARG_NONE},
{"gt", ARG_TYPE_PURE_TOKEN, -1, "GT", NULL, NULL, CMD_ARG_NONE},
{"lt", ARG_TYPE_PURE_TOKEN, -1, "LT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* PEXPIRE argument table */
struct redisCommandArg PEXPIRE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"milliseconds", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"condition", ARG_TYPE_ONEOF, -1, NULL, NULL, "7.0.0", CMD_ARG_OPTIONAL, .subargs=PEXPIRE
{0}
};

/***** PEXPIREAT *****/

/* PEXPIREAT history */
commandHistory PEXPIREAT_History[] = {
{"7.0.0", "Added options: `NX`, `XX`, `GT` and `LT`."},
{0}
};

/* PEXPIREAT tips */
#define PEXPIREAT_tips NULL

/* PEXPIREAT condition argument table */
struct redisCommandArg PEXPIREAT_condition_Subargs[] = {
{"nx", ARG_TYPE_PURE_TOKEN, -1, "NX", NULL, NULL, CMD_ARG_NONE},
{"xx", ARG_TYPE_PURE_TOKEN, -1, "XX", NULL, NULL, CMD_ARG_NONE},
{"gt", ARG_TYPE_PURE_TOKEN, -1, "GT", NULL, NULL, CMD_ARG_NONE},
{"lt", ARG_TYPE_PURE_TOKEN, -1, "LT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* PEXPIREAT argument table */
struct redisCommandArg PEXPIREAT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"unix-time-milliseconds", ARG_TYPE_UNIX_TIME, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"condition", ARG_TYPE_ONEOF, -1, NULL, NULL, "7.0.0", CMD_ARG_OPTIONAL, .subargs=PEXPIRE
{0}
};

/***** PEXPIRETIME *****/

/* PEXPIRETIME history */
#define PEXPIRETIME_History NULL

/* PEXPIRETIME tips */
#define PEXPIRETIME_tips NULL

```



```

/* PEXPIRETIME argument table */
struct redisCommandArg PEXPIRETIME_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** PTTL *****/

/* PTTL history */
commandHistory PTTL_History[] = {
{"2.8.0", "Added the -2 reply."},
{0}
};

/* PTTL tips */
const char *PTTL_tips[] = {
"nondeterministic_output",
NULL
};

/* PTTL argument table */
struct redisCommandArg PTTL_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** RANDOMKEY *****/

/* RANDOMKEY history */
#define RANDOMKEY_History NULL

/* RANDOMKEY tips */
const char *RANDOMKEY_tips[] = {
"request_policy:all_shards",
"nondeterministic_output",
NULL
};

/***** RENAME *****/

/* RENAME history */
#define RENAME_History NULL

/* RENAME tips */
#define RENAME_tips NULL

/* RENAME argument table */
struct redisCommandArg RENAME_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"newkey", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

};

/***** RENAMENX *****/

/* RENAMENX history */
commandHistory RENAMENX_History[] = {
{"3.2.0", "The command no longer returns an error when source and destination names are the same."},
{0}
};

/* RENAMENX tips */
#define RENAMENX_tips NULL

/* RENAMENX argument table */
struct redisCommandArg RENAMENX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"newkey", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** RESTORE *****/

/* RESTORE history */
commandHistory RESTORE_History[] = {
{"3.0.0", "Added the `REPLACE` modifier."},
{"5.0.0", "Added the `ABSTTL` modifier."},
{"5.0.0", "Added the `IDLETIME` and `FREQ` options."},
{0}
};

/* RESTORE tips */
#define RESTORE_tips NULL

/* RESTORE argument table */
struct redisCommandArg RESTORE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"ttl", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"serialized-value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"replace", ARG_TYPE_PURE_TOKEN, -1, "REPLACE", NULL, "3.0.0", CMD_ARG_OPTIONAL},
{"absttl", ARG_TYPE_PURE_TOKEN, -1, "ABSTTL", NULL, "5.0.0", CMD_ARG_OPTIONAL},
{"seconds", ARG_TYPE_INTEGER, -1, "IDLETIME", NULL, "5.0.0", CMD_ARG_OPTIONAL},
{"frequency", ARG_TYPE_INTEGER, -1, "FREQ", NULL, "5.0.0", CMD_ARG_OPTIONAL},
{0}
};

/***** SCAN *****/

/* SCAN history */
commandHistory SCAN_History[] = {
{"6.0.0", "Added the `TYPE` subcommand."},

```

```

{0}
};

/* SCAN tips */
const char *SCAN_tips[] = {
"nondeterministic_output",
"request_policy:special",
NULL
};

/* SCAN argument table */
struct redisCommandArg SCAN_Args[] = {
{"cursor", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"pattern", ARG_TYPE_PATTERN, -1, "MATCH", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{"type", ARG_TYPE_STRING, -1, "TYPE", NULL, "6.0.0", CMD_ARG_OPTIONAL},
{0}
};

/***** SORT *****/

/* SORT history */
#define SORT_History NULL

/* SORT tips */
#define SORT_tips NULL

/* SORT offset_count argument table */
struct redisCommandArg SORT_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SORT order argument table */
struct redisCommandArg SORT_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},
{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SORT argument table */
struct redisCommandArg SORT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"pattern", ARG_TYPE_PATTERN, 1, "BY", NULL, NULL, CMD_ARG_OPTIONAL},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=SORT_offset_count_Subargs},
{"pattern", ARG_TYPE_PATTERN, 1, "GET", NULL, NULL, CMD_ARG_OPTIONAL|CMD_ARG_MULTIPLE|CMD_ARG_REPEAT},
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=SORT_order_Subargs},
{"sorting", ARG_TYPE_PURE_TOKEN, -1, "ALPHA", NULL, NULL, CMD_ARG_OPTIONAL},
{"destination", ARG_TYPE_KEY, 2, "STORE", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

```

```

};

/***** SORT_R0 *****/

/* SORT_R0 history */
#define SORT_R0_History NULL

/* SORT_R0 tips */
#define SORT_R0_tips NULL

/* SORT_R0 offset_count argument table */
struct redisCommandArg SORT_R0_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SORT_R0 order argument table */
struct redisCommandArg SORT_R0_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},
{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SORT_R0 argument table */
struct redisCommandArg SORT_R0_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"pattern", ARG_TYPE_PATTERN, 1, "BY", NULL, NULL, CMD_ARG_OPTIONAL},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=SORT_R0_offset_count_Subargs},
{"pattern", ARG_TYPE_PATTERN, 1, "GET", NULL, NULL, CMD_ARG_OPTIONAL|CMD_ARG_MULTIPLE|CMD_ARG_REQUIRED},
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=SORT_R0_order_Subargs},
{"sorting", ARG_TYPE_PURE_TOKEN, -1, "ALPHA", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** TOUCH *****/

/* TOUCH history */
#define TOUCH_History NULL

/* TOUCH tips */
const char *TOUCH_tips[] = {
"request_policy:multi_shard",
"response_policy:agg_sum",
NULL
};

/* TOUCH argument table */
struct redisCommandArg TOUCH_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

```

```

};

/***** TTL *****/

/* TTL history */
commandHistory TTL_History[] = {
{"2.8.0", "Added the -2 reply."},
{0}
};

/* TTL tips */
const char *TTL_tips[] = {
"nondeterministic_output",
NULL
};

/* TTL argument table */
struct redisCommandArg TTL_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** TYPE *****/

/* TYPE history */
#define TYPE_History NULL

/* TYPE tips */
#define TYPE_tips NULL

/* TYPE argument table */
struct redisCommandArg TYPE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** UNLINK *****/

/* UNLINK history */
#define UNLINK_History NULL

/* UNLINK tips */
const char *UNLINK_tips[] = {
"request_policy:multi_shard",
"response_policy:agg_sum",
NULL
};

/* UNLINK argument table */
struct redisCommandArg UNLINK_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},

```

```

{0}
};

/***** WAIT *****/

/* WAIT history */
#define WAIT_History NULL

/* WAIT tips */
const char *WAIT_tips[] = {
    "request_policy:all_shards",
    "response_policy:agg_min",
    NULL
};

/* WAIT argument table */
struct redisCommandArg WAIT_Args[] = {
    {"numreplicas", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"timeout", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/***** GEOADD *****/

/* GEOADD history */
commandHistory GEOADD_History[] = {
    {"6.2.0", "Added the `CH`, `NX` and `XX` options."},
    {0}
};

/* GEOADD tips */
#define GEOADD_tips NULL

/* GEOADD condition argument table */
struct redisCommandArg GEOADD_condition_Subargs[] = {
    {"nx", ARG_TYPE_PURE_TOKEN, -1, "NX", NULL, NULL, CMD_ARG_NONE},
    {"xx", ARG_TYPE_PURE_TOKEN, -1, "XX", NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* GEOADD longitude_latitude_member argument table */
struct redisCommandArg GEOADD_longitude_latitude_member_Subargs[] = {
    {"longitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"latitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* GEOADD argument table */
struct redisCommandArg GEOADD_Args[] = {
    {"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{"condition", ARG_TYPE_ONEOF, -1, NULL, NULL, "6.2.0", CMD_ARG_OPTIONAL, .subargs=GEOADD_
{"change", ARG_TYPE_PURE_TOKEN, -1, "CH", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{"longitude_latitude_member", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .su
{0}
};

/***** GEODIST *****/

/* GEODIST history */
#define GEODIST_History NULL

/* GEODIST tips */
#define GEODIST_tips NULL

/* GEODIST unit argument table */
struct redisCommandArg GEODIST_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEODIST argument table */
struct redisCommandArg GEODIST_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member1", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"member2", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEODIST_unit_Su
{0}
};

/***** GEOHASH *****/

/* GEOHASH history */
#define GEOHASH_History NULL

/* GEOHASH tips */
#define GEOHASH_tips NULL

/* GEOHASH argument table */
struct redisCommandArg GEOHASH_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** GEOPOS *****/

/* GEOPOS history */
#define GEOPOS_History NULL

```

```

/* GEOPOS tips */
#define GEOPOS_tips NULL

/* GEOPOS argument table */
struct redisCommandArg GEOPOS_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** GEORADIUS *****/

/* GEORADIUS history */
commandHistory GEORADIUS_History[] = {
{"6.2.0", "Added the `ANY` option for `COUNT`."},
{0}
};

/* GEORADIUS tips */
#define GEORADIUS_tips NULL

/* GEORADIUS unit argument table */
struct redisCommandArg GEORADIUS_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUS count argument table */
struct redisCommandArg GEORADIUS_count_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_NONE},
{"any", ARG_TYPE_PURE_TOKEN, -1, "ANY", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{0}
};

/* GEORADIUS order argument table */
struct redisCommandArg GEORADIUS_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},
{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUS argument table */
struct redisCommandArg GEORADIUS_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"longitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"latitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"radius", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},

```



```

{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEORADIUS_unit_Subargs},
{"withcoord", ARG_TYPE_PURE_TOKEN, -1, "WITHCOORD", NULL, NULL, CMD_ARG_OPTIONAL},
{"withdist", ARG_TYPE_PURE_TOKEN, -1, "WITHDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{"withhash", ARG_TYPE_PURE_TOKEN, -1, "WITHHASH", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUS_count_Subargs},
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUS_order_Subargs},
{"key", ARG_TYPE_KEY, 1, "STORE", NULL, NULL, CMD_ARG_OPTIONAL},
{"key", ARG_TYPE_KEY, 2, "STOREDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** GEORADIUSBYMEMBER *****/

/* GEORADIUSBYMEMBER history */
#define GEORADIUSBYMEMBER_History NULL

/* GEORADIUSBYMEMBER tips */
#define GEORADIUSBYMEMBER_tips NULL

/* GEORADIUSBYMEMBER unit argument table */
struct redisCommandArg GEORADIUSBYMEMBER_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUSBYMEMBER count argument table */
struct redisCommandArg GEORADIUSBYMEMBER_count_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_NONE},
{"any", ARG_TYPE_PURE_TOKEN, -1, "ANY", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* GEORADIUSBYMEMBER order argument table */
struct redisCommandArg GEORADIUSBYMEMBER_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},
{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUSBYMEMBER argument table */
struct redisCommandArg GEORADIUSBYMEMBER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"radius", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEORADIUSBYMEMBER_unit_Subargs},
{"withcoord", ARG_TYPE_PURE_TOKEN, -1, "WITHCOORD", NULL, NULL, CMD_ARG_OPTIONAL},
{"withdist", ARG_TYPE_PURE_TOKEN, -1, "WITHDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{"withhash", ARG_TYPE_PURE_TOKEN, -1, "WITHHASH", NULL, NULL, CMD_ARG_OPTIONAL},

```

```

{"count", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUSBYMEM
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUSBYMEM
{"key", ARG_TYPE_KEY, 1, "STORE", NULL, NULL, CMD_ARG_OPTIONAL},
{"key", ARG_TYPE_KEY, 2, "STOREDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** GEORADIUSBYMEMBER_R0 *****/

/* GEORADIUSBYMEMBER_R0 history */
#define GEORADIUSBYMEMBER_R0_History NULL

/* GEORADIUSBYMEMBER_R0 tips */
#define GEORADIUSBYMEMBER_R0_tips NULL

/* GEORADIUSBYMEMBER_R0 unit argument table */
struct redisCommandArg GEORADIUSBYMEMBER_R0_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUSBYMEMBER_R0 count argument table */
struct redisCommandArg GEORADIUSBYMEMBER_R0_count_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_NONE},
{"any", ARG_TYPE_PURE_TOKEN, -1, "ANY", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* GEORADIUSBYMEMBER_R0 order argument table */
struct redisCommandArg GEORADIUSBYMEMBER_R0_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},
{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUSBYMEMBER_R0 argument table */
struct redisCommandArg GEORADIUSBYMEMBER_R0_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"radius", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEORADIUSBYMEMBER_R
{"withcoord", ARG_TYPE_PURE_TOKEN, -1, "WITHCOORD", NULL, NULL, CMD_ARG_OPTIONAL},
{"withdist", ARG_TYPE_PURE_TOKEN, -1, "WITHDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{"withhash", ARG_TYPE_PURE_TOKEN, -1, "WITHHASH", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUSBYMEM
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUSBYMEM
{0}
};

```

```

/***** GEORADIUS_R0 *****/

/* GEORADIUS_R0 history */
commandHistory GEORADIUS_R0_History[] = {
{"6.2.0", "Added the `ANY` option for `COUNT`."},
{0}
};

/* GEORADIUS_R0 tips */
#define GEORADIUS_R0_tips NULL

/* GEORADIUS_R0 unit argument table */
struct redisCommandArg GEORADIUS_R0_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUS_R0 count argument table */
struct redisCommandArg GEORADIUS_R0_count_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_NONE},
{"any", ARG_TYPE_PURE_TOKEN, -1, "ANY", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{0}
};

/* GEORADIUS_R0 order argument table */
struct redisCommandArg GEORADIUS_R0_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},
{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEORADIUS_R0 argument table */
struct redisCommandArg GEORADIUS_R0_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"longitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"latitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"radius", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEORADIUS_R0_unit_S
{"withcoord", ARG_TYPE_PURE_TOKEN, -1, "WITHCOORD", NULL, NULL, CMD_ARG_OPTIONAL},
{"withdist", ARG_TYPE_PURE_TOKEN, -1, "WITHDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{"withhash", ARG_TYPE_PURE_TOKEN, -1, "WITHHASH", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUS_R0_c
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEORADIUS_R0_o
{0}
};

/***** GEOSearch *****/

```

```

/* GEOSEARCH history */
#define GEOSEARCH_History NULL

/* GEOSEARCH tips */
#define GEOSEARCH_tips NULL

/* GEOSEARCH from longitude_latitude argument table */
struct redisCommandArg GEOSEARCH_from_longitude_latitude_Subargs[] = {
{"longitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"latitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEOSEARCH from argument table */
struct redisCommandArg GEOSEARCH_from_Subargs[] = {
{"member", ARG_TYPE_STRING, -1, "FROMMEMBER", NULL, NULL, CMD_ARG_NONE},
{"longitude_latitude", ARG_TYPE_BLOCK, -1, "FROMLONLAT", NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCH_from_longitude_latitude_Subargs,
{0}
};

/* GEOSEARCH by circle unit argument table */
struct redisCommandArg GEOSEARCH_by_circle_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEOSEARCH by circle argument table */
struct redisCommandArg GEOSEARCH_by_circle_Subargs[] = {
{"radius", ARG_TYPE_DOUBLE, -1, "BYRADIUS", NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCH_by_circle_unit_Subargs,
{0}
};

/* GEOSEARCH by box unit argument table */
struct redisCommandArg GEOSEARCH_by_box_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEOSEARCH by box argument table */
struct redisCommandArg GEOSEARCH_by_box_Subargs[] = {
{"width", ARG_TYPE_DOUBLE, -1, "BYBOX", NULL, NULL, CMD_ARG_NONE},
{"height", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCH_by_box_unit_Subargs,

```

```

{0}
};

/* GEOSEARCH by argument table */
struct redisCommandArg GEOSEARCH_by_Subargs[] = {
{"circle", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCH_by_circ
{"box", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCH_by_box_Sub
{0}
};

/* GEOSEARCH order argument table */
struct redisCommandArg GEOSEARCH_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},
{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEOSEARCH count argument table */
struct redisCommandArg GEOSEARCH_count_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_NONE},
{"any", ARG_TYPE_PURE_TOKEN, -1, "ANY", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* GEOSEARCH argument table */
struct redisCommandArg GEOSEARCH_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"from", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCH_from_Suba
{"by", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCH_by_Subargs}
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEOSEARCH_orde
{"count", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEOSEARCH_coun
{"withcoord", ARG_TYPE_PURE_TOKEN, -1, "WITHCOORD", NULL, NULL, CMD_ARG_OPTIONAL},
{"withdist", ARG_TYPE_PURE_TOKEN, -1, "WITHDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{"withhash", ARG_TYPE_PURE_TOKEN, -1, "WITHHASH", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** GEOSEARCHSTORE *****/

/* GEOSEARCHSTORE history */
#define GEOSEARCHSTORE_History NULL

/* GEOSEARCHSTORE tips */
#define GEOSEARCHSTORE_tips NULL

/* GEOSEARCHSTORE from longitude_latitude argument table */
struct redisCommandArg GEOSEARCHSTORE_from_longitude_latitude_Subargs[] = {
{"longitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"latitude", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/* GEOSEARCHSTORE from argument table */
struct redisCommandArg GEOSEARCHSTORE_from_Subargs[] = {
{"member", ARG_TYPE_STRING, -1, "FROMMEMBER", NULL, NULL, CMD_ARG_NONE},
{"longitude_latitude", ARG_TYPE_BLOCK, -1, "FROMLONLAT", NULL, NULL, CMD_ARG_NONE, .subargs=
{0}
};

/* GEOSEARCHSTORE by circle unit argument table */
struct redisCommandArg GEOSEARCHSTORE_by_circle_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEOSEARCHSTORE by circle argument table */
struct redisCommandArg GEOSEARCHSTORE_by_circle_Subargs[] = {
{"radius", ARG_TYPE_DOUBLE, -1, "BYRADIUS", NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCHSTORE_by_c
{0}
};

/* GEOSEARCHSTORE by box unit argument table */
struct redisCommandArg GEOSEARCHSTORE_by_box_unit_Subargs[] = {
{"m", ARG_TYPE_PURE_TOKEN, -1, "M", NULL, NULL, CMD_ARG_NONE},
{"km", ARG_TYPE_PURE_TOKEN, -1, "KM", NULL, NULL, CMD_ARG_NONE},
{"ft", ARG_TYPE_PURE_TOKEN, -1, "FT", NULL, NULL, CMD_ARG_NONE},
{"mi", ARG_TYPE_PURE_TOKEN, -1, "MI", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEOSEARCHSTORE by box argument table */
struct redisCommandArg GEOSEARCHSTORE_by_box_Subargs[] = {
{"width", ARG_TYPE_DOUBLE, -1, "BYBOX", NULL, NULL, CMD_ARG_NONE},
{"height", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"unit", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCHSTORE_by_b
{0}
};

/* GEOSEARCHSTORE by argument table */
struct redisCommandArg GEOSEARCHSTORE_by_Subargs[] = {
{"circle", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCHSTORE_by
{"box", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCHSTORE_by_bo
{0}
};

/* GEOSEARCHSTORE order argument table */
struct redisCommandArg GEOSEARCHSTORE_order_Subargs[] = {
{"asc", ARG_TYPE_PURE_TOKEN, -1, "ASC", NULL, NULL, CMD_ARG_NONE},

```

```

{"desc", ARG_TYPE_PURE_TOKEN, -1, "DESC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GEOSEARCHSTORE count argument table */
struct redisCommandArg GEOSEARCHSTORE_count_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_NONE},
{"any", ARG_TYPE_PURE_TOKEN, -1, "ANY", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* GEOSEARCHSTORE argument table */
struct redisCommandArg GEOSEARCHSTORE_Args[] = {
{"destination", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"source", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"from", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCHSTORE_from},
{"by", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=GEOSEARCHSTORE_by_Sub},
{"order", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEOSEARCHSTORE_order},
{"count", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GEOSEARCHSTORE_count_Subargs},
{"storedist", ARG_TYPE_PURE_TOKEN, -1, "STOREDIST", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** HDEL *****/

/* HDEL history */
commandHistory HDEL_History[] = {
{"2.4.0", "Accepts multiple `field` arguments."},
{0}
};

/* HDEL tips */
#define HDEL_tips NULL

/* HDEL argument table */
struct redisCommandArg HDEL_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** HEXISTS *****/

/* HEXISTS history */
#define HEXISTS_History NULL

/* HEXISTS tips */
#define HEXISTS_tips NULL

/* HEXISTS argument table */
struct redisCommandArg HEXISTS_Args[] = {

```

```

{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HGET *****/

/* HGET history */
#define HGET_History NULL

/* HGET tips */
#define HGET_tips NULL

/* HGET argument table */
struct redisCommandArg HGET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HGETALL *****/

/* HGETALL history */
#define HGETALL_History NULL

/* HGETALL tips */
const char *HGETALL_tips[] = {
"nondeterministic_output_order",
NULL
};

/* HGETALL argument table */
struct redisCommandArg HGETALL_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HINCRBY *****/

/* HINCRBY history */
#define HINCRBY_History NULL

/* HINCRBY tips */
#define HINCRBY_tips NULL

/* HINCRBY argument table */
struct redisCommandArg HINCRBY_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"increment", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```



```

};

/***** HINCRBYFLOAT *****/

/* HINCRBYFLOAT history */
#define HINCRBYFLOAT_History NULL

/* HINCRBYFLOAT tips */
#define HINCRBYFLOAT_tips NULL

/* HINCRBYFLOAT argument table */
struct redisCommandArg HINCRBYFLOAT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"increment", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HKEYS *****/

/* HKEYS history */
#define HKEYS_History NULL

/* HKEYS tips */
const char *HKEYS_tips[] = {
"nondeterministic_output_order",
NULL
};

/* HKEYS argument table */
struct redisCommandArg HKEYS_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HLEN *****/

/* HLEN history */
#define HLEN_History NULL

/* HLEN tips */
#define HLEN_tips NULL

/* HLEN argument table */
struct redisCommandArg HLEN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HMGET *****/

```

```

/* HMGET history */
#define HMGET_History NULL

/* HMGET tips */
#define HMGET_tips NULL

/* HMGET argument table */
struct redisCommandArg HMGET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** HMSET *****/

/* HMSET history */
#define HMSET_History NULL

/* HMSET tips */
#define HMSET_tips NULL

/* HMSET field_value argument table */
struct redisCommandArg HMSET_field_value_Subargs[] = {
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* HMSET argument table */
struct redisCommandArg HMSET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=HMSET_fi
{0}
};

/***** HRANDFIELD *****/

/* HRANDFIELD history */
#define HRANDFIELD_History NULL

/* HRANDFIELD tips */
const char *HRANDFIELD_tips[] = {
"nondeterministic_output",
NULL
};

/* HRANDFIELD options argument table */
struct redisCommandArg HRANDFIELD_options_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"withvalues", ARG_TYPE_PURE_TOKEN, -1, "WITHVALUES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
}

```

```

};

/* HRANDFIELD argument table */
struct redisCommandArg HRANDFIELD_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"options", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=HRANDFIELD_o
{0}
};

/***** HSCAN *****/

/* HSCAN history */
#define HSCAN_History NULL

/* HSCAN tips */
const char *HSCAN_tips[] = {
"nondeterministic_output",
NULL
};

/* HSCAN argument table */
struct redisCommandArg HSCAN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"cursor", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"pattern", ARG_TYPE_PATTERN, -1, "MATCH", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** HSET *****/

/* HSET history */
commandHistory HSET_History[] = {
{"4.0.0", "Accepts multiple `field` and `value` arguments."},
{0}
};

/* HSET tips */
#define HSET_tips NULL

/* HSET field_value argument table */
struct redisCommandArg HSET_field_value_Subargs[] = {
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* HSET argument table */
struct redisCommandArg HSET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=HSET_fie

```

```

{0}
};

/***** HSETNX *****/

/* HSETNX history */
#define HSETNX_History NULL

/* HSETNX tips */
#define HSETNX_tips NULL

/* HSETNX argument table */
struct redisCommandArg HSETNX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HSTRLEN *****/

/* HSTRLEN history */
#define HSTRLEN_History NULL

/* HSTRLEN tips */
#define HSTRLEN_tips NULL

/* HSTRLEN argument table */
struct redisCommandArg HSTRLEN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** HVALS *****/

/* HVALS history */
#define HVALS_History NULL

/* HVALS tips */
const char *HVALS_tips[] = {
"nondeterministic_output_order",
NULL
};

/* HVALS argument table */
struct redisCommandArg HVALS_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/***** PFADD *****/

/* PFADD history */
#define PFADD_History NULL

/* PFADD tips */
#define PFADD_tips NULL

/* PFADD argument table */
struct redisCommandArg PFADD_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** PFCOUNT *****/

/* PFCOUNT history */
#define PFCOUNT_History NULL

/* PFCOUNT tips */
#define PFCOUNT_tips NULL

/* PFCOUNT argument table */
struct redisCommandArg PFCOUNT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** PFDEBUG *****/

/* PFDEBUG history */
#define PFDEBUG_History NULL

/* PFDEBUG tips */
#define PFDEBUG_tips NULL

/***** PFMERGE *****/

/* PFMERGE history */
#define PFMERGE_History NULL

/* PFMERGE tips */
#define PFMERGE_tips NULL

/* PFMERGE argument table */
struct redisCommandArg PFMERGE_Args[] = {
{"destkey", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"sourcekey", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

```

```

/***** PFSELFTEST *****/

/* PFSELFTEST history */
#define PFSELFTEST_History NULL

/* PFSELFTEST tips */
#define PFSELFTEST_tips NULL

/***** BLMOVE *****/

/* BLMOVE history */
#define BLMOVE_History NULL

/* BLMOVE tips */
#define BLMOVE_tips NULL

/* BLMOVE wherefrom argument table */
struct redisCommandArg BLMOVE_wherefrom_Subargs[] = {
{"left", ARG_TYPE_PURE_TOKEN, -1, "LEFT", NULL, NULL, CMD_ARG_NONE},
{"right", ARG_TYPE_PURE_TOKEN, -1, "RIGHT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BLMOVE whereto argument table */
struct redisCommandArg BLMOVE_whereto_Subargs[] = {
{"left", ARG_TYPE_PURE_TOKEN, -1, "LEFT", NULL, NULL, CMD_ARG_NONE},
{"right", ARG_TYPE_PURE_TOKEN, -1, "RIGHT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BLMOVE argument table */
struct redisCommandArg BLMOVE_Args[] = {
{"source", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"destination", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"wherefrom", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=BLMOVE_wherefrom_Subargs},
{"whereto", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=BLMOVE_whereto_Subargs},
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** BLMPop *****/

/* BLMPop history */
#define BLMPop_History NULL

/* BLMPop tips */
#define BLMPop_tips NULL

/* BLMPop where argument table */
struct redisCommandArg BLMPop_where_Subargs[] = {

```

```

{"left", ARG_TYPE_PURE_TOKEN, -1, "LEFT", NULL, NULL, CMD_ARG_NONE},
{"right", ARG_TYPE_PURE_TOKEN, -1, "RIGHT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BLPOP argument table */
struct redisCommandArg BLPOP_Args[] = {
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"where", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=BLPOP_where_Subar},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** BLPOP *****/

/* BLPOP history */
commandHistory BLPOP_History[] = {
{"6.0.0", "`timeout` is interpreted as a double instead of an integer."},
{0}
};

/* BLPOP tips */
#define BLPOP_tips NULL

/* BLPOP argument table */
struct redisCommandArg BLPOP_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** BRPOP *****/

/* BRPOP history */
commandHistory BRPOP_History[] = {
{"6.0.0", "`timeout` is interpreted as a double instead of an integer."},
{0}
};

/* BRPOP tips */
#define BRPOP_tips NULL

/* BRPOP argument table */
struct redisCommandArg BRPOP_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/***** BRPOPLUSH *****/

/* BRPOPLUSH history */
commandHistory BRPOPLUSH_History[] = {
{"6.0.0", "`timeout` is interpreted as a double instead of an integer."},
{0}
};

/* BRPOPLUSH tips */
#define BRPOPLUSH_tips NULL

/* BRPOPLUSH argument table */
struct redisCommandArg BRPOPLUSH_Args[] = {
{"source", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"destination", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LINDEX *****/

/* LINDEX history */
#define LINDEX_History NULL

/* LINDEX tips */
#define LINDEX_tips NULL

/* LINDEX argument table */
struct redisCommandArg LINDEX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"index", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LINSERT *****/

/* LINSERT history */
#define LINSERT_History NULL

/* LINSERT tips */
#define LINSERT_tips NULL

/* LINSERT where argument table */
struct redisCommandArg LINSERT_where_Subargs[] = {
{"before", ARG_TYPE_PURE_TOKEN, -1, "BEFORE", NULL, NULL, CMD_ARG_NONE},
{"after", ARG_TYPE_PURE_TOKEN, -1, "AFTER", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* LINSERT argument table */
struct redisCommandArg LINSERT_Args[] = {

```



```

{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"where", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=LINSERT_where_Subargs},
{"pivot", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LLEN *****/

/* LLEN history */
#define LLEN_History NULL

/* LLEN tips */
#define LLEN_tips NULL

/* LLEN argument table */
struct redisCommandArg LLEN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LMOVE *****/

/* LMOVE history */
#define LMOVE_History NULL

/* LMOVE tips */
#define LMOVE_tips NULL

/* LMOVE wherefrom argument table */
struct redisCommandArg LMOVE_wherefrom_Subargs[] = {
{"left", ARG_TYPE_PURE_TOKEN, -1, "LEFT", NULL, NULL, CMD_ARG_NONE},
{"right", ARG_TYPE_PURE_TOKEN, -1, "RIGHT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* LMOVE whereto argument table */
struct redisCommandArg LMOVE_whereto_Subargs[] = {
{"left", ARG_TYPE_PURE_TOKEN, -1, "LEFT", NULL, NULL, CMD_ARG_NONE},
{"right", ARG_TYPE_PURE_TOKEN, -1, "RIGHT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* LMOVE argument table */
struct redisCommandArg LMOVE_Args[] = {
{"source", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"destination", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"wherefrom", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=LMOVE_wherefrom_Subargs},
{"whereto", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=LMOVE_whereto_Subargs},
{0}
};

```

```

/***** LMPOP *****/

/* LMPOP history */
#define LMPOP_History NULL

/* LMPOP tips */
#define LMPOP_tips NULL

/* LMPOP where argument table */
struct redisCommandArg LMPOP_where_Subargs[] = {
{"left", ARG_TYPE_PURE_TOKEN, -1, "LEFT", NULL, NULL, CMD_ARG_NONE},
{"right", ARG_TYPE_PURE_TOKEN, -1, "RIGHT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* LMPOP argument table */
struct redisCommandArg LMPOP_Args[] = {
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"where", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=LMPOP_where_Subargs},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** LPOP *****/

/* LPOP history */
commandHistory LPOP_History[] = {
{"6.2.0", "Added the `count` argument."},
{0}
};

/* LPOP tips */
#define LPOP_tips NULL

/* LPOP argument table */
struct redisCommandArg LPOP_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, "6.2.0", CMD_ARG_OPTIONAL},
{0}
};

/***** LPOS *****/

/* LPOS history */
#define LPOS_History NULL

/* LPOS tips */
#define LPOS_tips NULL

```

```

/* LPOS argument table */
struct redisCommandArg LPOS_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"rank", ARG_TYPE_INTEGER, -1, "RANK", NULL, NULL, CMD_ARG_OPTIONAL},
{"num-matches", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{"len", ARG_TYPE_INTEGER, -1, "MAXLEN", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** LPU SH *****/

/* LPU SH history */
commandHistory LPU SH_History[] = {
{"2.4.0", "Accepts multiple `element` arguments."},
{0}
};

/* LPU SH tips */
#define LPU SH_tips NULL

/* LPU SH argument table */
struct redisCommandArg LPU SH_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** LPU SHX *****/

/* LPU SHX history */
commandHistory LPU SHX_History[] = {
{"4.0.0", "Accepts multiple `element` arguments."},
{0}
};

/* LPU SHX tips */
#define LPU SHX_tips NULL

/* LPU SHX argument table */
struct redisCommandArg LPU SHX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** LRANGE *****/

/* LRANGE history */
#define LRANGE_History NULL

```

```

/* LRANGE tips */
#define LRANGE_tips NULL

/* LRANGE argument table */
struct redisCommandArg LRANGE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"stop", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LREM *****/

/* LREM history */
#define LREM_History NULL

/* LREM tips */
#define LREM_tips NULL

/* LREM argument table */
struct redisCommandArg LREM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LSET *****/

/* LSET history */
#define LSET_History NULL

/* LSET tips */
#define LSET_tips NULL

/* LSET argument table */
struct redisCommandArg LSET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"index", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LTRIM *****/

/* LTRIM history */
#define LTRIM_History NULL

/* LTRIM tips */
#define LTRIM_tips NULL

```

```

/* LTRIM argument table */
struct redisCommandArg LTRIM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"stop", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** RPOP *****/

/* RPOP history */
commandHistory RPOP_History[] = {
{"6.2.0", "Added the `count` argument."},
{0}
};

/* RPOP tips */
#define RPOP_tips NULL

/* RPOP argument table */
struct redisCommandArg RPOP_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, "6.2.0", CMD_ARG_OPTIONAL},
{0}
};

/***** RPOPLUSH *****/

/* RPOPLUSH history */
#define RPOPLUSH_History NULL

/* RPOPLUSH tips */
#define RPOPLUSH_tips NULL

/* RPOPLUSH argument table */
struct redisCommandArg RPOPLUSH_Args[] = {
{"source", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"destination", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** RPUSH *****/

/* RPUSH history */
commandHistory RPUSH_History[] = {
{"2.4.0", "Accepts multiple `element` arguments."},
{0}
};

/* RPUSH tips */
#define RPUSH_tips NULL

```

```

/* RPush argument table */
struct redisCommandArg RPush_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** RPUSHX *****/

/* RPUSHX history */
commandHistory RPUSHX_History[] = {
{"4.0.0", "Accepts multiple `element` arguments."},
{0}
};

/* RPUSHX tips */
#define RPUSHX_tips NULL

/* RPUSHX argument table */
struct redisCommandArg RPUSHX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"element", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** PSUBSCRIBE *****/

/* PSUBSCRIBE history */
#define PSUBSCRIBE_History NULL

/* PSUBSCRIBE tips */
#define PSUBSCRIBE_tips NULL

/* PSUBSCRIBE pattern argument table */
struct redisCommandArg PSUBSCRIBE_pattern_Subargs[] = {
{"pattern", ARG_TYPE_PATTERN, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* PSUBSCRIBE argument table */
struct redisCommandArg PSUBSCRIBE_Args[] = {
{"pattern", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=PSUBSCRIBE_p
{0}
};

/***** PUBLISH *****/

/* PUBLISH history */
#define PUBLISH_History NULL

```

```

/* PUBLISH tips */
#define PUBLISH_tips NULL

/* PUBLISH argument table */
struct redisCommandArg PUBLISH_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"message", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** PUBSUB CHANNELS *****/

/* PUBSUB CHANNELS history */
#define PUBSUB_CHANNELS_History NULL

/* PUBSUB CHANNELS tips */
#define PUBSUB_CHANNELS_tips NULL

/* PUBSUB CHANNELS argument table */
struct redisCommandArg PUBSUB_CHANNELS_Args[] = {
{"pattern", ARG_TYPE_PATTERN, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** PUBSUB HELP *****/

/* PUBSUB HELP history */
#define PUBSUB_HELP_History NULL

/* PUBSUB HELP tips */
#define PUBSUB_HELP_tips NULL

/***** PUBSUB NUNPAT *****/

/* PUBSUB NUNPAT history */
#define PUBSUB_NUNPAT_History NULL

/* PUBSUB NUNPAT tips */
#define PUBSUB_NUNPAT_tips NULL

/***** PUBSUB NUMSUB *****/

/* PUBSUB NUMSUB history */
#define PUBSUB_NUMSUB_History NULL

/* PUBSUB NUMSUB tips */
#define PUBSUB_NUMSUB_tips NULL

/* PUBSUB NUMSUB argument table */
struct redisCommandArg PUBSUB_NUMSUB_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},

```

```

{0}
};

/***** PUBSUB SHARDCHANNELS *****/

/* PUBSUB SHARDCHANNELS history */
#define PUBSUB_SHARDCHANNELS_History NULL

/* PUBSUB SHARDCHANNELS tips */
#define PUBSUB_SHARDCHANNELS_tips NULL

/* PUBSUB SHARDCHANNELS argument table */
struct redisCommandArg PUBSUB_SHARDCHANNELS_Args[] = {
{"pattern", ARG_TYPE_PATTERN, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** PUBSUB SHARDNUMSUB *****/

/* PUBSUB SHARDNUMSUB history */
#define PUBSUB_SHARDNUMSUB_History NULL

/* PUBSUB SHARDNUMSUB tips */
#define PUBSUB_SHARDNUMSUB_tips NULL

/* PUBSUB SHARDNUMSUB argument table */
struct redisCommandArg PUBSUB_SHARDNUMSUB_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/* PUBSUB command table */
struct redisCommand PUBSUB_Subcommands[] = {
{"channels", "List active channels", "0(N) where N is the number of active channels, and assuming constant time pattern matching (relatively short patterns)", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_PUBSUB, PUBSUB_CHANNELS_His
{"help", "Show helpful text about the different subcommands", "0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_PUBSUB, PUBSUB_HEL
{"numpat", "Get the count of unique patterns pattern subscriptions", "0(1)", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_PUBSUB, PUBSUB_N
{"numsub", "Get the count of subscribers for channels", "0(N) for the NUMSUB subcommand, where N is the number of requested channels", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_PUBSUB, PUBSUB_NUMSUB_Histor
{"shardchannels", "List active shard channels", "0(N) where N is the number of active shard channels, and assuming constant time pattern matching (relatively short channels).", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_PUBSUB, PUBSUB_SHARDCHANNE
{"shardnumsub", "Get the count of subscribers for shard channels", "0(N) for the SHARDNUMSUB subcommand, where N is the number of requested channels", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_PUBSUB, PUBSUB_SHARDNUMSUB_H

```



```

{0}
};

/***** PUBSUB *****/

/* PUBSUB history */
#define PUBSUB_History NULL

/* PUBSUB tips */
#define PUBSUB_tips NULL

/***** PUNSUBSCRIBE *****/

/* PUNSUBSCRIBE history */
#define PUNSUBSCRIBE_History NULL

/* PUNSUBSCRIBE tips */
#define PUNSUBSCRIBE_tips NULL

/* PUNSUBSCRIBE argument table */
struct redisCommandArg PUNSUBSCRIBE_Args[] = {
{"pattern", ARG_TYPE_PATTERN, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** SPUBLISH *****/

/* SPUBLISH history */
#define SPUBLISH_History NULL

/* SPUBLISH tips */
#define SPUBLISH_tips NULL

/* SPUBLISH argument table */
struct redisCommandArg SPUBLISH_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"message", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SSUBSCRIBE *****/

/* SSUBSCRIBE history */
#define SSUBSCRIBE_History NULL

/* SSUBSCRIBE tips */
#define SSUBSCRIBE_tips NULL

/* SSUBSCRIBE argument table */
struct redisCommandArg SSUBSCRIBE_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},

```

```

{0}
};

/***** SUBSCRIBE *****/

/* SUBSCRIBE history */
#define SUBSCRIBE_History NULL

/* SUBSCRIBE tips */
#define SUBSCRIBE_tips NULL

/* SUBSCRIBE argument table */
struct redisCommandArg SUBSCRIBE_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SUNSUBSCRIBE *****/

/* SUNSUBSCRIBE history */
#define SUNSUBSCRIBE_History NULL

/* SUNSUBSCRIBE tips */
#define SUNSUBSCRIBE_tips NULL

/* SUNSUBSCRIBE argument table */
struct redisCommandArg SUNSUBSCRIBE_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** UNSUBSCRIBE *****/

/* UNSUBSCRIBE history */
#define UNSUBSCRIBE_History NULL

/* UNSUBSCRIBE tips */
#define UNSUBSCRIBE_tips NULL

/* UNSUBSCRIBE argument table */
struct redisCommandArg UNSUBSCRIBE_Args[] = {
{"channel", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** EVAL *****/

/* EVAL history */
#define EVAL_History NULL

/* EVAL tips */

```

```

#define EVAL_tips NULL

/* EVAL argument table */
struct redisCommandArg EVAL_Args[] = {
{"script", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** EVALSHA *****/

/* EVALSHA history */
#define EVALSHA_History NULL

/* EVALSHA tips */
#define EVALSHA_tips NULL

/* EVALSHA argument table */
struct redisCommandArg EVALSHA_Args[] = {
{"sha1", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** EVALSHA_R0 *****/

/* EVALSHA_R0 history */
#define EVALSHA_R0_History NULL

/* EVALSHA_R0 tips */
#define EVALSHA_R0_tips NULL

/* EVALSHA_R0 argument table */
struct redisCommandArg EVALSHA_R0_Args[] = {
{"sha1", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** EVAL_R0 *****/

/* EVAL_R0 history */
#define EVAL_R0_History NULL

/* EVAL_R0 tips */

```

```

#define EVAL_R0_tips NULL

/* EVAL_R0 argument table */
struct redisCommandArg EVAL_R0_Args[] = {
{"script", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** FCALL *****/

/* FCALL history */
#define FCALL_History NULL

/* FCALL tips */
#define FCALL_tips NULL

/* FCALL argument table */
struct redisCommandArg FCALL_Args[] = {
{"function", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** FCALL_R0 *****/

/* FCALL_R0 history */
#define FCALL_R0_History NULL

/* FCALL_R0 tips */
#define FCALL_R0_tips NULL

/* FCALL_R0 argument table */
struct redisCommandArg FCALL_R0_Args[] = {
{"function", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** FUNCTION DELETE *****/

/* FUNCTION DELETE history */
#define FUNCTION_DELETE_History NULL

/* FUNCTION DELETE tips */

```

```

const char *FUNCTION_DELETE_tips[] = {
"request_policy:all_shards",
"response_policy:all_succeeded",
NULL
};

/* FUNCTION DELETE argument table */
struct redisCommandArg FUNCTION_DELETE_Args[] = {
{"library-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** FUNCTION DUMP *****/

/* FUNCTION DUMP history */
#define FUNCTION_DUMP_History NULL

/* FUNCTION DUMP tips */
#define FUNCTION_DUMP_tips NULL

/***** FUNCTION FLUSH *****/

/* FUNCTION FLUSH history */
#define FUNCTION_FLUSH_History NULL

/* FUNCTION FLUSH tips */
const char *FUNCTION_FLUSH_tips[] = {
"request_policy:all_shards",
"response_policy:all_succeeded",
NULL
};

/* FUNCTION FLUSH async argument table */
struct redisCommandArg FUNCTION_FLUSH_async_Subargs[] = {
{"async", ARG_TYPE_PURE_TOKEN, -1, "ASYNC", NULL, NULL, CMD_ARG_NONE},
{"sync", ARG_TYPE_PURE_TOKEN, -1, "SYNC", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* FUNCTION FLUSH argument table */
struct redisCommandArg FUNCTION_FLUSH_Args[] = {
{"async", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=FUNCTION_FLUSH_async_Subargs},
{0}
};

/***** FUNCTION HELP *****/

/* FUNCTION HELP history */
#define FUNCTION_HELP_History NULL

/* FUNCTION HELP tips */

```

```

#define FUNCTION_HELP_tips NULL

/***** FUNCTION KILL *****/

/* FUNCTION KILL history */
#define FUNCTION_KILL_History NULL

/* FUNCTION KILL tips */
const char *FUNCTION_KILL_tips[] = {
    "request_policy:all_shards",
    "response_policy:one_succeeded",
    NULL
};

/***** FUNCTION LIST *****/

/* FUNCTION LIST history */
#define FUNCTION_LIST_History NULL

/* FUNCTION LIST tips */
const char *FUNCTION_LIST_tips[] = {
    "nondeterministic_output_order",
    NULL
};

/* FUNCTION LIST argument table */
struct redisCommandArg FUNCTION_LIST_Args[] = {
    {"library-name-
pattern", ARG_TYPE_STRING, -1, "LIBRARYNAME", NULL, NULL, CMD_ARG_OPTIONAL},
    {"withcode", ARG_TYPE_PURE_TOKEN, -1, "WITHCODE", NULL, NULL, CMD_ARG_OPTIONAL},
    {0}
};

/***** FUNCTION LOAD *****/

/* FUNCTION LOAD history */
#define FUNCTION_LOAD_History NULL

/* FUNCTION LOAD tips */
const char *FUNCTION_LOAD_tips[] = {
    "request_policy:all_shards",
    "response_policy:all_succeeded",
    NULL
};

/* FUNCTION LOAD argument table */
struct redisCommandArg FUNCTION_LOAD_Args[] = {
    {"replace", ARG_TYPE_PURE_TOKEN, -1, "REPLACE", NULL, NULL, CMD_ARG_OPTIONAL},
    {"function-code", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

```

```

/***** FUNCTION RESTORE *****/

/* FUNCTION RESTORE history */
#define FUNCTION_RESTORE_History NULL

/* FUNCTION RESTORE tips */
const char *FUNCTION_RESTORE_tips[] = {
    "request_policy:all_shards",
    "response_policy:all_succeeded",
    NULL
};

/* FUNCTION RESTORE policy argument table */
struct redisCommandArg FUNCTION_RESTORE_policy_Subargs[] = {
    {"flush", ARG_TYPE_PURE_TOKEN, -1, "FLUSH", NULL, NULL, CMD_ARG_NONE},
    {"append", ARG_TYPE_PURE_TOKEN, -1, "APPEND", NULL, NULL, CMD_ARG_NONE},
    {"replace", ARG_TYPE_PURE_TOKEN, -1, "REPLACE", NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* FUNCTION RESTORE argument table */
struct redisCommandArg FUNCTION_RESTORE_Args[] = {
    {"serialized-value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"policy", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=FUNCTION_RESTORE_policy_Subargs},
    {0}
};

/***** FUNCTION STATS *****/

/* FUNCTION STATS history */
#define FUNCTION_STATS_History NULL

/* FUNCTION STATS tips */
const char *FUNCTION_STATS_tips[] = {
    "nondeterministic_output",
    "request_policy:all_shards",
    "response_policy:special",
    NULL
};

/* FUNCTION command table */
struct redisCommand FUNCTION_Subcommands[] = {
    {"delete", "Delete a function by",
     name, "0(1)", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, FUNCTION_DELETE},
    {"dump", "Dump all functions into a serialized binary payload", "0(N) where N is",
     the number of",
     functions", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, FUNCTION_DUMP_Hi},
    {"flush", "Deleting all functions", "0(N) where N is the number of functions",
     deleted", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, FUNCTION_FLUSH_His},
    {"help", "Show helpful text about the different"}
};

```

```

subcommands","0(1)","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FUNCTION
{"kill","Kill the function currently in
execution.","0(1)","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FUNCTION
{"list","List information about all the functions","0(N) where N is the number
of
functions","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FUNCTION_LIST_Hi
{"load","Create a function with the given arguments (name, code,
description)","0(1) (considering compilation time is
redundant)","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FUNCTION_LOAD_H
{"restore","Restore all the functions on the given payload","0(N) where N is
the number of functions on the
payload","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FUNCTION_RESTORE_H
{"stats","Return information about the function currently running (name,
description,
duration)","0(1)","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FUNCTION_
{0}
};

/***** FUNCTION *****/

/* FUNCTION history */
#define FUNCTION_History NULL

/* FUNCTION tips */
#define FUNCTION_tips NULL

/***** SCRIPT DEBUG *****/

/* SCRIPT DEBUG history */
#define SCRIPT_DEBUG_History NULL

/* SCRIPT DEBUG tips */
#define SCRIPT_DEBUG_tips NULL

/* SCRIPT DEBUG mode argument table */
struct redisCommandArg SCRIPT_DEBUG_mode_Subargs[] = {
{"yes",ARG_TYPE_PURE_TOKEN,-1,"YES",NULL,NULL,CMD_ARG_NONE},
{"sync",ARG_TYPE_PURE_TOKEN,-1,"SYNC",NULL,NULL,CMD_ARG_NONE},
{"no",ARG_TYPE_PURE_TOKEN,-1,"NO",NULL,NULL,CMD_ARG_NONE},
{0}
};

/* SCRIPT DEBUG argument table */
struct redisCommandArg SCRIPT_DEBUG_Args[] = {
{"mode",ARG_TYPE_ONEOF,-1,NULL,NULL,NULL,CMD_ARG_NONE,.subargs=SCRIPT_DEBUG_mode_S
{0}
};

/***** SCRIPT EXISTS *****/

/* SCRIPT EXISTS history */

```



```

#define SCRIPT_EXISTS_History NULL

/* SCRIPT EXISTS tips */
const char *SCRIPT_EXISTS_tips[] = {
    "request_policy:all_shards",
    "response_policy:agg_logical_and",
    NULL
};

/* SCRIPT EXISTS argument table */
struct redisCommandArg SCRIPT_EXISTS_Args[] = {
    {"sha1", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
    {0}
};

/***** SCRIPT FLUSH *****/

/* SCRIPT FLUSH history */
commandHistory SCRIPT_FLUSH_History[] = {
    {"6.2.0", "Added the `ASYNC` and `SYNC` flushing mode modifiers."},
    {0}
};

/* SCRIPT FLUSH tips */
const char *SCRIPT_FLUSH_tips[] = {
    "request_policy:all_nodes",
    "response_policy:all_succeeded",
    NULL
};

/* SCRIPT FLUSH async argument table */
struct redisCommandArg SCRIPT_FLUSH_async_Subargs[] = {
    {"async", ARG_TYPE_PURE_TOKEN, -1, "ASYNC", NULL, NULL, CMD_ARG_NONE},
    {"sync", ARG_TYPE_PURE_TOKEN, -1, "SYNC", NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* SCRIPT FLUSH argument table */
struct redisCommandArg SCRIPT_FLUSH_Args[] = {
    {"async", ARG_TYPE_ONEOF, -1, NULL, NULL, "6.2.0", CMD_ARG_OPTIONAL, .subargs=SCRIPT_FLUSH_async_Subargs},
    {0}
};

/***** SCRIPT HELP *****/

/* SCRIPT HELP history */
#define SCRIPT_HELP_History NULL

/* SCRIPT HELP tips */
#define SCRIPT_HELP_tips NULL

```

```

/***** SCRIPT KILL *****/

/* SCRIPT KILL history */
#define SCRIPT_KILL_History NULL

/* SCRIPT KILL tips */
const char *SCRIPT_KILL_tips[] = {
    "request_policy:all_shards",
    "response_policy:one_succeeded",
    NULL
};

/***** SCRIPT LOAD *****/

/* SCRIPT LOAD history */
#define SCRIPT_LOAD_History NULL

/* SCRIPT LOAD tips */
const char *SCRIPT_LOAD_tips[] = {
    "request_policy:all_nodes",
    "response_policy:all_succeeded",
    NULL
};

/* SCRIPT LOAD argument table */
struct redisCommandArg SCRIPT_LOAD_Args[] = {
    {"script", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* SCRIPT command table */
struct redisCommand SCRIPT_Subcommands[] = {
    {"debug", "Set the debug mode for executed scripts.", "0(1)", "3.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, SCRIPT_DEBUG},
    {"exists", "Check existence of scripts in the script cache.", "0(N) with N being the number of scripts to check (so checking a single script is an O(1) operation).", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, SCRIPT_EXISTS},
    {"flush", "Remove all the scripts from the script cache.", "0(N) with N being the number of scripts in cache", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, SCRIPT_FLUSH_History},
    {"help", "Show helpful text about the different subcommands", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, SCRIPT_HELP},
    {"kill", "Kill the script currently in execution.", "0(1)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, SCRIPT_KILL},
    {"load", "Load the specified Lua script into the script cache.", "0(N) with N being the length in bytes of the script body.", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SCRIPTING, SCRIPT_LOAD_History},
    {0}
};

/***** SCRIPT *****/

```

```

/* SCRIPT history */
#define SCRIPT_History NULL

/* SCRIPT tips */
#define SCRIPT_tips NULL

/***** SENTINEL CKQUORUM *****/

/* SENTINEL CKQUORUM history */
#define SENTINEL_CKQUORUM_History NULL

/* SENTINEL CKQUORUM tips */
#define SENTINEL_CKQUORUM_tips NULL

/* SENTINEL CKQUORUM argument table */
struct redisCommandArg SENTINEL_CKQUORUM_Args[] = {
{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL CONFIG *****/

/* SENTINEL CONFIG history */
#define SENTINEL_CONFIG_History NULL

/* SENTINEL CONFIG tips */
#define SENTINEL_CONFIG_tips NULL

/* SENTINEL CONFIG set_or_get set_param_value argument table */
struct redisCommandArg SENTINEL_CONFIG_set_or_get_set_param_value_Subargs[] = {
{"parameter", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SENTINEL CONFIG set_or_get argument table */
struct redisCommandArg SENTINEL_CONFIG_set_or_get_Subargs[] = {
{"set_param_value", ARG_TYPE_BLOCK, -1, "SET", NULL, NULL, CMD_ARG_MULTIPLE, .subargs=SENTINEL_CONFIG_set_or_get_set_param_value_Subargs},
{"parameter", ARG_TYPE_STRING, -1, "GET", NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/* SENTINEL CONFIG argument table */
struct redisCommandArg SENTINEL_CONFIG_Args[] = {
{"set_or_get", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=SENTINEL_CONFIG_set_or_get_Subargs},
{0}
};

/***** SENTINEL DEBUG *****/

```

```

/* SENTINEL DEBUG history */
#define SENTINEL_DEBUG_History NULL

/* SENTINEL DEBUG tips */
#define SENTINEL_DEBUG_tips NULL

/* SENTINEL DEBUG parameter_value argument table */
struct redisCommandArg SENTINEL_DEBUG_parameter_value_Subargs[] = {
{"parameter", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SENTINEL DEBUG argument table */
struct redisCommandArg SENTINEL_DEBUG_Args[] = {
{"parameter_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=SENT
{0}
};

/***** SENTINEL FAILOVER *****/

/* SENTINEL FAILOVER history */
#define SENTINEL_FAILOVER_History NULL

/* SENTINEL FAILOVER tips */
#define SENTINEL_FAILOVER_tips NULL

/* SENTINEL FAILOVER argument table */
struct redisCommandArg SENTINEL_FAILOVER_Args[] = {
{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL FLUSHCONFIG *****/

/* SENTINEL FLUSHCONFIG history */
#define SENTINEL_FLUSHCONFIG_History NULL

/* SENTINEL FLUSHCONFIG tips */
#define SENTINEL_FLUSHCONFIG_tips NULL

/***** SENTINEL GET_MASTER_ADDR_BY_NAME *****/

/* SENTINEL GET_MASTER_ADDR_BY_NAME history */
#define SENTINEL_GET_MASTER_ADDR_BY_NAME_History NULL

/* SENTINEL GET_MASTER_ADDR_BY_NAME tips */
#define SENTINEL_GET_MASTER_ADDR_BY_NAME_tips NULL

/* SENTINEL GET_MASTER_ADDR_BY_NAME argument table */
struct redisCommandArg SENTINEL_GET_MASTER_ADDR_BY_NAME_Args[] = {

```

```

{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL HELP *****/

/* SENTINEL HELP history */
#define SENTINEL_HELP_History NULL

/* SENTINEL HELP tips */
#define SENTINEL_HELP_tips NULL

/***** SENTINEL INFO_CACHE *****/

/* SENTINEL INFO_CACHE history */
#define SENTINEL_INFO_CACHE_History NULL

/* SENTINEL INFO_CACHE tips */
#define SENTINEL_INFO_CACHE_tips NULL

/* SENTINEL INFO_CACHE argument table */
struct redisCommandArg SENTINEL_INFO_CACHE_Args[] = {
{"nodename", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SENTINEL IS_MASTER_DOWN_BY_ADDR *****/

/* SENTINEL IS_MASTER_DOWN_BY_ADDR history */
#define SENTINEL_IS_MASTER_DOWN_BY_ADDR_History NULL

/* SENTINEL IS_MASTER_DOWN_BY_ADDR tips */
#define SENTINEL_IS_MASTER_DOWN_BY_ADDR_tips NULL

/* SENTINEL IS_MASTER_DOWN_BY_ADDR argument table */
struct redisCommandArg SENTINEL_IS_MASTER_DOWN_BY_ADDR_Args[] = {
{"ip", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"port", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"current-epoch", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"runid", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL MASTER *****/

/* SENTINEL MASTER history */
#define SENTINEL_MASTER_History NULL

/* SENTINEL MASTER tips */
#define SENTINEL_MASTER_tips NULL

```

```

/* SENTINEL MASTER argument table */
struct redisCommandArg SENTINEL_MASTER_Args[] = {
{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL MASTERS *****/

/* SENTINEL MASTERS history */
#define SENTINEL_MASTERS_History NULL

/* SENTINEL MASTERS tips */
#define SENTINEL_MASTERS_tips NULL

/***** SENTINEL MONITOR *****/

/* SENTINEL MONITOR history */
#define SENTINEL_MONITOR_History NULL

/* SENTINEL MONITOR tips */
#define SENTINEL_MONITOR_tips NULL

/* SENTINEL MONITOR argument table */
struct redisCommandArg SENTINEL_MONITOR_Args[] = {
{"name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"ip", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"port", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"quorum", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL MYID *****/

/* SENTINEL MYID history */
#define SENTINEL_MYID_History NULL

/* SENTINEL MYID tips */
#define SENTINEL_MYID_tips NULL

/***** SENTINEL PENDING_SCRIPTS *****/

/* SENTINEL PENDING_SCRIPTS history */
#define SENTINEL_PENDING_SCRIPTS_History NULL

/* SENTINEL PENDING_SCRIPTS tips */
#define SENTINEL_PENDING_SCRIPTS_tips NULL

/***** SENTINEL REMOVE *****/

/* SENTINEL REMOVE history */
#define SENTINEL_REMOVE_History NULL

```

```

/* SENTINEL REMOVE tips */
#define SENTINEL_REMOVE_tips NULL

/* SENTINEL REMOVE argument table */
struct redisCommandArg SENTINEL_REMOVE_Args[] = {
{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL REPLICAS *****/

/* SENTINEL REPLICAS history */
#define SENTINEL_REPLICAS_History NULL

/* SENTINEL REPLICAS tips */
#define SENTINEL_REPLICAS_tips NULL

/* SENTINEL REPLICAS argument table */
struct redisCommandArg SENTINEL_REPLICAS_Args[] = {
{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL RESET *****/

/* SENTINEL RESET history */
#define SENTINEL_RESET_History NULL

/* SENTINEL RESET tips */
#define SENTINEL_RESET_tips NULL

/* SENTINEL RESET argument table */
struct redisCommandArg SENTINEL_RESET_Args[] = {
{"pattern", ARG_TYPE_PATTERN, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SENTINEL SENTINELS *****/

/* SENTINEL SENTINELS history */
#define SENTINEL_SENTINELS_History NULL

/* SENTINEL SENTINELS tips */
#define SENTINEL_SENTINELS_tips NULL

/* SENTINEL SENTINELS argument table */
struct redisCommandArg SENTINEL_SENTINELS_Args[] = {
{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/***** SENTINEL SET *****/

/* SENTINEL SET history */
#define SENTINEL_SET_History NULL

/* SENTINEL SET tips */
#define SENTINEL_SET_tips NULL

/* SENTINEL SET option_value argument table */
struct redisCommandArg SENTINEL_SET_option_value_Subargs[] = {
{"option", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SENTINEL SET argument table */
struct redisCommandArg SENTINEL_SET_Args[] = {
{"master-name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"option_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=SENTINEL_SET_option_value_Subargs},
{0}
};

/***** SENTINEL SIMULATE_FAILURE *****/

/* SENTINEL SIMULATE_FAILURE history */
#define SENTINEL_SIMULATE_FAILURE_History NULL

/* SENTINEL SIMULATE_FAILURE tips */
#define SENTINEL_SIMULATE_FAILURE_tips NULL

/* SENTINEL SIMULATE_FAILURE mode argument table */
struct redisCommandArg SENTINEL_SIMULATE_FAILURE_mode_Subargs[] = {
{"crash-after-election", ARG_TYPE_PURE_TOKEN, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"crash-after-promotion", ARG_TYPE_PURE_TOKEN, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"help", ARG_TYPE_PURE_TOKEN, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SENTINEL SIMULATE_FAILURE argument table */
struct redisCommandArg SENTINEL_SIMULATE_FAILURE_Args[] = {
{"mode", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE, .subargs=SENTINEL_SIMULATE_FAILURE_mode_Subargs},
{0}
};

/* SENTINEL command table */
struct redisCommand SENTINEL_Subcommands[] = {
{"ckquorum", "Check for a Sentinel quorum", NULL, "2.8.4", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SENTINEL, SENTINEL_CKQUORUM},
{"config", "Configure Sentinel", "0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SENTINEL, SENTINEL_CONFIG},
{0}
};

```



```

{"debug","List or update the current configurable parameters","0(N) where N is
the number of configurable
parameters","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_DEBUG_H
{"failover","Force a
failover",NULL,"2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_FAIL
{"flushconfig","Rewrite configuration
file","0(1)","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_FLUSHC
{"get-master-addr-by-name","Get port and address of a
master","0(1)","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_GET_
{"help","Show helpful text about the different
subcommands","0(1)","6.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL
{"info-cache","Get cached INFO from the instances in the deployment","0(N)
where N is the number of
instances","3.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_INFO_CAC
{"is-master-down-by-addr","Check if a master is
down","0(1)","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_IS_MAS
{"master","Shows the state of a
master","0(1)","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_MAST
{"masters","List the monitored masters","0(N) where N is the number of
masters","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_MASTERS_Hi
{"monitor","Start
monitoring","0(1)","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_
{"myid","Get the Sentinel instance
ID","0(1)","6.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_MYID_His
{"pending-scripts","Get information about pending
scripts",NULL,"2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_PENDI
{"remove","Stop
monitoring","0(1)","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_
{"replicas","List the monitored replicas","0(N) where N is the number of
replicas","5.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_REPLICAS_
{"reset","Reset masters by name pattern","0(N) where N is the number of
monitored
masters","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_RESET_Hist
{"sentinels","List the Sentinel instances","0(N) where N is the number of
Sentinels","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_SENTINEL
{"set","Change the configuration of a monitored
master","0(1)","2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_SET_
{"simulate-failure","Simulate failover
scenarios",NULL,"3.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_SIM
{0}
};

```

```

/***** SENTINEL *****/

```

```

/* SENTINEL history */
#define SENTINEL_History NULL

```

```

/* SENTINEL tips */
#define SENTINEL_tips NULL

```

```

/***** ACL CAT *****/

```

```

/* ACL CAT history */
#define ACL_CAT_History NULL

/* ACL CAT tips */
#define ACL_CAT_tips NULL

/* ACL CAT argument table */
struct redisCommandArg ACL_CAT_Args[] = {
{"categoryname", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ACL DELUSER *****/

/* ACL DELUSER history */
#define ACL_DELUSER_History NULL

/* ACL DELUSER tips */
#define ACL_DELUSER_tips NULL

/* ACL DELUSER argument table */
struct redisCommandArg ACL_DELUSER_Args[] = {
{"username", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** ACL DRYRUN *****/

/* ACL DRYRUN history */
#define ACL_DRYRUN_History NULL

/* ACL DRYRUN tips */
#define ACL_DRYRUN_tips NULL

/* ACL DRYRUN argument table */
struct redisCommandArg ACL_DRYRUN_Args[] = {
{"username", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"command", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** ACL GENPASS *****/

/* ACL GENPASS history */
#define ACL_GENPASS_History NULL

/* ACL GENPASS tips */
#define ACL_GENPASS_tips NULL

```

```

/* ACL GENPASS argument table */
struct redisCommandArg ACL_GENPASS_Args[] = {
{"bits", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ACL GETUSER *****/

/* ACL GETUSER history */
commandHistory ACL_GETUSER_History[] = {
{"6.2.0", "Added Pub/Sub channel patterns."},
{"7.0.0", "Added selectors and changed the format of key and channel patterns
from a list to their rule representation."},
{0}
};

/* ACL GETUSER tips */
#define ACL_GETUSER_tips NULL

/* ACL GETUSER argument table */
struct redisCommandArg ACL_GETUSER_Args[] = {
{"username", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ACL HELP *****/

/* ACL HELP history */
#define ACL_HELP_History NULL

/* ACL HELP tips */
#define ACL_HELP_tips NULL

/***** ACL LIST *****/

/* ACL LIST history */
#define ACL_LIST_History NULL

/* ACL LIST tips */
#define ACL_LIST_tips NULL

/***** ACL LOAD *****/

/* ACL LOAD history */
#define ACL_LOAD_History NULL

/* ACL LOAD tips */
#define ACL_LOAD_tips NULL

/***** ACL LOG *****/

```

```

/* ACL LOG history */
#define ACL_LOG_History NULL

/* ACL LOG tips */
#define ACL_LOG_tips NULL

/* ACL LOG operation argument table */
struct redisCommandArg ACL_LOG_operation_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"reset", ARG_TYPE_PURE_TOKEN, -1, "RESET", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ACL LOG argument table */
struct redisCommandArg ACL_LOG_Args[] = {
{"operation", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ACL_LOG_op
{0}
};

/***** ACL SAVE *****/

/* ACL SAVE history */
#define ACL_SAVE_History NULL

/* ACL SAVE tips */
#define ACL_SAVE_tips NULL

/***** ACL SETUSER *****/

/* ACL SETUSER history */
commandHistory ACL_SETUSER_History[] = {
{"6.2.0", "Added Pub/Sub channel patterns."},
{"7.0.0", "Added selectors and key based permissions."},
{0}
};

/* ACL SETUSER tips */
#define ACL_SETUSER_tips NULL

/* ACL SETUSER argument table */
struct redisCommandArg ACL_SETUSER_Args[] = {
{"username", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"rule", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** ACL USERS *****/

/* ACL USERS history */
#define ACL_USERS_History NULL

```

```

/* ACL USERS tips */
#define ACL_USERS_tips NULL

/***** ACL WHOAMI *****/

/* ACL WHOAMI history */
#define ACL_WHOAMI_History NULL

/* ACL WHOAMI tips */
#define ACL_WHOAMI_tips NULL

/* ACL command table */
struct redisCommand ACL_Subcommands[] = {
{"cat","List the ACL categories or the commands inside a category","0(1) since
the categories and commands are a fixed
set.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_CAT_History, ACL_CAT_
{"deluser","Remove the specified ACL users and the associated rules","0(1)
amortized time considering the typical
user.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_DELUSER_History, ACL_
{"dryrun","Returns whether the user can execute the given command without
executing the
command.", "0(1).", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_DRYRUN_H
{"genpass","Generate a pseudorandom secure password to use for ACL
users", "0(1)", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_GENPASS_Hist
{"getuser","Get the rules for a specific ACL user", "0(N). Where N is the number
of password, command and pattern rules that the user
has.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_GETUSER_History, ACL_
{"help","Show helpful text about the different
subcommands", "0(1)", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_HELP_H
{"list","List the current ACL rules in ACL config file format", "0(N). Where N
is the number of configured
users.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_LIST_History, ACL_L
{"load","Reload the ACLs from the configured ACL file", "0(N). Where N is the
number of configured
users.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_LOAD_History, ACL_L
{"log","List latest events denied because of ACLs in place", "0(N) with N being
the number of entries
shown.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_LOG_History, ACL_LO
{"save","Save the current ACL rules in the configured ACL file", "0(N). Where N
is the number of configured
users.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_SAVE_History, ACL_S
{"setuser","Modify or create the rules for a specific ACL user", "0(N). Where N
is the number of rules
provided.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_SETUSER_History
{"users","List the username of all the configured ACL rules", "0(N). Where N is
the number of configured
users.", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_USERS_History, ACL_
{"whoami","Return the name of the user associated to the current
connection", "0(1)", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, ACL_WHOAMI_
{0}
};

```

```

/***** ACL *****/

/* ACL history */
#define ACL_History NULL

/* ACL tips */
#define ACL_tips NULL

/***** BGREWRITEAOF *****/

/* BGREWRITEAOF history */
#define BGREWRITEAOF_History NULL

/* BGREWRITEAOF tips */
#define BGREWRITEAOF_tips NULL

/***** BGSAVE *****/

/* BGSAVE history */
commandHistory BGSAVE_History[] = {
{"3.2.2","Added the `SCHEDULE` option."},
{0}
};

/* BGSAVE tips */
#define BGSAVE_tips NULL

/* BGSAVE argument table */
struct redisCommandArg BGSAVE_Args[] = {
{"schedule",ARG_TYPE_PURE_TOKEN,-1,"SCHEDULE",NULL,"3.2.2",CMD_ARG_OPTIONAL},
{0}
};

/***** COMMAND COUNT *****/

/* COMMAND COUNT history */
#define COMMAND_COUNT_History NULL

/* COMMAND COUNT tips */
#define COMMAND_COUNT_tips NULL

/***** COMMAND DOCS *****/

/* COMMAND DOCS history */
#define COMMAND_DOCS_History NULL

/* COMMAND DOCS tips */
const char *COMMAND_DOCS_tips[] = {
"nondeterministic_output_order",
NULL

```

```

};

/* COMMAND DOCS argument table */
struct redisCommandArg COMMAND_DOCS_Args[] = {
{"command-
name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** COMMAND GETKEYS *****/

/* COMMAND GETKEYS history */
#define COMMAND_GETKEYS_History NULL

/* COMMAND GETKEYS tips */
#define COMMAND_GETKEYS_tips NULL

/***** COMMAND GETKEYSANDFLAGS *****/

/* COMMAND GETKEYSANDFLAGS history */
#define COMMAND_GETKEYSANDFLAGS_History NULL

/* COMMAND GETKEYSANDFLAGS tips */
#define COMMAND_GETKEYSANDFLAGS_tips NULL

/***** COMMAND HELP *****/

/* COMMAND HELP history */
#define COMMAND_HELP_History NULL

/* COMMAND HELP tips */
#define COMMAND_HELP_tips NULL

/***** COMMAND INFO *****/

/* COMMAND INFO history */
commandHistory COMMAND_INFO_History[] = {
{"7.0.0", "Allowed to be called with no argument to get info on all commands."},
{0}
};

/* COMMAND INFO tips */
const char *COMMAND_INFO_tips[] = {
"nondeterministic_output_order",
NULL
};

/* COMMAND INFO argument table */
struct redisCommandArg COMMAND_INFO_Args[] = {
{"command-
name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},

```

```

{0}
};

/***** COMMAND LIST *****/

/* COMMAND LIST history */
#define COMMAND_LIST_History NULL

/* COMMAND LIST tips */
const char *COMMAND_LIST_tips[] = {
"nondeterministic_output_order",
NULL
};

/* COMMAND LIST filterby argument table */
struct redisCommandArg COMMAND_LIST_filterby_Subargs[] = {
{"module-name", ARG_TYPE_STRING, -1, "MODULE", NULL, NULL, CMD_ARG_NONE},
{"category", ARG_TYPE_STRING, -1, "ACLCAT", NULL, NULL, CMD_ARG_NONE},
{"pattern", ARG_TYPE_PATTERN, -1, "PATTERN", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* COMMAND LIST argument table */
struct redisCommandArg COMMAND_LIST_Args[] = {
{"filterby", ARG_TYPE_ONEOF, -1, "FILTERBY", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=COMMAND_LIST_filterby_Subargs},
{0}
};

/* COMMAND command table */
struct redisCommand COMMAND_Subcommands[] = {
{"count", "Get total number of Redis commands", "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_COUNT, "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_COUNT, "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_COUNT},
{"docs", "Get array of specific Redis command documentation", "0(N) where N is the number of commands to look up", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_DOCS_History, COMMAND_DOCS_History, "0(N) where N is the number of commands to look up", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_DOCS_History, COMMAND_DOCS_History, "0(N) where N is the number of commands to look up", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_DOCS_History, COMMAND_DOCS_History},
{"getkeys", "Extract keys given a full Redis command", "0(N) where N is the number of arguments to the command", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_GETKEYS_History, COMMAND_GETKEYS_History, "0(N) where N is the number of arguments to the command", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_GETKEYS_History, COMMAND_GETKEYS_History, "0(N) where N is the number of arguments to the command", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_GETKEYS_History, COMMAND_GETKEYS_History},
{"getkeysandflags", "Extract keys and access flags given a full Redis command", "0(N) where N is the number of arguments to the command", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_GETKEYSANDFLAGS_History, COMMAND_GETKEYSANDFLAGS_History, "0(N) where N is the number of arguments to the command", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_GETKEYSANDFLAGS_History, COMMAND_GETKEYSANDFLAGS_History, "0(N) where N is the number of arguments to the command", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_GETKEYSANDFLAGS_History, COMMAND_GETKEYSANDFLAGS_History},
{"help", "Show helpful text about the different subcommands", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_HELP_History, COMMAND_HELP_History, "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_HELP_History, COMMAND_HELP_History, "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_HELP_History, COMMAND_HELP_History},
{"info", "Get array of specific Redis command details, or all when no argument is given.", "0(N) where N is the number of commands to look up", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_INFO_History, COMMAND_INFO_History, "0(N) where N is the number of commands to look up", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_INFO_History, COMMAND_INFO_History, "0(N) where N is the number of commands to look up", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_INFO_History, COMMAND_INFO_History},
{"list", "Get an array of Redis command names", "0(N) where N is the total number of Redis commands", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_LIST_History, COMMAND_LIST_History, "0(N) where N is the total number of Redis commands", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_LIST_History, COMMAND_LIST_History, "0(N) where N is the total number of Redis commands", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, COMMAND_LIST_History, COMMAND_LIST_History},
{0}
};

```



```

/***** COMMAND *****/

/* COMMAND history */
#define COMMAND_History NULL

/* COMMAND tips */
const char *COMMAND_tips[] = {
"nondeterministic_output_order",
NULL
};

/***** CONFIG GET *****/

/* CONFIG GET history */
commandHistory CONFIG_GET_History[] = {
{"7.0.0","Added the ability to pass multiple pattern parameters in one call"},
{0}
};

/* CONFIG GET tips */
#define CONFIG_GET_tips NULL

/* CONFIG GET parameter argument table */
struct redisCommandArg CONFIG_GET_parameter_Subargs[] = {
{"parameter",ARG_TYPE_STRING,-1,NULL,NULL,NULL,CMD_ARG_NONE},
{0}
};

/* CONFIG GET argument table */
struct redisCommandArg CONFIG_GET_Args[] = {
{"parameter",ARG_TYPE_BLOCK,-1,NULL,NULL,NULL,CMD_ARG_MULTIPLE,.subargs=CONFIG_GET
{0}
};

/***** CONFIG HELP *****/

/* CONFIG HELP history */
#define CONFIG_HELP_History NULL

/* CONFIG HELP tips */
#define CONFIG_HELP_tips NULL

/***** CONFIG RESETSTAT *****/

/* CONFIG RESETSTAT history */
#define CONFIG_RESETSTAT_History NULL

/* CONFIG RESETSTAT tips */
#define CONFIG_RESETSTAT_tips NULL

```

```

/***** CONFIG REWRITE *****/

/* CONFIG REWRITE history */
#define CONFIG_REWRITE_History NULL

/* CONFIG REWRITE tips */
#define CONFIG_REWRITE_tips NULL

/***** CONFIG SET *****/

/* CONFIG SET history */
commandHistory CONFIG_SET_History[] = {
{"7.0.0", "Added the ability to set multiple parameters in one call."},
{0}
};

/* CONFIG SET tips */
const char *CONFIG_SET_tips[] = {
"request_policy:all_nodes",
"response_policy:all_succeeded",
NULL
};

/* CONFIG SET parameter_value argument table */
struct redisCommandArg CONFIG_SET_parameter_value_Subargs[] = {
{"parameter", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* CONFIG SET argument table */
struct redisCommandArg CONFIG_SET_Args[] = {
{"parameter_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=CONFIG_SET_parameter_value_Subargs},
{0}
};

/* CONFIG command table */
struct redisCommand CONFIG_Subcommands[] = {
{"get", "Get the values of configuration parameters", "0(N) when N is the number of configuration parameters provided", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, CONFIG_GET_History, CONFIG_GET_tips},
{"help", "Show helpful text about the different subcommands", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, CONFIG_HELP_History, CONFIG_HELP_tips},
{"resetstat", "Reset the stats returned by INFO", "0(1)", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, CONFIG_RESETSTAT_History, CONFIG_RESETSTAT_tips},
{"rewrite", "Rewrite the configuration file with the in memory configuration", "0(1)", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, CONFIG_REWRITE_History, CONFIG_REWRITE_tips},
{"set", "Set configuration parameters to the given values", "0(N) when N is the number of configuration parameters provided", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, CONFIG_SET_History, CONFIG_SET_tips},
{0}
};

```

```

};

/***** CONFIG *****/

/* CONFIG history */
#define CONFIG_History NULL

/* CONFIG tips */
#define CONFIG_tips NULL

/***** DBSIZE *****/

/* DBSIZE history */
#define DBSIZE_History NULL

/* DBSIZE tips */
const char *DBSIZE_tips[] = {
    "request_policy:all_shards",
    "response_policy:agg_sum",
    NULL
};

/***** DEBUG *****/

/* DEBUG history */
#define DEBUG_History NULL

/* DEBUG tips */
#define DEBUG_tips NULL

/***** FAILOVER *****/

/* FAILOVER history */
#define FAILOVER_History NULL

/* FAILOVER tips */
#define FAILOVER_tips NULL

/* FAILOVER target argument table */
struct redisCommandArg FAILOVER_target_Subargs[] = {
    {"host", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"port", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"force", ARG_TYPE_PURE_TOKEN, -1, "FORCE", NULL, NULL, CMD_ARG_OPTIONAL},
    {0}
};

/* FAILOVER argument table */
struct redisCommandArg FAILOVER_Args[] = {
    {"target", ARG_TYPE_BLOCK, -1, "TO", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=FAILOVER_target_Subargs},
    {"abort", ARG_TYPE_PURE_TOKEN, -1, "ABORT", NULL, NULL, CMD_ARG_OPTIONAL},
    {"milliseconds", ARG_TYPE_INTEGER, -1, "TIMEOUT", NULL, NULL, CMD_ARG_OPTIONAL},

```

```

{0}
};

/***** FLUSHALL *****/

/* FLUSHALL history */
commandHistory FLUSHALL_History[] = {
{"4.0.0", "Added the `ASYNC` flushing mode modifier."},
{"6.2.0", "Added the `SYNC` flushing mode modifier."},
{0}
};

/* FLUSHALL tips */
const char *FLUSHALL_tips[] = {
"request_policy:all_shards",
"response_policy:all_succeeded",
NULL
};

/* FLUSHALL async argument table */
struct redisCommandArg FLUSHALL_async_Subargs[] = {
{"async", ARG_TYPE_PURE_TOKEN, -1, "ASYNC", NULL, "4.0.0", CMD_ARG_NONE},
{"sync", ARG_TYPE_PURE_TOKEN, -1, "SYNC", NULL, "6.2.0", CMD_ARG_NONE},
{0}
};

/* FLUSHALL argument table */
struct redisCommandArg FLUSHALL_Args[] = {
{"async", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=FLUSHALL_async
{0}
};

/***** FLUSHDB *****/

/* FLUSHDB history */
commandHistory FLUSHDB_History[] = {
{"4.0.0", "Added the `ASYNC` flushing mode modifier."},
{"6.2.0", "Added the `SYNC` flushing mode modifier."},
{0}
};

/* FLUSHDB tips */
const char *FLUSHDB_tips[] = {
"request_policy:all_shards",
"response_policy:all_succeeded",
NULL
};

/* FLUSHDB async argument table */
struct redisCommandArg FLUSHDB_async_Subargs[] = {
{"async", ARG_TYPE_PURE_TOKEN, -1, "ASYNC", NULL, "4.0.0", CMD_ARG_NONE},

```

```

{"sync", ARG_TYPE_PURE_TOKEN, -1, "SYNC", NULL, "6.2.0", CMD_ARG_NONE},
{0}
};

/* FLUSHDB argument table */
struct redisCommandArg FLUSHDB_Args[] = {
{"async", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=FLUSHDB_async_
{0}
};

/***** INFO *****/

/* INFO history */
commandHistory INFO_History[] = {
{"7.0.0", "Added support for taking multiple section arguments."},
{0}
};

/* INFO tips */
const char *INFO_tips[] = {
"nondeterministic_output",
"request_policy:all_shards",
"response_policy:special",
NULL
};

/* INFO argument table */
struct redisCommandArg INFO_Args[] = {
{"section", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

/***** LASTSAVE *****/

/* LASTSAVE history */
#define LASTSAVE_History NULL

/* LASTSAVE tips */
const char *LASTSAVE_tips[] = {
"nondeterministic_output",
NULL
};

/***** LATENCY DOCTOR *****/

/* LATENCY DOCTOR history */
#define LATENCY_DOCTOR_History NULL

/* LATENCY DOCTOR tips */
const char *LATENCY_DOCTOR_tips[] = {
"nondeterministic_output",

```

```

"request_policy:all_nodes",
"response_policy:special",
NULL
};

/***** LATENCY GRAPH *****/

/* LATENCY GRAPH history */
#define LATENCY_GRAPH_History NULL

/* LATENCY GRAPH tips */
const char *LATENCY_GRAPH_tips[] = {
"nondeterministic_output",
"request_policy:all_nodes",
"response_policy:special",
NULL
};

/* LATENCY GRAPH argument table */
struct redisCommandArg LATENCY_GRAPH_Args[] = {
{"event", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LATENCY HELP *****/

/* LATENCY HELP history */
#define LATENCY_HELP_History NULL

/* LATENCY HELP tips */
#define LATENCY_HELP_tips NULL

/***** LATENCY HISTOGRAM *****/

/* LATENCY HISTOGRAM history */
#define LATENCY_HISTOGRAM_History NULL

/* LATENCY HISTOGRAM tips */
const char *LATENCY_HISTOGRAM_tips[] = {
"nondeterministic_output",
"request_policy:all_nodes",
"response_policy:special",
NULL
};

/* LATENCY HISTOGRAM argument table */
struct redisCommandArg LATENCY_HISTOGRAM_Args[] = {
{"command", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{0}
};

```

```

/***** LATENCY HISTORY *****/

/* LATENCY HISTORY history */
#define LATENCY_HISTORY_History NULL

/* LATENCY HISTORY tips */
const char *LATENCY_HISTORY_tips[] = {
    "nondeterministic_output",
    "request_policy:all_nodes",
    "response_policy:special",
    NULL
};

/* LATENCY HISTORY argument table */
struct redisCommandArg LATENCY_HISTORY_Args[] = {
    {"event", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/***** LATENCY LATEST *****/

/* LATENCY LATEST history */
#define LATENCY_LATEST_History NULL

/* LATENCY LATEST tips */
const char *LATENCY_LATEST_tips[] = {
    "nondeterministic_output",
    "request_policy:all_nodes",
    "response_policy:special",
    NULL
};

/***** LATENCY RESET *****/

/* LATENCY RESET history */
#define LATENCY_RESET_History NULL

/* LATENCY RESET tips */
const char *LATENCY_RESET_tips[] = {
    "request_policy:all_nodes",
    "response_policy:all_succeeded",
    NULL
};

/* LATENCY RESET argument table */
struct redisCommandArg LATENCY_RESET_Args[] = {
    {"event", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
    {0}
};

/* LATENCY command table */

```

```

struct redisCommand LATENCY_Subcommands[] = {
{"doctor","Return a human readable latency analysis
report.", "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, LATENCY_DOCTO
{"graph","Return a latency graph for the
event.", "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, LATENCY_GRAPH_
{"help","Show helpful text about the different
subcommands.", "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, LATENCY_
{"histogram","Return the cumulative distribution of latencies of a subset of
commands or all.", "0(N) where N is the number of commands with latency
information being
retrieved.", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, LATENCY_HISTOGRAM_
{"history","Return timestamp-latency samples for the
event.", "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, LATENCY_HISTOR
{"latest","Return the latest latency samples for all
events.", "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, LATENCY_LATES
{"reset","Reset latency data for one or more
events.", "0(1)", "2.8.13", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, LATENCY_RESET
{0}
};

/***** LATENCY *****/

/* LATENCY history */
#define LATENCY_History NULL

/* LATENCY tips */
#define LATENCY_tips NULL

/***** LOLWUT *****/

/* LOLWUT history */
#define LOLWUT_History NULL

/* LOLWUT tips */
#define LOLWUT_tips NULL

/* LOLWUT argument table */
struct redisCommandArg LOLWUT_Args[] = {
{"version", ARG_TYPE_INTEGER, -1, "VERSION", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** MEMORY DOCTOR *****/

/* MEMORY DOCTOR history */
#define MEMORY_DOCTOR_History NULL

/* MEMORY DOCTOR tips */
const char *MEMORY_DOCTOR_tips[] = {
"nondeterministic_output",
"request_policy:all_shards",

```



```
"response_policy:special",
NULL
};

/***** MEMORY HELP *****/

/* MEMORY HELP history */
#define MEMORY_HELP_History NULL

/* MEMORY HELP tips */
#define MEMORY_HELP_tips NULL

/***** MEMORY MALLOC_STATS *****/

/* MEMORY MALLOC_STATS history */
#define MEMORY_MALLOC_STATS_History NULL

/* MEMORY MALLOC_STATS tips */
const char *MEMORY_MALLOC_STATS_tips[] = {
"nondeterministic_output",
"request_policy:all_shards",
"response_policy:special",
NULL
};

/***** MEMORY PURGE *****/

/* MEMORY PURGE history */
#define MEMORY_PURGE_History NULL

/* MEMORY PURGE tips */
const char *MEMORY_PURGE_tips[] = {
"request_policy:all_shards",
"response_policy:all_succeeded",
NULL
};

/***** MEMORY STATS *****/

/* MEMORY STATS history */
#define MEMORY_STATS_History NULL

/* MEMORY STATS tips */
const char *MEMORY_STATS_tips[] = {
"nondeterministic_output",
"request_policy:all_shards",
"response_policy:special",
NULL
};

/***** MEMORY USAGE *****/
```

```

/* MEMORY USAGE history */
#define MEMORY_USAGE_History NULL

/* MEMORY USAGE tips */
#define MEMORY_USAGE_tips NULL

/* MEMORY USAGE argument table */
struct redisCommandArg MEMORY_USAGE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, "SAMPLES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* MEMORY command table */
struct redisCommand MEMORY_Subcommands[] = {
{"doctor", "Outputs memory problems
report", "0(1)", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MEMORY_DOCTOR_H
{"help", "Show helpful text about the different
subcommands", "0(1)", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MEMORY_HEL
{"malloc-stats", "Show allocator internal stats", "Depends on how much memory is
allocated, could be
slow", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MEMORY_MALLOC_STATS_Hist
{"purge", "Ask the allocator to release memory", "Depends on how much memory is
allocated, could be
slow", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MEMORY_PURGE_History, MEM
{"stats", "Show memory usage
details", "0(1)", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MEMORY_STATS_H
{"usage", "Estimate the memory usage of a key", "0(N) where N is the number of
samples.", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MEMORY_USAGE_History
{{NULL, CMD_KEY_RO, KSPEC_BS_INDEX, .bs.index={2}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, .args=MEMORY_USAGE_Args},
{0}
};

/***** MEMORY *****/

/* MEMORY history */
#define MEMORY_History NULL

/* MEMORY tips */
#define MEMORY_tips NULL

/***** MODULE HELP *****/

/* MODULE HELP history */
#define MODULE_HELP_History NULL

/* MODULE HELP tips */
#define MODULE_HELP_tips NULL

```

```

/***** MODULE LIST *****/

/* MODULE LIST history */
#define MODULE_LIST_History NULL

/* MODULE LIST tips */
const char *MODULE_LIST_tips[] = {
    "nondeterministic_output_order",
    NULL
};

/***** MODULE LOAD *****/

/* MODULE LOAD history */
#define MODULE_LOAD_History NULL

/* MODULE LOAD tips */
#define MODULE_LOAD_tips NULL

/* MODULE LOAD argument table */
struct redisCommandArg MODULE_LOAD_Args[] = {
    {"path", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
    {0}
};

/***** MODULE LOADEX *****/

/* MODULE LOADEX history */
#define MODULE_LOADEX_History NULL

/* MODULE LOADEX tips */
#define MODULE_LOADEX_tips NULL

/* MODULE LOADEX configs argument table */
struct redisCommandArg MODULE_LOADEX_configs_Subargs[] = {
    {"name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* MODULE LOADEX args argument table */
struct redisCommandArg MODULE_LOADEX_args_Subargs[] = {
    {"arg", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* MODULE LOADEX argument table */
struct redisCommandArg MODULE_LOADEX_Args[] = {
    {"path", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {"configs", ARG_TYPE_BLOCK, -1, "CONFIG", NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE |

```

```

{"args", ARG_TYPE_BLOCK, -1, "ARGS", NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE, .suba
{0}
};

/***** MODULE UNLOAD *****/

/* MODULE UNLOAD history */
#define MODULE_UNLOAD_History NULL

/* MODULE UNLOAD tips */
#define MODULE_UNLOAD_tips NULL

/* MODULE UNLOAD argument table */
struct redisCommandArg MODULE_UNLOAD_Args[] = {
{"name", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* MODULE command table */
struct redisCommand MODULE_Subcommands[] = {
{"help", "Show helpful text about the different
subcommands", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MODULE_HEL
{"list", "List all modules loaded by the server", "0(N) where N is the number of
loaded
modules.", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MODULE_LIST_History,
{"load", "Load a
module", "0(1)", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MODULE_LOAD_His
{"loadex", "Load a module with extended
parameters", "0(1)", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MODULE_LOAD
{"unload", "Unload a
module", "0(1)", "4.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SERVER, MODULE_UNLOAD_H
{0}
};

/***** MODULE *****/

/* MODULE history */
#define MODULE_History NULL

/* MODULE tips */
#define MODULE_tips NULL

/***** MONITOR *****/

/* MONITOR history */
#define MONITOR_History NULL

/* MONITOR tips */
#define MONITOR_tips NULL

/***** PSYNC *****/

```

```

/* PSYNC history */
#define PSYNC_History NULL

/* PSYNC tips */
#define PSYNC_tips NULL

/* PSYNC argument table */
struct redisCommandArg PSYNC_Args[] = {
{"replicationid", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** REPLCONF *****/

/* REPLCONF history */
#define REPLCONF_History NULL

/* REPLCONF tips */
#define REPLCONF_tips NULL

/***** REPLICAOF *****/

/* REPLICAOF history */
#define REPLICAOF_History NULL

/* REPLICAOF tips */
#define REPLICAOF_tips NULL

/* REPLICAOF argument table */
struct redisCommandArg REPLICAOF_Args[] = {
{"host", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"port", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** RESTORE Asking *****/

/* RESTORE Asking history */
#define RESTORE Asking_History NULL

/* RESTORE Asking tips */
#define RESTORE Asking_tips NULL

/***** ROLE *****/

/* ROLE history */
#define ROLE_History NULL

/* ROLE tips */

```

```

#define ROLE_tips NULL

/***** SAVE *****/

/* SAVE history */
#define SAVE_History NULL

/* SAVE tips */
#define SAVE_tips NULL

/***** SHUTDOWN *****/

/* SHUTDOWN history */
commandHistory SHUTDOWN_History[] = {
{"7.0.0", "Added the `NOW`, `FORCE` and `ABORT` modifiers."},
{0}
};

/* SHUTDOWN tips */
#define SHUTDOWN_tips NULL

/* SHUTDOWN nosave_save argument table */
struct redisCommandArg SHUTDOWN_nosave_save_Subargs[] = {
{"nosave", ARG_TYPE_PURE_TOKEN, -1, "NOSAVE", NULL, NULL, CMD_ARG_NONE},
{"save", ARG_TYPE_PURE_TOKEN, -1, "SAVE", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SHUTDOWN argument table */
struct redisCommandArg SHUTDOWN_Args[] = {
{"nosave_save", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=SHUTDOWN_nosave_save_Subargs},
{"now", ARG_TYPE_PURE_TOKEN, -1, "NOW", NULL, "7.0.0", CMD_ARG_OPTIONAL},
{"force", ARG_TYPE_PURE_TOKEN, -1, "FORCE", NULL, "7.0.0", CMD_ARG_OPTIONAL},
{"abort", ARG_TYPE_PURE_TOKEN, -1, "ABORT", NULL, "7.0.0", CMD_ARG_OPTIONAL},
{0}
};

/***** SLAVEOF *****/

/* SLAVEOF history */
#define SLAVEOF_History NULL

/* SLAVEOF tips */
#define SLAVEOF_tips NULL

/* SLAVEOF argument table */
struct redisCommandArg SLAVEOF_Args[] = {
{"host", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"port", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/***** SLOWLOG GET *****/

/* SLOWLOG GET history */
commandHistory SLOWLOG_GET_History[] = {
{"4.0.0", "Added client IP address, port and name to the reply."},
{0}
};

/* SLOWLOG GET tips */
const char *SLOWLOG_GET_tips[] = {
"request_policy:all_nodes",
"nondeterministic_output",
NULL
};

/* SLOWLOG GET argument table */
struct redisCommandArg SLOWLOG_GET_Args[] = {
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** SLOWLOG HELP *****/

/* SLOWLOG HELP history */
#define SLOWLOG_HELP_History NULL

/* SLOWLOG HELP tips */
#define SLOWLOG_HELP_tips NULL

/***** SLOWLOG LEN *****/

/* SLOWLOG LEN history */
#define SLOWLOG_LEN_History NULL

/* SLOWLOG LEN tips */
const char *SLOWLOG_LEN_tips[] = {
"request_policy:all_nodes",
"response_policy:agg_sum",
"nondeterministic_output",
NULL
};

/***** SLOWLOG RESET *****/

/* SLOWLOG RESET history */
#define SLOWLOG_RESET_History NULL

/* SLOWLOG RESET tips */
const char *SLOWLOG_RESET_tips[] = {
"request_policy:all_nodes",

```

```

"response_policy:all_succeeded",
NULL
};

/* SLOWLOG command table */
struct redisCommand SLOWLOG_Subcommands[] = {
{"get","Get the slow log's entries","0(N) where N is the number of entries
returned","2.2.12",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SLOWLOG_GET_History
{"help","Show helpful text about the different
subcommands","0(1)","6.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SLOWLOG_HE
{"len","Get the slow log's
length","0(1)","2.2.12",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SLOWLOG_LEN_Hi
{"reset","Clear all entries from the slow log","0(N) where N is the number of
entries in the
slowlog","2.2.12",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SLOWLOG_RESET_Histor
{0}
};

/***** SLOWLOG *****/

/* SLOWLOG history */
#define SLOWLOG_History NULL

/* SLOWLOG tips */
#define SLOWLOG_tips NULL

/***** SWAPDB *****/

/* SWAPDB history */
#define SWAPDB_History NULL

/* SWAPDB tips */
#define SWAPDB_tips NULL

/* SWAPDB argument table */
struct redisCommandArg SWAPDB_Args[] = {
{"index1",ARG_TYPE_INTEGER,-1,NULL,NULL,NULL,CMD_ARG_NONE},
{"index2",ARG_TYPE_INTEGER,-1,NULL,NULL,NULL,CMD_ARG_NONE},
{0}
};

/***** SYNC *****/

/* SYNC history */
#define SYNC_History NULL

/* SYNC tips */
#define SYNC_tips NULL

/***** TIME *****/

```



```

/* TIME history */
#define TIME_History NULL

/* TIME tips */
const char *TIME_tips[] = {
"nondeterministic_output",
NULL
};

/***** SADD *****/

/* SADD history */
commandHistory SADD_History[] = {
{"2.4.0","Accepts multiple `member` arguments."},
{0}
};

/* SADD tips */
#define SADD_tips NULL

/* SADD argument table */
struct redisCommandArg SADD_Args[] = {
{"key",ARG_TYPE_KEY,0,NULL,NULL,NULL,CMD_ARG_NONE},
{"member",ARG_TYPE_STRING,-1,NULL,NULL,NULL,CMD_ARG_MULTIPLE},
{0}
};

/***** SCARD *****/

/* SCARD history */
#define SCARD_History NULL

/* SCARD tips */
#define SCARD_tips NULL

/* SCARD argument table */
struct redisCommandArg SCARD_Args[] = {
{"key",ARG_TYPE_KEY,0,NULL,NULL,NULL,CMD_ARG_NONE},
{0}
};

/***** SDIFF *****/

/* SDIFF history */
#define SDIFF_History NULL

/* SDIFF tips */
const char *SDIFF_tips[] = {
"nondeterministic_output_order",
NULL
};

```

```

/* SDIFF argument table */
struct redisCommandArg SDIFF_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SDIFFSTORE *****/

/* SDIFFSTORE history */
#define SDIFFSTORE_History NULL

/* SDIFFSTORE tips */
#define SDIFFSTORE_tips NULL

/* SDIFFSTORE argument table */
struct redisCommandArg SDIFFSTORE_Args[] = {
{"destination", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SINTER *****/

/* SINTER history */
#define SINTER_History NULL

/* SINTER tips */
const char *SINTER_tips[] = {
"nondeterministic_output_order",
NULL
};

/* SINTER argument table */
struct redisCommandArg SINTER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SINTERCARD *****/

/* SINTERCARD history */
#define SINTERCARD_History NULL

/* SINTERCARD tips */
#define SINTERCARD_tips NULL

/* SINTERCARD argument table */
struct redisCommandArg SINTERCARD_Args[] = {
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},

```

```

{"limit", ARG_TYPE_INTEGER, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** SINTERSTORE *****/

/* SINTERSTORE history */
#define SINTERSTORE_History NULL

/* SINTERSTORE tips */
#define SINTERSTORE_tips NULL

/* SINTERSTORE argument table */
struct redisCommandArg SINTERSTORE_Args[] = {
{"destination", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SISMEMBER *****/

/* SISMEMBER history */
#define SISMEMBER_History NULL

/* SISMEMBER tips */
#define SISMEMBER_tips NULL

/* SISMEMBER argument table */
struct redisCommandArg SISMEMBER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SMEMBERS *****/

/* SMEMBERS history */
#define SMEMBERS_History NULL

/* SMEMBERS tips */
const char *SMEMBERS_tips[] = {
"nondeterministic_output_order",
NULL
};

/* SMEMBERS argument table */
struct redisCommandArg SMEMBERS_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/***** SMISMEMBER *****/

/* SMISMEMBER history */
#define SMISMEMBER_History NULL

/* SMISMEMBER tips */
#define SMISMEMBER_tips NULL

/* SMISMEMBER argument table */
struct redisCommandArg SMISMEMBER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SMOVE *****/

/* SMOVE history */
#define SMOVE_History NULL

/* SMOVE tips */
#define SMOVE_tips NULL

/* SMOVE argument table */
struct redisCommandArg SMOVE_Args[] = {
{"source", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"destination", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SPOP *****/

/* SPOP history */
commandHistory SPOP_History[] = {
{"3.2.0", "Added the `count` argument."},
{0}
};

/* SPOP tips */
const char *SPOP_tips[] = {
"nondeterministic_output",
NULL
};

/* SPOP argument table */
struct redisCommandArg SPOP_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, "3.2.0", CMD_ARG_OPTIONAL},
{0}
};

```

```

/***** SRANDMEMBER *****/

/* SRANDMEMBER history */
commandHistory SRANDMEMBER_History[] = {
{"2.6.0", "Added the optional `count` argument."},
{0}
};

/* SRANDMEMBER tips */
const char *SRANDMEMBER_tips[] = {
"nondeterministic_output",
NULL
};

/* SRANDMEMBER argument table */
struct redisCommandArg SRANDMEMBER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, "2.6.0", CMD_ARG_OPTIONAL},
{0}
};

/***** SREM *****/

/* SREM history */
commandHistory SREM_History[] = {
{"2.4.0", "Accepts multiple `member` arguments."},
{0}
};

/* SREM tips */
#define SREM_tips NULL

/* SREM argument table */
struct redisCommandArg SREM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SSCAN *****/

/* SSCAN history */
#define SSCAN_History NULL

/* SSCAN tips */
const char *SSCAN_tips[] = {
"nondeterministic_output",
NULL
};

```

```

/* SSCAN argument table */
struct redisCommandArg SSCAN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"cursor", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"pattern", ARG_TYPE_PATTERN, -1, "MATCH", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** SUNION *****/

/* SUNION history */
#define SUNION_History NULL

/* SUNION tips */
const char *SUNION_tips[] = {
"nondeterministic_output_order",
NULL
};

/* SUNION argument table */
struct redisCommandArg SUNION_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** SUNIONSTORE *****/

/* SUNIONSTORE history */
#define SUNIONSTORE_History NULL

/* SUNIONSTORE tips */
#define SUNIONSTORE_tips NULL

/* SUNIONSTORE argument table */
struct redisCommandArg SUNIONSTORE_Args[] = {
{"destination", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** BZMPOP *****/

/* BZMPOP history */
#define BZMPOP_History NULL

/* BZMPOP tips */
#define BZMPOP_tips NULL

/* BZMPOP where argument table */
struct redisCommandArg BZMPOP_where_Subargs[] = {

```

```

{"min", ARG_TYPE_PURE_TOKEN, -1, "MIN", NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_PURE_TOKEN, -1, "MAX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* BZMPOP argument table */
struct redisCommandArg BZMPOP_Args[] = {
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"where", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=BZMPOP_where_Subar
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** BZPOPMAX *****/

/* BZPOPMAX history */
commandHistory BZPOPMAX_History[] = {
{"6.0.0", "`timeout` is interpreted as a double instead of an integer."},
{0}
};

/* BZPOPMAX tips */
#define BZPOPMAX_tips NULL

/* BZPOPMAX argument table */
struct redisCommandArg BZPOPMAX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** BZPOPMIN *****/

/* BZPOPMIN history */
commandHistory BZPOPMIN_History[] = {
{"6.0.0", "`timeout` is interpreted as a double instead of an integer."},
{0}
};

/* BZPOPMIN tips */
#define BZPOPMIN_tips NULL

/* BZPOPMIN argument table */
struct redisCommandArg BZPOPMIN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"timeout", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/***** ZADD *****/

/* ZADD history */
commandHistory ZADD_History[] = {
{"2.4.0", "Accepts multiple elements."},
{"3.0.2", "Added the `XX`, `NX`, `CH` and `INCR` options."},
{"6.2.0", "Added the `GT` and `LT` options."},
{0}
};

/* ZADD tips */
#define ZADD_tips NULL

/* ZADD condition argument table */
struct redisCommandArg ZADD_condition_Subargs[] = {
{"nx", ARG_TYPE_PURE_TOKEN, -1, "NX", NULL, NULL, CMD_ARG_NONE},
{"xx", ARG_TYPE_PURE_TOKEN, -1, "XX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZADD comparison argument table */
struct redisCommandArg ZADD_comparison_Subargs[] = {
{"gt", ARG_TYPE_PURE_TOKEN, -1, "GT", NULL, NULL, CMD_ARG_NONE},
{"lt", ARG_TYPE_PURE_TOKEN, -1, "LT", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZADD score_member argument table */
struct redisCommandArg ZADD_score_member_Subargs[] = {
{"score", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZADD argument table */
struct redisCommandArg ZADD_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"condition", ARG_TYPE_ONEOF, -1, NULL, NULL, "3.0.2", CMD_ARG_OPTIONAL, .subargs=ZADD_co},
{"comparison", ARG_TYPE_ONEOF, -1, NULL, NULL, "6.2.0", CMD_ARG_OPTIONAL, .subargs=ZADD_c},
{"change", ARG_TYPE_PURE_TOKEN, -1, "CH", NULL, "3.0.2", CMD_ARG_OPTIONAL},
{"increment", ARG_TYPE_PURE_TOKEN, -1, "INCR", NULL, "3.0.2", CMD_ARG_OPTIONAL},
{"score_member", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=ZADD_sc},
{0}
};

/***** ZCARD *****/

/* ZCARD history */
#define ZCARD_History NULL

/* ZCARD tips */

```



```

#define ZCARD_tips NULL

/* ZCARD argument table */
struct redisCommandArg ZCARD_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZCOUNT *****/

/* ZCOUNT history */
#define ZCOUNT_History NULL

/* ZCOUNT tips */
#define ZCOUNT_tips NULL

/* ZCOUNT argument table */
struct redisCommandArg ZCOUNT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZDIFF *****/

/* ZDIFF history */
#define ZDIFF_History NULL

/* ZDIFF tips */
#define ZDIFF_tips NULL

/* ZDIFF argument table */
struct redisCommandArg ZDIFF_Args[] = {
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZDIFFSTORE *****/

/* ZDIFFSTORE history */
#define ZDIFFSTORE_History NULL

/* ZDIFFSTORE tips */
#define ZDIFFSTORE_tips NULL

/* ZDIFFSTORE argument table */
struct redisCommandArg ZDIFFSTORE_Args[] = {
{"destination", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** ZINCRBY *****/

/* ZINCRBY history */
#define ZINCRBY_History NULL

/* ZINCRBY tips */
#define ZINCRBY_tips NULL

/* ZINCRBY argument table */
struct redisCommandArg ZINCRBY_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"increment", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZINTER *****/

/* ZINTER history */
#define ZINTER_History NULL

/* ZINTER tips */
#define ZINTER_tips NULL

/* ZINTER aggregate argument table */
struct redisCommandArg ZINTER_aggregate_Subargs[] = {
{"sum", ARG_TYPE_PURE_TOKEN, -1, "SUM", NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_PURE_TOKEN, -1, "MIN", NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_PURE_TOKEN, -1, "MAX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZINTER argument table */
struct redisCommandArg ZINTER_Args[] = {
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"weight", ARG_TYPE_INTEGER, -1, "WEIGHTS", NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{"aggregate", ARG_TYPE_ONEOF, -1, "AGGREGATE", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZIN},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZINTERCARD *****/

/* ZINTERCARD history */
#define ZINTERCARD_History NULL

```

```

/* ZINTERCARD tips */
#define ZINTERCARD_tips NULL

/* ZINTERCARD argument table */
struct redisCommandArg ZINTERCARD_Args[] = {
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"limit", ARG_TYPE_INTEGER, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZINTERSTORE *****/

/* ZINTERSTORE history */
#define ZINTERSTORE_History NULL

/* ZINTERSTORE tips */
#define ZINTERSTORE_tips NULL

/* ZINTERSTORE aggregate argument table */
struct redisCommandArg ZINTERSTORE_aggregate_Subargs[] = {
{"sum", ARG_TYPE_PURE_TOKEN, -1, "SUM", NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_PURE_TOKEN, -1, "MIN", NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_PURE_TOKEN, -1, "MAX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZINTERSTORE argument table */
struct redisCommandArg ZINTERSTORE_Args[] = {
{"destination", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"weight", ARG_TYPE_INTEGER, -1, "WEIGHTS", NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{"aggregate", ARG_TYPE_ONEOF, -1, "AGGREGATE", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZINTERSTORE_aggregate_Subargs},
{0}
};

/***** ZLEXCOUNT *****/

/* ZLEXCOUNT history */
#define ZLEXCOUNT_History NULL

/* ZLEXCOUNT tips */
#define ZLEXCOUNT_tips NULL

/* ZLEXCOUNT argument table */
struct redisCommandArg ZLEXCOUNT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{0}
};

/***** ZMPOP *****/

/* ZMPOP history */
#define ZMPOP_History NULL

/* ZMPOP tips */
#define ZMPOP_tips NULL

/* ZMPOP where argument table */
struct redisCommandArg ZMPOP_where_Subargs[] = {
{"min", ARG_TYPE_PURE_TOKEN, -1, "MIN", NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_PURE_TOKEN, -1, "MAX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZMPOP argument table */
struct redisCommandArg ZMPOP_Args[] = {
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"where", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=ZMPOP_where_Subargs},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZMSCORE *****/

/* ZMSCORE history */
#define ZMSCORE_History NULL

/* ZMSCORE tips */
#define ZMSCORE_tips NULL

/* ZMSCORE argument table */
struct redisCommandArg ZMSCORE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** ZPOPMAX *****/

/* ZPOPMAX history */
#define ZPOPMAX_History NULL

/* ZPOPMAX tips */
#define ZPOPMAX_tips NULL

/* ZPOPMAX argument table */

```

```

struct redisCommandArg ZPOPMAX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZPOPMIN *****/

/* ZPOPMIN history */
#define ZPOPMIN_History NULL

/* ZPOPMIN tips */
#define ZPOPMIN_tips NULL

/* ZPOPMIN argument table */
struct redisCommandArg ZPOPMIN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZRANDMEMBER *****/

/* ZRANDMEMBER history */
#define ZRANDMEMBER_History NULL

/* ZRANDMEMBER tips */
const char *ZRANDMEMBER_tips[] = {
"nondeterministic_output",
NULL
};

/* ZRANDMEMBER options argument table */
struct redisCommandArg ZRANDMEMBER_options_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* ZRANDMEMBER argument table */
struct redisCommandArg ZRANDMEMBER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"options", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZRANDMEMBER_options_Subargs},
{0}
};

/***** ZRANGE *****/

/* ZRANGE history */
commandHistory ZRANGE_History[] = {
{"6.2.0", "Added the `REV`, `BYSCORE`, `BYLEX` and `LIMIT` options."},

```

```

{0}
};

/* ZRANGE tips */
#define ZRANGE_tips NULL

/* ZRANGE sortby argument table */
struct redisCommandArg ZRANGE_sortby_Subargs[] = {
{"byscore", ARG_TYPE_PURE_TOKEN, -1, "BYScore", NULL, NULL, CMD_ARG_NONE},
{"bylex", ARG_TYPE_PURE_TOKEN, -1, "BYLEX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZRANGE offset_count argument table */
struct redisCommandArg ZRANGE_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZRANGE argument table */
struct redisCommandArg ZRANGE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"sortby", ARG_TYPE_ONEOF, -1, NULL, NULL, "6.2.0", CMD_ARG_OPTIONAL, .subargs=ZRANGE_sortby_Subargs},
{"rev", ARG_TYPE_PURE_TOKEN, -1, "REV", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, "6.2.0", CMD_ARG_OPTIONAL, .subargs=ZRANGE_offset_count_Subargs},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZRANGEBYLEX *****/

/* ZRANGEBYLEX history */
#define ZRANGEBYLEX_History NULL

/* ZRANGEBYLEX tips */
#define ZRANGEBYLEX_tips NULL

/* ZRANGEBYLEX offset_count argument table */
struct redisCommandArg ZRANGEBYLEX_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZRANGEBYLEX argument table */
struct redisCommandArg ZRANGEBYLEX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{"max", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZRAN
{0}
};

/***** ZRANGEBYSCORE *****/

/* ZRANGEBYSCORE history */
commandHistory ZRANGEBYSCORE_History[] = {
{"2.0.0", "Added the `WITHSCORES` modifier."},
{0}
};

/* ZRANGEBYSCORE tips */
#define ZRANGEBYSCORE_tips NULL

/* ZRANGEBYSCORE offset_count argument table */
struct redisCommandArg ZRANGEBYSCORE_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZRANGEBYSCORE argument table */
struct redisCommandArg ZRANGEBYSCORE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, "2.0.0", CMD_ARG_OPTIONAL},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZRAN
{0}
};

/***** ZRANGESTORE *****/

/* ZRANGESTORE history */
#define ZRANGESTORE_History NULL

/* ZRANGESTORE tips */
#define ZRANGESTORE_tips NULL

/* ZRANGESTORE sortby argument table */
struct redisCommandArg ZRANGESTORE_sortby_Subargs[] = {
{"byscore", ARG_TYPE_PURE_TOKEN, -1, "BYSORE", NULL, NULL, CMD_ARG_NONE},
{"bylex", ARG_TYPE_PURE_TOKEN, -1, "BYLEX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZRANGESTORE offset_count argument table */
struct redisCommandArg ZRANGESTORE_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZRANGESTORE argument table */
struct redisCommandArg ZRANGESTORE_Args[] = {
{"dst", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"src", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"sortby", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZRANGESTORE_s
{"rev", ARG_TYPE_PURE_TOKEN, -1, "REV", NULL, NULL, CMD_ARG_OPTIONAL},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZRAN
{0}
};

/***** ZRANK *****/

/* ZRANK history */
#define ZRANK_History NULL

/* ZRANK tips */
#define ZRANK_tips NULL

/* ZRANK argument table */
struct redisCommandArg ZRANK_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZREM *****/

/* ZREM history */
commandHistory ZREM_History[] = {
{"2.4.0", "Accepts multiple elements."},
{0}
};

/* ZREM tips */
#define ZREM_tips NULL

/* ZREM argument table */
struct redisCommandArg ZREM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** ZREMRANGEBYLEX *****/

```



```

/* ZREMRANGEBYLEX history */
#define ZREMRANGEBYLEX_History NULL

/* ZREMRANGEBYLEX tips */
#define ZREMRANGEBYLEX_tips NULL

/* ZREMRANGEBYLEX argument table */
struct redisCommandArg ZREMRANGEBYLEX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZREMRANGEBYRANK *****/

/* ZREMRANGEBYRANK history */
#define ZREMRANGEBYRANK_History NULL

/* ZREMRANGEBYRANK tips */
#define ZREMRANGEBYRANK_tips NULL

/* ZREMRANGEBYRANK argument table */
struct redisCommandArg ZREMRANGEBYRANK_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"stop", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZREMRANGEBYSCORE *****/

/* ZREMRANGEBYSCORE history */
#define ZREMRANGEBYSCORE_History NULL

/* ZREMRANGEBYSCORE tips */
#define ZREMRANGEBYSCORE_tips NULL

/* ZREMRANGEBYSCORE argument table */
struct redisCommandArg ZREMRANGEBYSCORE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZREVRANGE *****/

/* ZREVRANGE history */
#define ZREVRANGE_History NULL

```

```

/* ZREVRANGE tips */
#define ZREVRANGE_tips NULL

/* ZREVRANGE argument table */
struct redisCommandArg ZREVRANGE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"stop", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZREVRANGEBYLEX *****/

/* ZREVRANGEBYLEX history */
#define ZREVRANGEBYLEX_History NULL

/* ZREVRANGEBYLEX tips */
#define ZREVRANGEBYLEX_tips NULL

/* ZREVRANGEBYLEX offset_count argument table */
struct redisCommandArg ZREVRANGEBYLEX_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZREVRANGEBYLEX argument table */
struct redisCommandArg ZREVRANGEBYLEX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZREV
{0}
};

/***** ZREVRANGEBYSCORE *****/

/* ZREVRANGEBYSCORE history */
commandHistory ZREVRANGEBYSCORE_History[] = {
{"2.1.6", "`min` and `max` can be exclusive."},
{0}
};

/* ZREVRANGEBYSCORE tips */
#define ZREVRANGEBYSCORE_tips NULL

/* ZREVRANGEBYSCORE offset_count argument table */
struct redisCommandArg ZREVRANGEBYSCORE_offset_count_Subargs[] = {
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},

```

```

{0}
};

/* ZREVRANGEBYSCORE argument table */
struct redisCommandArg ZREVRANGEBYSCORE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, NULL, CMD_ARG_OPTIONAL},
{"offset_count", ARG_TYPE_BLOCK, -1, "LIMIT", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZREV
{0}
};

/***** ZREVRANK *****/

/* ZREVRANK history */
#define ZREVRANK_History NULL

/* ZREVRANK tips */
#define ZREVRANK_tips NULL

/* ZREVRANK argument table */
struct redisCommandArg ZREVRANK_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZSCAN *****/

/* ZSCAN history */
#define ZSCAN_History NULL

/* ZSCAN tips */
const char *ZSCAN_tips[] = {
"nondeterministic_output",
NULL
};

/* ZSCAN argument table */
struct redisCommandArg ZSCAN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"cursor", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"pattern", ARG_TYPE_PATTERN, -1, "MATCH", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZSCORE *****/

/* ZSCORE history */

```

```

#define ZSCORE_History NULL

/* ZSCORE tips */
#define ZSCORE_tips NULL

/* ZSCORE argument table */
struct redisCommandArg ZSCORE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"member", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** ZUNION *****/

/* ZUNION history */
#define ZUNION_History NULL

/* ZUNION tips */
#define ZUNION_tips NULL

/* ZUNION aggregate argument table */
struct redisCommandArg ZUNION_aggregate_Subargs[] = {
{"sum", ARG_TYPE_PURE_TOKEN, -1, "SUM", NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_PURE_TOKEN, -1, "MIN", NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_PURE_TOKEN, -1, "MAX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* ZUNION argument table */
struct redisCommandArg ZUNION_Args[] = {
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"weight", ARG_TYPE_INTEGER, -1, "WEIGHTS", NULL, NULL, CMD_ARG_OPTIONAL|CMD_ARG_MULTIPLE},
{"aggregate", ARG_TYPE_ONEOF, -1, "AGGREGATE", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZUNION_aggregate_Subargs},
{"withscores", ARG_TYPE_PURE_TOKEN, -1, "WITHSCORES", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** ZUNIONSTORE *****/

/* ZUNIONSTORE history */
#define ZUNIONSTORE_History NULL

/* ZUNIONSTORE tips */
#define ZUNIONSTORE_tips NULL

/* ZUNIONSTORE aggregate argument table */
struct redisCommandArg ZUNIONSTORE_aggregate_Subargs[] = {
{"sum", ARG_TYPE_PURE_TOKEN, -1, "SUM", NULL, NULL, CMD_ARG_NONE},
{"min", ARG_TYPE_PURE_TOKEN, -1, "MIN", NULL, NULL, CMD_ARG_NONE},
{"max", ARG_TYPE_PURE_TOKEN, -1, "MAX", NULL, NULL, CMD_ARG_NONE},

```

```

{0}
};

/* ZUNIONSTORE argument table */
struct redisCommandArg ZUNIONSTORE_Args[] = {
{"destination", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"numkeys", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"key", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"weight", ARG_TYPE_INTEGER, -1, "WEIGHTS", NULL, NULL, CMD_ARG_OPTIONAL | CMD_ARG_MULTIPLE},
{"aggregate", ARG_TYPE_ONEOF, -1, "AGGREGATE", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=ZUNIONSTORE_ARGS},
{0}
};

/***** XACK *****/

/* XACK history */
#define XACK_History NULL

/* XACK tips */
#define XACK_tips NULL

/* XACK argument table */
struct redisCommandArg XACK_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"group", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** XADD *****/

/* XADD history */
commandHistory XADD_History[] = {
{"6.2.0", "Added the `NOMKSTREAM` option, `MINID` trimming strategy and the `LIMIT` option."},
{"7.0.0", "Added support for the `<ms>-*` explicit ID form."},
{0}
};

/* XADD tips */
const char *XADD_tips[] = {
"nondeterministic_output",
NULL
};

/* XADD trim strategy argument table */
struct redisCommandArg XADD_trim_strategy_Subargs[] = {
{"maxlen", ARG_TYPE_PURE_TOKEN, -1, "MAXLEN", NULL, NULL, CMD_ARG_NONE},
{"minid", ARG_TYPE_PURE_TOKEN, -1, "MINID", NULL, "6.2.0", CMD_ARG_NONE},
{0}
};

```

```

/* XADD trim operator argument table */
struct redisCommandArg XADD_trim_operator_Subargs[] = {
{"equal", ARG_TYPE_PURE_TOKEN, -1, "=", NULL, NULL, CMD_ARG_NONE},
{"approximately", ARG_TYPE_PURE_TOKEN, -1, "~", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* XADD trim argument table */
struct redisCommandArg XADD_trim_Subargs[] = {
{"strategy", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=XADD_trim_strat
{"operator", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=XADD_trim_o
{"threshold", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, "LIMIT", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{0}
};

/* XADD id_or_auto argument table */
struct redisCommandArg XADD_id_or_auto_Subargs[] = {
{"auto_id", ARG_TYPE_PURE_TOKEN, -1, "*", NULL, NULL, CMD_ARG_NONE},
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* XADD field_value argument table */
struct redisCommandArg XADD_field_value_Subargs[] = {
{"field", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* XADD argument table */
struct redisCommandArg XADD_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"nomkstream", ARG_TYPE_PURE_TOKEN, -1, "NOMKSTREAM", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{"trim", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=XADD_trim_Subar
{"id_or_auto", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=XADD_id_or_au
{"field_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=XADD_fie
{0}
};

/***** XAUTOCLAIM *****/

/* XAUTOCLAIM history */
commandHistory XAUTOCLAIM_History[] = {
{"7.0.0", "Added an element to the reply array, containing deleted entries the
command cleared from the PEL"},
{0}
};

/* XAUTOCLAIM tips */

```

```

const char *XAUTOCLAIM_tips[] = {
"nondeterministic_output",
NULL
};

/* XAUTOCLAIM argument table */
struct redisCommandArg XAUTOCLAIM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"group", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"consumer", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"min-idle-time", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{"justid", ARG_TYPE_PURE_TOKEN, -1, "JUSTID", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** XCLAIM *****/

/* XCLAIM history */
#define XCLAIM_History NULL

/* XCLAIM tips */
const char *XCLAIM_tips[] = {
"nondeterministic_output",
NULL
};

/* XCLAIM argument table */
struct redisCommandArg XCLAIM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"group", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"consumer", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"min-idle-time", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"ms", ARG_TYPE_INTEGER, -1, "IDLE", NULL, NULL, CMD_ARG_OPTIONAL},
{"unix-time-
milliseconds", ARG_TYPE_UNIX_TIME, -1, "TIME", NULL, NULL, CMD_ARG_OPTIONAL},
{"count", ARG_TYPE_INTEGER, -1, "RETRYCOUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{"force", ARG_TYPE_PURE_TOKEN, -1, "FORCE", NULL, NULL, CMD_ARG_OPTIONAL},
{"justid", ARG_TYPE_PURE_TOKEN, -1, "JUSTID", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** XDEL *****/

/* XDEL history */
#define XDEL_History NULL

/* XDEL tips */
#define XDEL_tips NULL

```

```

/* XDEL argument table */
struct redisCommandArg XDEL_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** XGROUP CREATE *****/

/* XGROUP CREATE history */
commandHistory XGROUP_CREATE_History[] = {
{"7.0.0", "Added the `entries_read` named argument."},
{0}
};

/* XGROUP CREATE tips */
#define XGROUP_CREATE_tips NULL

/* XGROUP CREATE id argument table */
struct redisCommandArg XGROUP_CREATE_id_Subargs[] = {
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"new_id", ARG_TYPE_PURE_TOKEN, -1, "{{content}}quot", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* XGROUP CREATE argument table */
struct redisCommandArg XGROUP_CREATE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"groupname", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"id", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=XGROUP_CREATE_id_Subargs},
{"mkstream", ARG_TYPE_PURE_TOKEN, -1, "MKSTREAM", NULL, NULL, CMD_ARG_OPTIONAL},
{"entries_read", ARG_TYPE_INTEGER, -1, "ENTRIESREAD", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** XGROUP CREATECONSUMER *****/

/* XGROUP CREATECONSUMER history */
#define XGROUP_CREATECONSUMER_History NULL

/* XGROUP CREATECONSUMER tips */
#define XGROUP_CREATECONSUMER_tips NULL

/* XGROUP CREATECONSUMER argument table */
struct redisCommandArg XGROUP_CREATECONSUMER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"groupname", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"consumername", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```



```

/***** XGROUP DELCONSUMER *****/

/* XGROUP DELCONSUMER history */
#define XGROUP_DELCONSUMER_History NULL

/* XGROUP DELCONSUMER tips */
#define XGROUP_DELCONSUMER_tips NULL

/* XGROUP DELCONSUMER argument table */
struct redisCommandArg XGROUP_DELCONSUMER_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"groupname", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"consumername", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** XGROUP DESTROY *****/

/* XGROUP DESTROY history */
#define XGROUP_DESTROY_History NULL

/* XGROUP DESTROY tips */
#define XGROUP_DESTROY_tips NULL

/* XGROUP DESTROY argument table */
struct redisCommandArg XGROUP_DESTROY_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"groupname", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** XGROUP HELP *****/

/* XGROUP HELP history */
#define XGROUP_HELP_History NULL

/* XGROUP HELP tips */
#define XGROUP_HELP_tips NULL

/***** XGROUP SETID *****/

/* XGROUP SETID history */
commandHistory XGROUP_SETID_History[] = {
{"7.0.0", "Added the optional `entries_read` argument."},
{0}
};

/* XGROUP SETID tips */
#define XGROUP_SETID_tips NULL

```

```

/* XGROUP SETID id argument table */
struct redisCommandArg XGROUP_SETID_id_Subargs[] = {
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"new_id", ARG_TYPE_PURE_TOKEN, -1, "{{content}}" , NULL, NULL, CMD_ARG_NONE},
{0}
};

/* XGROUP SETID argument table */
struct redisCommandArg XGROUP_SETID_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"groupname", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"id", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=XGROUP_SETID_id_Subar
{"entries_read", ARG_TYPE_INTEGER, -1, "ENTRIESREAD", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* XGROUP command table */
struct redisCommand XGROUP_Subcommands[] = {
{"create", "Create a consumer
group.", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XGROUP_CREATE_H
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XGROUP_CREATE_Args},
{"createconsumer", "Create a consumer in a consumer
group.", "0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XGROUP_CREATECO
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XGROUP_CREATECONSUMER_Args},
{"delconsumer", "Delete a consumer from a consumer
group.", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XGROUP_DELCONSU
{{NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XGROUP_DELCONSUMER_Args},
{"destroy", "Destroy a consumer group.", "0(N) where N is the number of entries
in the group's pending entries list
(PEL).", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XGROUP_DESTROY_History
{{NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XGROUP_DESTROY_Args},
{"help", "Show helpful text about the different
subcommands", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XGROUP_HEL
{"setid", "Set a consumer group to an arbitrary last delivered ID
value.", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XGROUP_SETID_Hi
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XGROUP_SETID_Args},
{0}
};

/* ***** XGROUP ***** */

/* XGROUP history */
#define XGROUP_History NULL

/* XGROUP tips */
#define XGROUP_tips NULL

```

```

/***** XINFO CONSUMERS *****/

/* XINFO CONSUMERS history */
#define XINFO_CONSUMERS_History NULL

/* XINFO CONSUMERS tips */
const char *XINFO_CONSUMERS_tips[] = {
    "nondeterministic_output",
    NULL
};

/* XINFO CONSUMERS argument table */
struct redisCommandArg XINFO_CONSUMERS_Args[] = {
    {"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
    {"groupname", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/***** XINFO GROUPS *****/

/* XINFO GROUPS history */
commandHistory XINFO_GROUPS_History[] = {
    {"7.0.0", "Added the `entries-read` and `lag` fields"},
    {0}
};

/* XINFO GROUPS tips */
#define XINFO_GROUPS_tips NULL

/* XINFO GROUPS argument table */
struct redisCommandArg XINFO_GROUPS_Args[] = {
    {"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/***** XINFO HELP *****/

/* XINFO HELP history */
#define XINFO_HELP_History NULL

/* XINFO HELP tips */
#define XINFO_HELP_tips NULL

/***** XINFO STREAM *****/

/* XINFO STREAM history */
commandHistory XINFO_STREAM_History[] = {
    {"7.0.0", "Added the `max-deleted-entry-id`, `entries-added`, `recorded-first-entry-id`, `entries-read` and `lag` fields"},
    {0}
};

```

```

};

/* XINFO STREAM tips */
#define XINFO_STREAM_tips NULL

/* XINFO STREAM full argument table */
struct redisCommandArg XINFO_STREAM_full_Subargs[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* XINFO STREAM argument table */
struct redisCommandArg XINFO_STREAM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"full", ARG_TYPE_BLOCK, -1, "FULL", NULL, NULL, CMD_ARG_OPTIONAL, .subargs=XINFO_STREAM_
{0}
};

/* XINFO command table */
struct redisCommand XINFO_Subcommands[] = {
{"consumers", "List the consumers in a consumer
group", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XINFO_CONSUMERS_
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XINFO_CONSUMERS_Args},
{"groups", "List the consumer groups of a
stream", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XINFO_GROUPS_Hi
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XINFO_GROUPS_Args},
{"help", "Show helpful text about the different
subcommands", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XINFO_HELP
{"stream", "Get information about a
stream", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XINFO_STREAM_Hi
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XINFO_STREAM_Args},
{0}
};

/***** XINFO *****/

/* XINFO history */
#define XINFO_History NULL

/* XINFO tips */
#define XINFO_tips NULL

/***** XLEN *****/

/* XLEN history */
#define XLEN_History NULL

/* XLEN tips */

```

```

#define XLEN_tips NULL

/* XLEN argument table */
struct redisCommandArg XLEN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** XPENDING *****/

/* XPENDING history */
commandHistory XPENDING_History[] = {
{"6.2.0", "Added the `IDLE` option and exclusive range intervals."},
{0}
};

/* XPENDING tips */
const char *XPENDING_tips[] = {
"nondeterministic_output",
NULL
};

/* XPENDING filters argument table */
struct redisCommandArg XPENDING_filters_Subargs[] = {
{"min-idle-time", ARG_TYPE_INTEGER, -1, "IDLE", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{"start", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"end", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"consumer", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/* XPENDING argument table */
struct redisCommandArg XPENDING_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"group", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"filters", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=XPENDING_filters_Subargs},
{0}
};

/***** XRange *****/

/* XRange history */
commandHistory XRange_History[] = {
{"6.2.0", "Added exclusive ranges."},
{0}
};

/* XRange tips */
#define XRange_tips NULL

```

```

/* XRANGE argument table */
struct redisCommandArg XRANGE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"end", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** XREAD *****/

/* XREAD history */
#define XREAD_History NULL

/* XREAD tips */
#define XREAD_tips NULL

/* XREAD streams argument table */
struct redisCommandArg XREAD_streams_Subargs[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/* XREAD argument table */
struct redisCommandArg XREAD_Args[] = {
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{"milliseconds", ARG_TYPE_INTEGER, -1, "BLOCK", NULL, NULL, CMD_ARG_OPTIONAL},
{"streams", ARG_TYPE_BLOCK, -1, "STREAMS", NULL, NULL, CMD_ARG_NONE, .subargs=XREAD_streams_Subargs},
{0}
};

/***** XREADGROUP *****/

/* XREADGROUP history */
#define XREADGROUP_History NULL

/* XREADGROUP tips */
#define XREADGROUP_tips NULL

/* XREADGROUP group_consumer argument table */
struct redisCommandArg XREADGROUP_group_consumer_Subargs[] = {
{"group", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"consumer", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/* XREADGROUP streams argument table */
struct redisCommandArg XREADGROUP_streams_Subargs[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{"id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE},

```

```

{0}
};

/* XREADGROUP argument table */
struct redisCommandArg XREADGROUP_Args[] = {
{"group_consumer", ARG_TYPE_BLOCK, -1, "GROUP", NULL, NULL, CMD_ARG_NONE, .subargs=XREADG
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{"milliseconds", ARG_TYPE_INTEGER, -1, "BLOCK", NULL, NULL, CMD_ARG_OPTIONAL},
{"noack", ARG_TYPE_PURE_TOKEN, -1, "NOACK", NULL, NULL, CMD_ARG_OPTIONAL},
{"streams", ARG_TYPE_BLOCK, -1, "STREAMS", NULL, NULL, CMD_ARG_NONE, .subargs=XREADGROUP_
{0}
};

/***** XREVRANGE *****/

/* XREVRANGE history */
commandHistory XREVRANGE_History[] = {
{"6.2.0", "Added exclusive ranges."},
{0}
};

/* XREVRANGE tips */
#define XREVRANGE_tips NULL

/* XREVRANGE argument table */
struct redisCommandArg XREVRANGE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"end", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, "COUNT", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** XSETID *****/

/* XSETID history */
commandHistory XSETID_History[] = {
{"7.0.0", "Added the `entries_added` and `max_deleted_entry_id` arguments."},
{0}
};

/* XSETID tips */
#define XSETID_tips NULL

/* XSETID argument table */
struct redisCommandArg XSETID_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"last-id", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"entries_added", ARG_TYPE_INTEGER, -1, "ENTRIESADDED", NULL, NULL, CMD_ARG_OPTIONAL},
{"max_deleted_entry_id", ARG_TYPE_STRING, -1, "MAXDELETEDID", NULL, NULL, CMD_ARG_OPTION
{0}

```

```

};

/***** XTRIM *****/

/* XTRIM history */
commandHistory XTRIM_History[] = {
{"6.2.0", "Added the `MINID` trimming strategy and the `LIMIT` option."},
{0}
};

/* XTRIM tips */
const char *XTRIM_tips[] = {
"nondeterministic_output",
NULL
};

/* XTRIM trim strategy argument table */
struct redisCommandArg XTRIM_trim_strategy_Subargs[] = {
{"maxlen", ARG_TYPE_PURE_TOKEN, -1, "MAXLEN", NULL, NULL, CMD_ARG_NONE},
{"minid", ARG_TYPE_PURE_TOKEN, -1, "MINID", NULL, "6.2.0", CMD_ARG_NONE},
{0}
};

/* XTRIM trim operator argument table */
struct redisCommandArg XTRIM_trim_operator_Subargs[] = {
{"equal", ARG_TYPE_PURE_TOKEN, -1, "=", NULL, NULL, CMD_ARG_NONE},
{"approximately", ARG_TYPE_PURE_TOKEN, -1, "~", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* XTRIM trim argument table */
struct redisCommandArg XTRIM_trim_Subargs[] = {
{"strategy", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=XTRIM_trim_stra},
{"operator", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=XTRIM_trim_},
{"threshold", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"count", ARG_TYPE_INTEGER, -1, "LIMIT", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{0}
};

/* XTRIM argument table */
struct redisCommandArg XTRIM_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"trim", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_NONE, .subargs=XTRIM_trim_Subargs},
{0}
};

/***** APPEND *****/

/* APPEND history */
#define APPEND_History NULL

```



```

/* APPEND tips */
#define APPEND_tips NULL

/* APPEND argument table */
struct redisCommandArg APPEND_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** DECR *****/

/* DECR history */
#define DECR_History NULL

/* DECR tips */
#define DECR_tips NULL

/* DECR argument table */
struct redisCommandArg DECR_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** DECRBY *****/

/* DECRBY history */
#define DECRBY_History NULL

/* DECRBY tips */
#define DECRBY_tips NULL

/* DECRBY argument table */
struct redisCommandArg DECRBY_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"decrement", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** GET *****/

/* GET history */
#define GET_History NULL

/* GET tips */
#define GET_tips NULL

/* GET argument table */
struct redisCommandArg GET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

};

/***** GETDEL *****/

/* GETDEL history */
#define GETDEL_History NULL

/* GETDEL tips */
#define GETDEL_tips NULL

/* GETDEL argument table */
struct redisCommandArg GETDEL_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** GETEX *****/

/* GETEX history */
#define GETEX_History NULL

/* GETEX tips */
#define GETEX_tips NULL

/* GETEX expiration argument table */
struct redisCommandArg GETEX_expiration_Subargs[] = {
{"seconds", ARG_TYPE_INTEGER, -1, "EX", NULL, NULL, CMD_ARG_NONE},
{"milliseconds", ARG_TYPE_INTEGER, -1, "PX", NULL, NULL, CMD_ARG_NONE},
{"unix-time-seconds", ARG_TYPE_UNIX_TIME, -1, "EXAT", NULL, NULL, CMD_ARG_NONE},
{"unix-time-milliseconds", ARG_TYPE_UNIX_TIME, -1, "PXAT", NULL, NULL, CMD_ARG_NONE},
{"persist", ARG_TYPE_PURE_TOKEN, -1, "PERSIST", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* GETEX argument table */
struct redisCommandArg GETEX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"expiration", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=GETEX_exp
{0}
};

/***** GETRANGE *****/

/* GETRANGE history */
#define GETRANGE_History NULL

/* GETRANGE tips */
#define GETRANGE_tips NULL

/* GETRANGE argument table */
struct redisCommandArg GETRANGE_Args[] = {

```

```

{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"end", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** GETSET *****/

/* GETSET history */
#define GETSET_History NULL

/* GETSET tips */
#define GETSET_tips NULL

/* GETSET argument table */
struct redisCommandArg GETSET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** INCR *****/

/* INCR history */
#define INCR_History NULL

/* INCR tips */
#define INCR_tips NULL

/* INCR argument table */
struct redisCommandArg INCR_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** INCRBY *****/

/* INCRBY history */
#define INCRBY_History NULL

/* INCRBY tips */
#define INCRBY_tips NULL

/* INCRBY argument table */
struct redisCommandArg INCRBY_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"increment", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** INCRBYFLOAT *****/

```

```

/* INCRBYFLOAT history */
#define INCRBYFLOAT_History NULL

/* INCRBYFLOAT tips */
#define INCRBYFLOAT_tips NULL

/* INCRBYFLOAT argument table */
struct redisCommandArg INCRBYFLOAT_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"increment", ARG_TYPE_DOUBLE, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** LCS *****/

/* LCS history */
#define LCS_History NULL

/* LCS tips */
#define LCS_tips NULL

/* LCS argument table */
struct redisCommandArg LCS_Args[] = {
{"key1", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"key2", ARG_TYPE_KEY, 1, NULL, NULL, NULL, CMD_ARG_NONE},
{"len", ARG_TYPE_PURE_TOKEN, -1, "LEN", NULL, NULL, CMD_ARG_OPTIONAL},
{"idx", ARG_TYPE_PURE_TOKEN, -1, "IDX", NULL, NULL, CMD_ARG_OPTIONAL},
{"len", ARG_TYPE_INTEGER, -1, "MINMATCHLEN", NULL, NULL, CMD_ARG_OPTIONAL},
{"withmatchlen", ARG_TYPE_PURE_TOKEN, -1, "WITHMATCHLEN", NULL, NULL, CMD_ARG_OPTIONAL},
{0}
};

/***** MGET *****/

/* MGET history */
#define MGET_History NULL

/* MGET tips */
const char *MGET_tips[] = {
"request_policy:multi_shard",
NULL
};

/* MGET argument table */
struct redisCommandArg MGET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/***** MSET *****/

```

```

/* MSET history */
#define MSET_History NULL

/* MSET tips */
const char *MSET_tips[] = {
    "request_policy:multi_shard",
    "response_policy:all_succeeded",
    NULL
};

/* MSET key_value argument table */
struct redisCommandArg MSET_key_value_Subargs[] = {
    {"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
    {"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* MSET argument table */
struct redisCommandArg MSET_Args[] = {
    {"key_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=MSET_key_v
    {0}
};

/***** MSETNX *****/

/* MSETNX history */
#define MSETNX_History NULL

/* MSETNX tips */
const char *MSETNX_tips[] = {
    "request_policy:multi_shard",
    "response_policy:agg_min",
    NULL
};

/* MSETNX key_value argument table */
struct redisCommandArg MSETNX_key_value_Subargs[] = {
    {"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
    {"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
    {0}
};

/* MSETNX argument table */
struct redisCommandArg MSETNX_Args[] = {
    {"key_value", ARG_TYPE_BLOCK, -1, NULL, NULL, NULL, CMD_ARG_MULTIPLE, .subargs=MSETNX_key
    {0}
};

/***** PSETEX *****/

```

```

/* PSETEX history */
#define PSETEX_History NULL

/* PSETEX tips */
#define PSETEX_tips NULL

/* PSETEX argument table */
struct redisCommandArg PSETEX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"milliseconds", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SET *****/

/* SET history */
commandHistory SET_History[] = {
{"2.6.12", "Added the `EX`, `PX`, `NX` and `XX` options."},
{"6.0.0", "Added the `KEEPTTL` option."},
{"6.2.0", "Added the `GET`, `EXAT` and `PXAT` option."},
{"7.0.0", "Allowed the `NX` and `GET` options to be used together."},
{0}
};

/* SET tips */
#define SET_tips NULL

/* SET condition argument table */
struct redisCommandArg SET_condition_Subargs[] = {
{"nx", ARG_TYPE_PURE_TOKEN, -1, "NX", NULL, NULL, CMD_ARG_NONE},
{"xx", ARG_TYPE_PURE_TOKEN, -1, "XX", NULL, NULL, CMD_ARG_NONE},
{0}
};

/* SET expiration argument table */
struct redisCommandArg SET_expiration_Subargs[] = {
{"seconds", ARG_TYPE_INTEGER, -1, "EX", NULL, "2.6.12", CMD_ARG_NONE},
{"milliseconds", ARG_TYPE_INTEGER, -1, "PX", NULL, "2.6.12", CMD_ARG_NONE},
{"unix-time-seconds", ARG_TYPE_UNIX_TIME, -1, "EXAT", NULL, "6.2.0", CMD_ARG_NONE},
{"unix-time-
milliseconds", ARG_TYPE_UNIX_TIME, -1, "PXAT", NULL, "6.2.0", CMD_ARG_NONE},
{"keepttl", ARG_TYPE_PURE_TOKEN, -1, "KEEPTTL", NULL, "6.0.0", CMD_ARG_NONE},
{0}
};

/* SET argument table */
struct redisCommandArg SET_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"condition", ARG_TYPE_ONEOF, -1, NULL, NULL, "2.6.12", CMD_ARG_OPTIONAL, .subargs=SET_co

```

```

{"get", ARG_TYPE_PURE_TOKEN, -1, "GET", NULL, "6.2.0", CMD_ARG_OPTIONAL},
{"expiration", ARG_TYPE_ONEOF, -1, NULL, NULL, NULL, CMD_ARG_OPTIONAL, .subargs=SET_expir
{0}
};

/***** SETEX *****/

/* SETEX history */
#define SETEX_History NULL

/* SETEX tips */
#define SETEX_tips NULL

/* SETEX argument table */
struct redisCommandArg SETEX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"seconds", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SETNX *****/

/* SETNX history */
#define SETNX_History NULL

/* SETNX tips */
#define SETNX_tips NULL

/* SETNX argument table */
struct redisCommandArg SETNX_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SETRANGE *****/

/* SETRANGE history */
#define SETRANGE_History NULL

/* SETRANGE tips */
#define SETRANGE_tips NULL

/* SETRANGE argument table */
struct redisCommandArg SETRANGE_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"offset", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"value", ARG_TYPE_STRING, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

```

```

/***** STRLEN *****/

/* STRLEN history */
#define STRLEN_History NULL

/* STRLEN tips */
#define STRLEN_tips NULL

/* STRLEN argument table */
struct redisCommandArg STRLEN_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** SUBSTR *****/

/* SUBSTR history */
#define SUBSTR_History NULL

/* SUBSTR tips */
#define SUBSTR_tips NULL

/* SUBSTR argument table */
struct redisCommandArg SUBSTR_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_NONE},
{"start", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{"end", ARG_TYPE_INTEGER, -1, NULL, NULL, NULL, CMD_ARG_NONE},
{0}
};

/***** DISCARD *****/

/* DISCARD history */
#define DISCARD_History NULL

/* DISCARD tips */
#define DISCARD_tips NULL

/***** EXEC *****/

/* EXEC history */
#define EXEC_History NULL

/* EXEC tips */
#define EXEC_tips NULL

/***** MULTI *****/

/* MULTI history */
#define MULTI_History NULL

```



```

/* MULTI tips */
#define MULTI_tips NULL

/***** UNWATCH *****/

/* UNWATCH history */
#define UNWATCH_History NULL

/* UNWATCH tips */
#define UNWATCH_tips NULL

/***** WATCH *****/

/* WATCH history */
#define WATCH_History NULL

/* WATCH tips */
#define WATCH_tips NULL

/* WATCH argument table */
struct redisCommandArg WATCH_Args[] = {
{"key", ARG_TYPE_KEY, 0, NULL, NULL, NULL, CMD_ARG_MULTIPLE},
{0}
};

/* Main command table */
struct redisCommand redisCommandTable[] = {
/* bitmap */
{"bitcount", "Count set bits in a string", "0(N)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_BITMAP, BITCOUNT_History,
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=BITCOUNT_Args},
{"bitfield", "Perform arbitrary bitfield integer operations on strings", "0(1) for each subcommand specified", "3.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_BITMAP, BITFIELD_History, BI
{"This command allows both access and modification of the key", CMD_KEY_RW|CMD_KEY_UPDATE|CMD_KEY_ACCESS|CMD_KEY_VARIABLE_FLAGS, KSPEC_BS_INDE
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, bitfieldGetKeys, .args=BITFIELD_Args},
{"bitfield_ro", "Perform arbitrary bitfield integer operations on strings. Read-only variant of BITFIELD", "0(1) for each subcommand specified", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_BITMAP, BITFIELD_RO_History
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=BITFIELD_RO_Args},
{"bitop", "Perform bitwise operations between strings", "0(N)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_BITMAP, BITOP_History,
{{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{3}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=BITOP_Args},
{"bitpos", "Find first bit set or clear in a

```

```

string", "0(N)", "2.8.7", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_BITMAP, BITPOS_History,
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=BITPOS_Args},
{"getbit", "Returns the bit value at offset in the string value stored at
key", "0(1)", "2.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_BITMAP, GETBIT_History, GET
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GETBIT_Args},
{"setbit", "Sets or clears the bit at offset in the string value stored at
key", "0(1)", "2.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_BITMAP, SETBIT_History, SET
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SETBIT_Args},
/* cluster */
{"asking", "Sent by cluster clients after an -ASK
redirect", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, ASKING_Histo
{"cluster", "A container for cluster commands", "Depends on
subcommand.", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, CLUSTER_History,
{"readonly", "Enables read queries for a connection to a cluster replica
node", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, READONLY_History
{"readwrite", "Disables read queries for a connection to a cluster replica
node", "0(1)", "3.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CLUSTER, READWRITE_Histor
/* connection */
{"auth", "Authenticate to the server", "0(N) where N is the number of passwords
defined for the
user", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, AUTH_History, AUTH_ti
{"client", "A container for client connection commands", "Depends on
subcommand.", "2.4.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, CLIENT_Histor
{"echo", "Echo the given
string", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, ECHO_Histor
{"hello", "Handshake with
Redis", "0(1)", "6.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, HELLO_Histor
{"ping", "Ping the
server", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, PING_Histor
{"quit", "Close the
connection", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, QUIT_Hi
{"reset", "Reset the
connection", "0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, RESET_H
{"select", "Change the selected database for the current
connection", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_CONNECTION, SELECT_
/* generic */
{"copy", "Copy a key", "0(N) worst case for collections, where N is the number of
nested items. 0(1) for string
values.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, COPY_History, COPY_ti
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=COPY_Args},
{"del", "Delete a key", "0(N) where N is the number of keys that will be removed.
When a key to remove holds a value other than a string, the individual
complexity for this key is O(M) where M is the number of elements in the list,
set, sorted set or hash. Removing a single key that holds a string value is
0(1).", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, DEL_History, DEL_tips, d

```

```

{{NULL,CMD_KEY_RM|CMD_KEY_DELETE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={-1,1,0}}},.args=DEL_Args},
{"dump","Return a serialized version of the value stored at the specified
key.", "0(1) to access the key and additional O(N*M) to serialize it, where N is
the number of Redis objects composing the value and M their average size. For
small string values the time complexity is thus O(1)+O(1*M) where M is small,
so simply
O(1).", "2.6.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,DUMP_History,DUMP_tips
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=DUMP_Args},
{"exists","Determine if a key exists", "0(N) where N is the number of keys to
check.", "1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,EXISTS_History,EXISTS
{{NULL,CMD_KEY_RO,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{-1,1,0}}},.args=EXISTS_Args},
{"expire","Set a key's time to live in
seconds", "0(1)", "1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,EXPIRE_Histor
{{NULL,CMD_KEY_RW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=EXPIRE_Args},
{"expireat","Set the expiration for a key as a UNIX
timestamp", "0(1)", "1.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,EXPIREAT_Hi
{{NULL,CMD_KEY_RW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=EXPIREAT_Args},
{"expiretime","Get the expiration Unix timestamp for a
key", "0(1)", "7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,EXPIRETIME_Histor
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=EXPIRETIME_Args},
{"keys","Find all keys matching the given pattern", "0(N) with N being the
number of keys in the database, under the assumption that the key names in the
database and the given pattern have limited
length.", "1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,KEYS_History,KEYS_ti
{"migrate","Atomically transfer a key from a Redis instance to another
one.", "This command actually executes a DUMP+DEL in the source instance, and a
RESTORE in the target instance. See the pages of these commands for time
complexity. Also an O(N) data transfer between the two instances is
performed.", "2.6.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,MIGRATE_History,M
{{NULL,CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE,KSPEC_BS_INDEX,.bs.index=
{3},KSPEC_FK_RANGE,.fk.range={0,1,0}},
{NULL,CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE|CMD_KEY_INCOMPLETE,KSPEC_BS_KEYWORD
{"KEYS",-2},KSPEC_FK_RANGE,.fk.range=
{-1,1,0}}},migrateGetKeys,.args=MIGRATE_Args},
{"move","Move a key to another
database", "0(1)", "1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,MOVE_History
{{NULL,CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=MOVE_Args},
{"object","A container for object introspection commands", "Depends on
subcommand.", "2.2.3",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,OBJECT_History,0
{"persist","Remove the expiration from a
key", "0(1)", "2.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,PERSIST_History,P
{{NULL,CMD_KEY_RW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=PERSIST_Args},
{"pexpire","Set a key's time to live in

```

```

milliseconds", "0(1)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, PEXPIRE_
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=PEXPIRE_Args},
{"pexpireat", "Set the expiration for a key as a UNIX timestamp specified in
milliseconds", "0(1)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, PEXPIREA
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=PEXPIREAT_Args},
{"pexpiretime", "Get the expiration Unix timestamp for a key in
milliseconds", "0(1)", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, PEXPIRET
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=PEXPIRETIME_Args},
{"pttl", "Get the time to live for a key in
milliseconds", "0(1)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, PTTL_His
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=PTTL_Args},
{"randomkey", "Return a random key from the
keyspace", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, RANDOMKEY_Hi
{"rename", "Rename a
key", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, RENAME_History, RE
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=RENAME_Args},
{"renamenx", "Rename a key, only if the new key does not
exist", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, RENAMENX_Histor
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_OW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=RENAMENX_Args},
{"restore", "Create a key using the provided serialized value, previously
obtained using DUMP.", "0(1) to create the new key and additional O(N*M) to
reconstruct the serialized value, where N is the number of Redis objects
composing the value and M their average size. For small string values the time
complexity is thus O(1)+O(1*M) where M is small, so simply O(1). However for
sorted set values the complexity is O(N*M*log(N)) because inserting values into
sorted sets is
O(log(N)).", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, RESTORE_History, R
{{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=RESTORE_Args},
{"scan", "Incrementally iterate the keys space", "0(1) for every call. O(N) for a
complete iteration, including enough command calls for the cursor to return
back to 0. N is the number of elements inside the
collection.", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, SCAN_History, SCA
{"sort", "Sort the elements in a list, set or sorted set", "O(N*M*log(M)) where N
is the number of elements in the list or set to sort, and M the number of
returned elements. When the elements are not sorted, complexity is
O(N).", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GENERIC, SORT_History, SORT_tips
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, {"For the optional BY/GET keyword. It is
marked 'unknown' because the key names derive from the content of the key we
sort", CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_UNKNOWN, {{0}}, KSPEC_FK_UNKNOWN, {{0}}},

```

```

{"For the optional STORE keyword. It is marked 'unknown' because the keyword
can appear anywhere in the argument
array",CMD_KEY_OW|CMD_KEY_UPDATE,KSPEC_BS_UNKNOWN,{{0}},KSPEC_FK_UNKNOWN,
{{0}}}},sortGetKeys,.args=SORT_Args},
{"sort_ro","Sort the elements in a list, set or sorted set. Read-only variant
of SORT.", "0(N+M*log(M)) where N is the number of elements in the list or set
to sort, and M the number of returned elements. When the elements are not
sorted, complexity is
0(N).", "7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,SORT_RO_History,SORT_R
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}}, {"For the optional BY/GET keyword. It is
marked 'unknown' because the key names derive from the content of the key we
sort",CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_UNKNOWN,{{0}},KSPEC_FK_UNKNOWN,
{{0}}}},sortROGetKeys,.args=SORT_RO_Args},
{"touch","Alters the last access time of a key(s). Returns the number of
existing keys specified.", "0(N) where N is the number of keys that will be
touched.", "3.2.1",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,TOUCH_History,TOUCH
{{NULL,CMD_KEY_RO,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{-1,1,0}}},.args=TOUCH_Args},
{"ttl","Get the time to live for a key in
seconds", "0(1)", "1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,TTL_History,T
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=TTL_Args},
{"type","Determine the type stored at
key", "0(1)", "1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,TYPE_History,TYPE
{{NULL,CMD_KEY_RO,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{0,1,0}}},.args=TYPE_Args},
{"unlink","Delete a key asynchronously in another thread. Otherwise it is just
as DEL, but non blocking.", "0(1) for each key removed regardless of its size.
Then the command does 0(N) work in a different thread in order to reclaim
memory, where N is the number of allocations the deleted objects where composed
of.", "4.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,UNLINK_History,UNLINK_ti
{{NULL,CMD_KEY_RM|CMD_KEY_DELETE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={-1,1,0}}},.args=UNLINK_Args},
{"wait","Wait for the synchronous replication of all the write commands sent in
the context of the current
connection", "0(1)", "3.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GENERIC,WAIT_Histo
/* geo */
{"geoadd","Add one or more geospatial items in the geospatial index represented
using a sorted set", "0(log(N)) for each item added, where N is the number of
elements in the sorted
set.", "3.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GEO,GEOADD_History,GEOADD_tips,
{{NULL,CMD_KEY_RW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=GEOADD_Args},
{"geodist","Returns the distance between two members of a geospatial
index", "0(log(N))", "3.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_GEO,GEODIST_Histor
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=GEODIST_Args},
{"geohash","Returns members of a geospatial index as standard geohash
strings", "0(log(N)) for each member requested, where N is the number of
elements in the sorted

```



```

set.", "3.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GEO, GEOHASH_History, GEOHASH_tip
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GEOHASH_Args},
{"geopos", "Returns longitude and latitude of members of a geospatial
index", "0(N) where N is the number of members
requested.", "3.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GEO, GEOPOS_History, GEOPOS
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GEOPOS_Args},
{"georadius", "Query a sorted set representing a geospatial index to fetch
members matching a given maximum distance from a point", "0(N+log(M)) where N is
the number of elements inside the bounding box of the circular area delimited
by center and radius and M is the number of items inside the
index.", "3.2.0", CMD_DOC_DEPRECATED, "`GEOSEARCH` and `GEOSEARCHSTORE` with the
`BYRADIUS`
argument", "6.2.0", COMMAND_GROUP_GEO, GEORADIUS_History, GEORADIUS_tips, georadiusComm
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_KEYWORD, .bs.keyword=
{"STORE", 6}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_KEYWORD, .bs.keyword=
{"STOREDIST", 6}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, georadiusGetKeys, .args=GEORADIUS_Args},
{"georadiusbymember", "Query a sorted set representing a geospatial index to
fetch members matching a given maximum distance from a member", "0(N+log(M))
where N is the number of elements inside the bounding box of the circular area
delimited by center and radius and M is the number of items inside the
index.", "3.2.0", CMD_DOC_DEPRECATED, "`GEOSEARCH` and `GEOSEARCHSTORE` with the
`BYRADIUS` and `FROMMEMBER`
arguments", "6.2.0", COMMAND_GROUP_GEO, GEORADIUSBYMEMBER_History, GEORADIUSBYMEMBER_t
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_KEYWORD, .bs.keyword=
{"STORE", 5}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_KEYWORD, .bs.keyword=
{"STOREDIST", 5}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, georadiusGetKeys, .args=GEORADIUSBYMEMBER_Args},
{"georadiusbymember_ro", "A read-only variant for
GEORADIUSBYMEMBER", "0(N+log(M)) where N is the number of elements inside the
bounding box of the circular area delimited by center and radius and M is the
number of items inside the index.", "3.2.10", CMD_DOC_DEPRECATED, "`GEOSEARCH`
with the `BYRADIUS` and `FROMMEMBER`
arguments", "6.2.0", COMMAND_GROUP_GEO, GEORADIUSBYMEMBER_RO_History, GEORADIUSBYMEMBE
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GEORADIUSBYMEMBER_RO_Args},
{"georadius_ro", "A read-only variant for GEORADIUS", "0(N+log(M)) where N is the
number of elements inside the bounding box of the circular area delimited by
center and radius and M is the number of items inside the
index.", "3.2.10", CMD_DOC_DEPRECATED, "`GEOSEARCH` with the `BYRADIUS`
argument", "6.2.0", COMMAND_GROUP_GEO, GEORADIUS_RO_History, GEORADIUS_RO_tips, georadi
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GEORADIUS_RO_Args},

```

```

{"geosearch","Query a sorted set representing a geospatial index to fetch
members inside an area of a box or a circle.", "O(N+log(M)) where N is the
number of elements in the grid-aligned bounding box area around the shape
provided as the filter and M is the number of items inside the
shape", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GEO, GEOSEARCH_History, GEOSEARCH
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GEOSEARCH_Args},
{"geosearchstore","Query a sorted set representing a geospatial index to fetch
members inside an area of a box or a circle, and store the result in another
key.", "O(N+log(M)) where N is the number of elements in the grid-aligned
bounding box area around the shape provided as the filter and M is the number
of items inside the
shape", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_GEO, GEOSEARCHSTORE_History, GEO
{{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GEOSEARCHSTORE_Args},
/* hash */
{"hdel","Delete one or more hash fields", "O(N) where N is the number of fields
to be
removed.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HDEL_History, HDEL_tips
{{NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=HDEL_Args},
{"hexists","Determine if a hash field
exists", "O(1)", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HEXISTS_History, H
{{NULL, CMD_KEY_RO, KSPEC_BS_INDEX, .bs.index={1}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, .args=HEXISTS_Args},
{"hget","Get the value of a hash
field", "O(1)", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HGET_History, HGET_
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=HGET_Args},
{"hgetall","Get all the fields and values in a hash", "O(N) where N is the size
of the
hash.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HGETALL_History, HGETALL_t
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=HGETALL_Args},
{"hincrby","Increment the integer value of a hash field by the given
number", "O(1)", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HINCRBY_History, H
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=HINCRBY_Args},
{"hincrbyfloat","Increment the float value of a hash field by the given
amount", "O(1)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HINCRBYFLOAT_Hist
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=HINCRBYFLOAT_Args},
{"hkeys","Get all the fields in a hash", "O(N) where N is the size of the
hash.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HKEYS_History, HKEYS_tips,
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=HKEYS_Args},
{"hlen","Get the number of fields in a
hash", "O(1)", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HLEN_History, HLEN_t
{{NULL, CMD_KEY_RO, KSPEC_BS_INDEX, .bs.index={1}, KSPEC_FK_RANGE, .fk.range=

```

```

{0,1,0}}},.args=HLEN_Args},
{"hmmget","Get the values of all the given hash fields","0(N) where N is the
number of fields being
requested.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HMGET_History, HMGET_
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=HMGET_Args},
{"hmmset","Set multiple hash fields to multiple values","0(N) where N is the
number of fields being set.", "2.0.0", CMD_DOC_DEPRECATED, "`HSET` with multiple
field-value
pairs", "4.0.0", COMMAND_GROUP_HASH, HMSET_History, HMSET_tips, hsetCommand, -4, CMD_WRIT
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=HMSET_Args},
{"hrandfield","Get one or multiple random fields from a hash","0(N) where N is
the number of fields
returned", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HRANDFIELD_History, HRA
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=HRANDFIELD_Args},
{"hscan","Incrementally iterate hash fields and associated values","0(1) for
every call. 0(N) for a complete iteration, including enough command calls for
the cursor to return back to 0. N is the number of elements inside the
collection..", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HSCAN_History, HSCA
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=HSCAN_Args},
{"hset","Set the string value of a hash field","0(1) for each field/value pair
added, so 0(N) to add N field/value pairs when the command is called with
multiple field/value
pairs.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HSET_History, HSET_tips, h
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=HSET_Args},
{"hsetnx","Set the value of a hash field, only if the field does not
exist", "0(1)", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HSETNX_History, HSE
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=HSETNX_Args},
{"hstrlen","Get the length of the value of a hash
field", "0(1)", "3.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HSTRLEN_History, HS
{{NULL, CMD_KEY_RO, KSPEC_BS_INDEX, .bs.index={1}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}},.args=HSTRLEN_Args},
{"hvals","Get all the values in a hash","0(N) where N is the size of the
hash.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HASH, HVALS_History, HVALS_tips,
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=HVALS_Args},
/* hyperloglog */
{"pfadd","Adds the specified elements to the specified HyperLogLog.", "0(1) to
add every
element.", "2.8.9", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HYPERLOGLOG, PFADD_History, P
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=PFADD_Args},
{"pfcoun", "Return the approximated cardinality of the set(s) observed by the
HyperLogLog at key(s).", "0(1) with a very small average constant time when
called with a single key. 0(N) with N being the number of keys, and much bigger
constant times, when called with multiple

```



```

keys.", "2.8.9", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HYPERLOGLOG, PFCOUNT_History, PF
{"RW because it may change the internal representation of the key, and
propagate to replicas", CMD_KEY_RW|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=PFCOUNT_Args},
{"pfdebug", "Internal commands for debugging HyperLogLog
values", "N/A", "2.8.9", CMD_DOC_SYSCMD, NULL, NULL, COMMAND_GROUP_HYPERLOGLOG, PFDEBUG_H
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}},
{"pfmerge", "Merge N different HyperLogLogs into a single one.", "0(N) to merge N
HyperLogLogs, but with high constant
times.", "2.8.9", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_HYPERLOGLOG, PFMERGE_History, P
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_R0|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=PFMERGE_Args},
{"pfselftest", "An internal command for testing HyperLogLog
values", "N/A", "2.8.9", CMD_DOC_SYSCMD, NULL, NULL, COMMAND_GROUP_HYPERLOGLOG, PFSELFTESTES
/* list */
{"blmove", "Pop an element from a list, push it to another list and return it;
or block until one is
available", "0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, BLMOVE_History
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=BLMOVE_Args},
{"blmpop", "Pop elements from a list, or block until one is available", "0(N+M)
where N is the number of provided keys and M is the number of elements
returned.", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, BLMPOP_History, BLMPOP
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_KEYNUM, .fk.keynum={0,1,1}}}, blmpopGetKeys, .args=BLMPOP_Args},
{"blpop", "Remove and get the first element in a list, or block until one is
available", "0(N) where N is the number of provided
keys.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, BLPPOP_History, BLPPOP_tips,
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-2,1,0}}}, .args=BLPOP_Args},
{"brpop", "Remove and get the last element in a list, or block until one is
available", "0(N) where N is the number of provided
keys.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, BRPOP_History, BRPOP_tips,
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-2,1,0}}}, .args=BRPOP_Args},
{"brpoplpush", "Pop an element from a list, push it to another list and return
it; or block until one is
available", "0(1)", "2.2.0", CMD_DOC_DEPRECATED, "`BLMOVE` with the `RIGHT` and
`LEFT`
arguments", "6.2.0", COMMAND_GROUP_LIST, BRPOPLPUSH_History, BRPOPLPUSH_tips, brpoplpush
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=BRPOPLPUSH_Args},
{"lindex", "Get an element from a list by its index", "0(N) where N is the number
of elements to traverse to get to the element at index. This makes asking for

```

the first or the last element of the list

```
0(1).", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LINDEX_History, LINDEX_tips
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LINDEX_Args},
{"linsert", "Insert an element before or after another element in a list", "0(N)
where N is the number of elements to traverse before seeing the value pivot.
This means that inserting somewhere on the left end on the list (head) can be
considered 0(1) and inserting somewhere on the right end (tail) is
0(N).", "2.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LINSERT_History, LINSERT_t
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LINSERT_Args},
{"llen", "Get the length of a
list", "0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LLEN_History, LLEN_t
{{NULL, CMD_KEY_RO, KSPEC_BS_INDEX, .bs.index={1}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, .args=LLEN_Args},
{"lmove", "Pop an element from a list, push it to another list and return
it", "0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LMOVE_History, LMOVE_t
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LMOVE_Args},
{"lmpop", "Pop elements from a list", "0(N+M) where N is the number of provided
keys and M is the number of elements
returned.", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LMPOP_History, LMPOP_t
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_KEYNUM, .fk.keynum={0,1,1}}}, lmpopGetKeys, .args=LMPOP_Args},
{"lpop", "Remove and get the first elements in a list", "0(N) where N is the
number of elements
returned", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LPOP_History, LPOP_tips
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LPOP_Args},
{"lpos", "Return the index of matching elements on a list", "0(N) where N is the
number of elements in the list, for the average case. When searching for
elements near the head or the tail of the list, or when the MAXLEN option is
provided, the command may run in constant
time.", "6.0.6", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LPOS_History, LPOS_tips, lp
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LPOS_Args},
{"lpush", "Prepend one or multiple elements to a list", "0(1) for each element
added, so 0(N) to add N elements when the command is called with multiple
arguments.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LPUSH_History, LPUSH_
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LPUSH_Args},
{"lpushx", "Prepend an element to a list, only if the list exists", "0(1) for
each element added, so 0(N) to add N elements when the command is called with
multiple
arguments.", "2.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LPUSHX_History, LPUSH
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LPUSHX_Args},
{"lrange", "Get a range of elements from a list", "0(S+N) where S is the distance
of start offset from HEAD for small lists, from nearest end (HEAD or TAIL) for
```

```

large lists; and N is the number of elements in the specified
range.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LRANGE_History, LRANGE_tips
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LRANGE_Args},
{"lrem", "Remove elements from a list", "O(N+M) where N is the length of the list
and M is the number of elements
removed.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LREM_History, LREM_tips
{{NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LREM_Args},
{"lset", "Set the value of an element in a list by its index", "O(N) where N is
the length of the list. Setting either the first or the last element of the
list is
O(1).", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LSET_History, LSET_tips, ls
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LSET_Args},
{"ltrim", "Trim a list to the specified range", "O(N) where N is the number of
elements to be removed by the
operation.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, LTRIM_History, LTRIM_
{{NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=LTRIM_Args},
{"rpop", "Remove and get the last elements in a list", "O(N) where N is the
number of elements
returned", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, RPOP_History, RPOP_tips
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=RPOP_Args},
{"rpoplpush", "Remove the last element in a list, prepend it to another list and
return it", "O(1)", "1.2.0", CMD_DOC_DEPRECATED, "`LMOVE` with the `RIGHT` and
`LEFT`
arguments", "6.2.0", COMMAND_GROUP_LIST, RPOPLPUSH_History, RPOPLPUSH_tips, rpoplpushCo
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=RPOPLPUSH_Args},
{"rpush", "Append one or multiple elements to a list", "O(1) for each element
added, so O(N) to add N elements when the command is called with multiple
arguments.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, RPUSH_History, RPUSH_
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=RPUSH_Args},
{"rpushx", "Append an element to a list, only if the list exists", "O(1) for each
element added, so O(N) to add N elements when the command is called with
multiple
arguments.", "2.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_LIST, RPUSHX_History, RPUSH
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=RPUSHX_Args},
/* pubsub */
{"psubscribe", "Listen for messages published to channels matching the given
patterns", "O(N) where N is the number of patterns the client is already
subscribed
to.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_PUBSUB, PSUBSCRIBE_History, PSUBSC
{"publish", "Post a message to a channel", "O(N+M) where N is the number of
clients subscribed to the receiving channel and M is the total number of

```

```

subscribed patterns (by any
client).","2.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,PUBLISH_History,PUBL
{"pubsub","A container for Pub/Sub commands","Depends on
subcommand.","2.8.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,PUBSUB_History,PUB
{"punsubscribe","Stop listening for messages posted to channels matching the
given patterns","0(N+M) where N is the number of patterns the client is already
subscribed and M is the number of total patterns subscribed in the system (by
any
client).","2.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,PUNSUBSCRIBE_History
{"spublish","Post a message to a shard channel","0(N) where N is the number of
clients subscribed to the receiving shard
channel.","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,SPUBLISH_History,SPU
{{NULL,CMD_KEY_NOT_KEY,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{0,1,0}}},.args=SPUBLISH_Args},
{"ssubscribe","Listen for messages published to the given shard channels","0(N)
where N is the number of shard channels to subscribe
to.","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,SSUBSCRIBE_History,SSUBSC
{{NULL,CMD_KEY_NOT_KEY,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{-1,1,0}}},.args=SSUBSCRIBE_Args},
{"subscribe","Listen for messages published to the given channels","0(N) where
N is the number of channels to subscribe
to.","2.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,SUBSCRIBE_History,SUBSCRI
{"sunsubscribe","Stop listening for messages posted to the given shard
channels","0(N) where N is the number of clients already subscribed to a
channel.","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,SUNSUBSCRIBE_History
{{NULL,CMD_KEY_NOT_KEY,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{-1,1,0}}},.args=SUNSUBSCRIBE_Args},
{"unsubscribe","Stop listening for messages posted to the given channels","0(N)
where N is the number of clients already subscribed to a
channel.","2.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_PUBSUB,UNSUBSCRIBE_History,
/* scripting */
{"eval","Execute a Lua script server side","Depends on the script that is
executed.","2.6.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,EVAL_History,EVA
{"We cannot tell how the keys will be used so we assume the worst, RW and
UPDATE",CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{2},KSPEC_FK_KEYNUM,.fk.keynum={0,1,1}}},evalGetKeys,.args=EVAL_Args},
{"evalsha","Execute a Lua script server side","Depends on the script that is
executed.","2.6.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,EVALSHA_History,
{{NULL,CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{2},KSPEC_FK_KEYNUM,.fk.keynum={0,1,1}}},evalGetKeys,.args=EVALSHA_Args},
{"evalsha_ro","Execute a read-only Lua script server side","Depends on the
script that is
executed.","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,EVALSHA_RO_Histo
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{2},KSPEC_FK_KEYNUM,.fk.keynum={0,1,1}}},evalGetKeys,.args=EVALSHA_RO_Args},
{"eval_ro","Execute a read-only Lua script server side","Depends on the script
that is
executed.","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,EVAL_RO_History,
{"We cannot tell how the keys will be used so we assume the worst, R0 and
ACCESS",CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{2},KSPEC_FK_KEYNUM,.fk.keynum={0,1,1}}},evalGetKeys,.args=EVAL_RO_Args},

```

```

{"fcall","Invoke a function","Depends on the function that is
executed.", "7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FCALL_History,FC
{"We cannot tell how the keys will be used so we assume the worst, RW and
UPDATE",CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{2},KSPEC_FK_KEYNUM,.fk.keynum={0,1,1}},functionGetKeys,.args=FCALL_Args},
{"fcall_ro","Invoke a read-only function","Depends on the function that is
executed.", "7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FCALL_RO_History
{"We cannot tell how the keys will be used so we assume the worst, R0 and
ACCESS",CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{2},KSPEC_FK_KEYNUM,.fk.keynum={0,1,1}},functionGetKeys,.args=FCALL_RO_Args},
{"function","A container for function commands","Depends on
subcommand.", "7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,FUNCTION_Histo
{"script","A container for Lua scripts management commands","Depends on
subcommand.", "2.6.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SCRIPTING,SCRIPT_History
/* sentinel */
{"sentinel","A container for Sentinel commands","Depends on
subcommand.", "2.8.4",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SENTINEL,SENTINEL_Histor
/* server */
{"acl","A container for Access List Control commands ","Depends on
subcommand.", "6.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,ACL_History,ACL_t
{"bgrewriteaof","Asynchronously rewrite the append-only
file","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,BGREWRITEAOF_Hist
{"bgsave","Asynchronously save the dataset to
disk","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,BGSAVE_History,BG
{"command","Get array of Redis command details","0(N) where N is the total
number of Redis
commands","2.8.13",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,COMMAND_History,COM
{"config","A container for server configuration commands","Depends on
subcommand.", "2.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,CONFIG_History,CO
{"dbsize","Return the number of keys in the selected
database","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,DBSIZE_Histor
{"debug","A container for debugging commands","Depends on
subcommand.", "1.0.0",CMD_DOC_SYSCMD,NULL,NULL,COMMAND_GROUP_SERVER,DEBUG_History,D
{"failover","Start a coordinated failover between this server and one of its
replicas.", "0(1)","6.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,FAILOVER_His
{"flushall","Remove all keys from all databases","0(N) where N is the total
number of keys in all
databases","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,FLUSHALL_History,FL
{"flushdb","Remove all keys from the current database","0(N) where N is the
number of keys in the selected
database","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,FLUSHDB_History,FLUS
{"info","Get information and statistics about the
server","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,INFO_History,IN
{"lastsave","Get the UNIX time stamp of the last successful save to
disk","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,LASTSAVE_History,
{"latency","A container for latency diagnostics commands","Depends on
subcommand.", "2.8.13",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,LATENCY_History,
{"lolwut","Display some computer art and the Redis
version",NULL,"5.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,LOLWUT_History,L
{"memory","A container for memory diagnostics commands","Depends on
subcommand.", "4.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,MEMORY_History,ME

```



```

{"module","A container for module commands","Depends on
subcommand.","4.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,MODULE_History,MO
{"monitor","Listen for all requests received by the server in real
time",NULL,"1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,MONITOR_History,MON
{"psync","Internal command used for
replication",NULL,"2.8.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,PSYNC_Histor
{"replconf","An internal command for configuring the replication
stream","0(1)","3.0.0",CMD_DOC_SYSCMD,NULL,NULL,COMMAND_GROUP_SERVER,REPLCONF_Hist
{"replicaof","Make the server a replica of another instance, or promote it as
master.","0(1)","5.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,REPLICAOF_Hist
{"restore-asking","An internal command for migrating keys in a cluster","0(1)
to create the new key and additional O(N*M) to reconstruct the serialized
value, where N is the number of Redis objects composing the value and M their
average size. For small string values the time complexity is thus O(1)+O(1*M)
where M is small, so simply O(1). However for sorted set values the complexity
is O(N*M*log(N)) because inserting values into sorted sets is
O(log(N)).","3.0.0",CMD_DOC_SYSCMD,NULL,NULL,COMMAND_GROUP_SERVER,RESTORE_ASKING_H
{{NULL,CMD_KEY_OW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}}},
{"role","Return the role of the instance in the context of
replication","0(1)","2.8.12",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,ROLE_Hist
{"save","Synchronously save the dataset to disk","O(N) where N is the total
number of keys in all
databases","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SAVE_History,SAVE_t
{"shutdown","Synchronously save the dataset to disk and then shut down the
server","O(N) when saving, where N is the total number of keys in all databases
when saving data, otherwise
O(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SHUTDOWN_History,SHUTDOW
{"slaveof","Make the server a replica of another instance, or promote it as
master.","0(1)","1.0.0",CMD_DOC_DEPRECATED,"`REPLICAOF`","5.0.0",COMMAND_GROUP_SER
{"slowlog","A container for slow log commands","Depends on
subcommand.","2.2.12",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SLOWLOG_History,
{"swapdb","Swaps two Redis databases","O(N) where N is the count of clients
watching or blocking on keys from both
databases.","4.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SWAPDB_History,SWA
{"sync","Internal command used for
replication",NULL,"1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,SYNC_History
{"time","Return the current server
time","0(1)","2.6.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SERVER,TIME_History,TIME
/* set */
{"sadd","Add one or more members to a set","O(1) for each element added, so
O(N) to add N elements when the command is called with multiple
arguments.","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SET,SADD_History,SADD_tip
{{NULL,CMD_KEY_RW|CMD_KEY_INSERT,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=SADD_Args},
{"scard","Get the number of members in a
set","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_SET,SCARD_History,SCARD_t
{{NULL,CMD_KEY_R0,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{0,1,0}}},.args=SCARD_Args},
{"sdiff","Subtract multiple sets","O(N) where N is the total number of elements
in all given

```

```

sets.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SDIFF_History, SDIFF_tips, s
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=SDIFF_Args},
{"sdiffstore", "Subtract multiple sets and store the resulting set in a
key", "O(N) where N is the total number of elements in all given
sets.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SDIFFSTORE_History, SDIFFST
{{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=SDIFFSTORE_Args},
{"sinter", "Intersect multiple sets", "O(N*M) worst case where N is the
cardinality of the smallest set and M is the number of
sets.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SINTER_History, SINTER_tips
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=SINTER_Args},
{"sintercard", "Intersect multiple sets and return the cardinality of the
result", "O(N*M) worst case where N is the cardinality of the smallest set and M
is the number of
sets.", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SINTERCARD_History, SINTERC
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_KEYNUM, .fk.keynum=
{0,1,1}}}, sintercardGetKeys, .args=SINTERCARD_Args},
{"sinterstore", "Intersect multiple sets and store the resulting set in a
key", "O(N*M) worst case where N is the cardinality of the smallest set and M is
the number of
sets.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SINTERSTORE_History, SINTER
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=SINTERSTORE_Args},
{"sismember", "Determine if a given value is a member of a
set", "O(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SISMEMBER_History, SIS
{{NULL, CMD_KEY_RO, KSPEC_BS_INDEX, .bs.index={1}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, .args=SISMEMBER_Args},
{"smembers", "Get all the members in a set", "O(N) where N is the set
cardinality.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SMEMBERS_History, SM
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SMEMBERS_Args},
{"smismember", "Returns the membership associated with the given elements for a
set", "O(N) where N is the number of elements being checked for
membership", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SMISMEMBER_History, SM
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SMISMEMBER_Args},
{"smove", "Move a member from one set to
another", "O(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SMOVE_History, SMO
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SMOVE_Args},
{"spop", "Remove and return one or multiple random members from a set", "Without
the count argument O(1), otherwise O(N) where N is the value of the passed

```

```

count.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SPOP_History, SPOP_tips, sp
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SPOP_Args},
{"srandmember", "Get one or multiple random members from a set", "Without the
count argument O(1), otherwise O(N) where N is the absolute value of the passed
count.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SRANDMEMBER_History, SRAND
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SRANDMEMBER_Args},
{"srem", "Remove one or more members from a set", "O(N) where N is the number of
members to be
removed.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SREM_History, SREM_tips,
{{NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SREM_Args},
{"sscan", "Incrementally iterate Set elements", "O(1) for every call. O(N) for a
complete iteration, including enough command calls for the cursor to return
back to 0. N is the number of elements inside the
collection..", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SSCAN_History, SSCAN
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SSCAN_Args},
{"sunion", "Add multiple sets", "O(N) where N is the total number of elements in
all given
sets.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SUNION_History, SUNION_tips
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=SUNION_Args},
{"sunionstore", "Add multiple sets and store the resulting set in a key", "O(N)
where N is the total number of elements in all given
sets.", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SET, SUNIONSTORE_History, SUNION
{{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={-1,1,0}}}, .args=SUNIONSTORE_Args},
/* sorted_set */
{"bzmpop", "Remove and return members with scores in a sorted set or block until
one is available", "O(K) + O(N*log(M)) where K is the number of provided keys, N
being the number of elements in the sorted set, and M being the number of
elements
popped.", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, BZMPOP_History, BZ
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_KEYNUM, .fk.keynum={0,1,1}}}, blmpopGetKeys, .args=BZMPOP_Args},
{"bzpopmax", "Remove and return the member with the highest score from one or
more sorted sets, or block until one is available", "O(log(N)) with N being the
number of elements in the sorted
set.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, BZPOP_MAX_History, BZP
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-2,1,0}}}, .args=BZPOP_MAX_Args},
{"bzpopmin", "Remove and return the member with the lowest score from one or
more sorted sets, or block until one is available", "O(log(N)) with N being the
number of elements in the sorted
set.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, BZPOP_MIN_History, BZP
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={-2,1,0}}}, .args=BZPOP_MIN_Args},

```



```

{"zadd","Add one or more members to a sorted set, or update its score if it
already exists"," $O(\log(N))$  for each item added, where N is the number of
elements in the sorted
set.", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZADD_History, ZADD_ti
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZADD_Args},
{"zcard","Get the number of members in a sorted
set", "0(1)", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZCARD_History,
{{NULL, CMD_KEY_R0, KSPEC_BS_INDEX, .bs.index={1}, KSPEC_FK_RANGE, .fk.range=
{0,1,0}}}, .args=ZCARD_Args},
{"zcount","Count the members in a sorted set with scores within the given
values", "0( $\log(N)$ ) with N being the number of elements in the sorted
set.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZCOUNT_History, ZCOUN
{{NULL, CMD_KEY_R0|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZCOUNT_Args},
{"zdiff","Subtract multiple sorted sets", "0( $L + (N-K)\log(N)$ ) worst case where L
is the total number of elements in all the sets, N is the size of the first
set, and K is the size of the result
set.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZDIFF_History, ZDIFF_
{{NULL, CMD_KEY_R0|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_KEYNUM, .fk.keynum=
{0,1,1}}}, zunionInterDiffGetKeys, .args=ZDIFF_Args},
{"zdiffstore","Subtract multiple sorted sets and store the resulting sorted set
in a new key", "0( $L + (N-K)\log(N)$ ) worst case where L is the total number of
elements in all the sets, N is the size of the first set, and K is the size of
the result
set.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZDIFFSTORE_History, Z
{{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_R0|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_KEYNUM, .fk.keynum=
{0,1,1}}}, zunionInterDiffStoreGetKeys, .args=ZDIFFSTORE_Args},
{"zincrby","Increment the score of a member in a sorted set", "0( $\log(N)$ ) where N
is the number of elements in the sorted
set.", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZINCRBY_History, ZINC
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZINCRBY_Args},
{"zinter","Intersect multiple sorted sets", "0( $N*K$ )+0( $M*\log(M)$ ) worst case with
N being the smallest input sorted set, K being the number of input sorted sets
and M being the number of elements in the resulting sorted
set.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZINTER_History, ZINTE
{{NULL, CMD_KEY_R0|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_KEYNUM, .fk.keynum=
{0,1,1}}}, zunionInterDiffGetKeys, .args=ZINTER_Args},
{"zintercard","Intersect multiple sorted sets and return the cardinality of the
result", "0( $N*K$ ) worst case with N being the smallest input sorted set, K being
the number of input sorted
sets.", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZINTERCARD_History,
{{NULL, CMD_KEY_R0|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_KEYNUM, .fk.keynum=
{0,1,1}}}, zunionInterDiffGetKeys, .args=ZINTERCARD_Args},

```

```

{"zinterstore","Intersect multiple sorted sets and store the resulting sorted
set in a new key"," $O(N*K)+O(M*\log(M))$  worst case with N being the smallest
input sorted set, K being the number of input sorted sets and M being the
number of elements in the resulting sorted
set.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZINTERSTORE_History,
{{NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}},
{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_KEYNUM, .fk.keynum=
{0,1,1}}}, zunionInterDiffStoreGetKeys, .args=ZINTERSTORE_Args},
{"zlexcount","Count the number of members in a sorted set between a given
lexicographical range"," $O(\log(N))$  with N being the number of elements in the
sorted
set.", "2.8.9", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZLEXCOUNT_History, ZL
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZLEXCOUNT_Args},
{"zmpop","Remove and return members with scores in a sorted set"," $O(K) +
O(N*\log(M))$  where K is the number of provided keys, N being the number of
elements in the sorted set, and M being the number of elements
popped.", "7.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZMPOP_History, ZMP
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_KEYNUM, .fk.keynum={0,1,1}}}, zmpopGetKeys, .args=ZMPOP_Args},
{"zmscore","Get the score associated with the given members in a sorted
set"," $O(N)$  where N is the number of members being
requested.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZMSCORE_Histor
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZMSCORE_Args},
{"zpopmax","Remove and return members with the highest scores in a sorted
set"," $O(\log(N)*M)$  with N being the number of elements in the sorted set, and M
being the number of elements
popped.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZPOPMAX_History, Z
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZPOPMAX_Args},
{"zpopmin","Remove and return members with the lowest scores in a sorted
set"," $O(\log(N)*M)$  with N being the number of elements in the sorted set, and M
being the number of elements
popped.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZPOPMIN_History, Z
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZPOPMIN_Args},
{"zrandmember","Get one or multiple random elements from a sorted set"," $O(N)$ 
where N is the number of elements
returned", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZRANDMEMBER_Hist
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZRANDMEMBER_Args},
{"zrange","Return a range of members in a sorted set"," $O(\log(N)+M)$  with N being
the number of elements in the sorted set and M the number of elements
returned.", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZRANGE_History,
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZRANGE_Args},
{"zrangebylex","Return a range of members in a sorted set, by lexicographical
range"," $O(\log(N)+M)$  with N being the number of elements in the sorted set and M

```

the number of elements being returned. If M is constant (e.g. always asking for the first 10 elements with LIMIT), you can consider it $O(\log(N))$.", "2.8.9", CMD_DOC_DEPRECATED, "`ZRANGE` with the `BYLEX` argument", "6.2.0", COMMAND_GROUP_SORTED_SET, ZRANGEBYLEX_History, ZRANGEBYLEX_tips, zr

```
{ {NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZRANGEBYLEX_Args},
{"zrangebyscore", "Return a range of members in a sorted set, by
score", "O(log(N)+M) with N being the number of elements in the sorted set and M
the number of elements being returned. If M is constant (e.g. always asking for
the first 10 elements with LIMIT), you can consider it
O(log(N)).", "1.0.5", CMD_DOC_DEPRECATED, "`ZRANGE` with the `BYScore`
argument", "6.2.0", COMMAND_GROUP_SORTED_SET, ZRANGEBYScore_History, ZRANGEBYScore_tip
{ {NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZRANGEBYScore_Args},
{"zrangestore", "Store a range of members from sorted set into another
key", "O(log(N)+M) with N being the number of elements in the sorted set and M
the number of elements stored into the destination
key.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZRANGESTORE_History,
{ {NULL, CMD_KEY_OW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},
{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{2}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZRANGESTORE_Args},
{"zrank", "Determine the index of a member in a sorted
set", "O(log(N))", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZRANK_His
{ {NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZRANK_Args},
{"zrem", "Remove one or more members from a sorted set", "O(M*log(N)) with N
being the number of elements in the sorted set and M the number of elements to
be
removed.", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZREM_History, ZRE
{ {NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZREM_Args},
{"zremrangebylex", "Remove all members in a sorted set between the given
lexicographical range", "O(log(N)+M) with N being the number of elements in the
sorted set and M the number of elements removed by the
operation.", "2.8.9", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZREMRANGEBYLEX
{ {NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZREMRANGEBYLEX_Args},
{"zremrangebyrank", "Remove all members in a sorted set within the given
indexes", "O(log(N)+M) with N being the number of elements in the sorted set and
M the number of elements removed by the
operation.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZREMRANGEBYRAN
{ {NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZREMRANGEBYRANK_Args},
{"zremrangebyscore", "Remove all members in a sorted set within the given
scores", "O(log(N)+M) with N being the number of elements in the sorted set and
M the number of elements removed by the
operation.", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZREMRANGEBYSCO
{ {NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=ZREMRANGEBYSCORE_Args},
{"zrevrange", "Return a range of members in a sorted set, by index, with scores
```

ordered from high to low", "O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements returned.", "1.2.0", CMD_DOC_DEPRECATED, "`ZRANGE` with the `REV` argument", "6.2.0", COMMAND_GROUP_SORTED_SET, ZREVRANGE_History, ZREVRANGE_tips, zrevrange

```
{ {NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}} }, .args=ZREVRANGE_Args},
{"zrevrangebylex", "Return a range of members in a sorted set, by
lexicographical range, ordered from higher to lower strings.", "O(log(N)+M) with
N being the number of elements in the sorted set and M the number of elements
being returned. If M is constant (e.g. always asking for the first 10 elements
with LIMIT), you can consider it
O(log(N)).", "2.8.9", CMD_DOC_DEPRECATED, "`ZRANGE` with the `REV` and `BYLEX`
arguments", "6.2.0", COMMAND_GROUP_SORTED_SET, ZREVRANGEBYLEX_History, ZREVRANGEBYLEX_
{ {NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}} }, .args=ZREVRANGEBYLEX_Args},
{"zrevrangebyscore", "Return a range of members in a sorted set, by score, with
scores ordered from high to low", "O(log(N)+M) with N being the number of
elements in the sorted set and M the number of elements being returned. If M is
constant (e.g. always asking for the first 10 elements with LIMIT), you can
consider it O(log(N)).", "2.2.0", CMD_DOC_DEPRECATED, "`ZRANGE` with the `REV` and
`BYScore`
arguments", "6.2.0", COMMAND_GROUP_SORTED_SET, ZREVRANGEBYSCORE_History, ZREVRANGEBYSC
{ {NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}} }, .args=ZREVRANGEBYSCORE_Args},
{"zrevrank", "Determine the index of a member in a sorted set, with scores
ordered from high to
low", "O(log(N))", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZREVRANK_
{ {NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}} }, .args=ZREVRANK_Args},
{"zscan", "Incrementally iterate sorted sets elements and associated
scores", "O(1) for every call. O(N) for a complete iteration, including enough
command calls for the cursor to return back to 0. N is the number of elements
inside the
collection..", "2.8.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZSCAN_Histor
{ {NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}} }, .args=ZSCAN_Args},
{"zscore", "Get the score associated with the given member in a sorted
set", "O(1)", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZSCORE_History
{ {NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}} }, .args=ZSCORE_Args},
{"zunion", "Add multiple sorted sets", "O(N)+O(M*log(M)) with N being the sum of
the sizes of the input sorted sets, and M being the number of elements in the
resulting sorted
set.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZUNION_History, ZUNIO
{ {NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_KEYNUM, .fk.keynum=
{0,1,1}} }, zunionInterDiffGetKeys, .args=ZUNION_Args},
{"zunionstore", "Add multiple sorted sets and store the resulting sorted set in
a new key", "O(N)+O(M log(M)) with N being the sum of the sizes of the input
sorted sets, and M being the number of elements in the resulting sorted
set.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_SORTED_SET, ZUNIONSTORE_History,
```

```

{{NULL,CMD_KEY_OW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}},
{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{2},KSPEC_FK_KEYNUM,.fk.keynum=
{0,1,1}}},zunionInterDiffStoreGetKeys,.args=ZUNIONSTORE_Args},
/* stream */
{"xack","Marks a pending message as correctly processed, effectively removing
it from the pending entries list of the consumer group. Return value of the
command is the number of messages successfully acknowledged, that is, the IDs
we were actually able to resolve in the PEL.", "0(1) for each message ID
processed.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XACK_History, XACK_
{{NULL,CMD_KEY_RW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=XACK_Args},
{"xadd","Appends a new entry to a stream", "0(1) when adding a new entry, 0(N)
when trimming where N being the number of entries
evicted.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XADD_History, XADD_ti
{"UPDATE instead of INSERT because of the optional trimming
feature", CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}},.args=XADD_Args},
{"xautoclaim","Changes (or acquires) ownership of messages in a consumer group,
as if the messages were delivered to the specified consumer.", "0(1) if COUNT is
small.", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XAUTOCLAIM_History, XAU
{{NULL,CMD_KEY_RW|CMD_KEY_DELETE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=XAUTOCLAIM_Args},
{"xclaim","Changes (or acquires) ownership of a message in a consumer group, as
if the message was delivered to the specified consumer.", "0(log N) with N being
the number of messages in the PEL of the consumer
group.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XCLAIM_History, XCLAIM_
{{NULL,CMD_KEY_RW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=XCLAIM_Args},
{"xdel","Removes the specified entries from the stream. Returns the number of
items actually deleted, that may be different from the number of IDs passed in
case certain IDs do not exist.", "0(1) for each single item to delete in the
stream, regardless of the stream
size.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XDEL_History, XDEL_tips,
{{NULL,CMD_KEY_RW|CMD_KEY_DELETE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=XDEL_Args},
{"xgroup","A container for consumer groups commands", "Depends on
subcommand.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XGROUP_History, XG
{"xinfo","A container for stream introspection commands", "Depends on
subcommand.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XINFO_History, XIN
{"xlen","Return the number of entries in a
stream", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XLEN_History, XL
{{NULL,CMD_KEY_RO,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{0,1,0}}},.args=XLEN_Args},
{"xpending","Return information and entries from a stream consumer group
pending entries list, that are messages fetched but never acknowledged.", "0(N)
with N being the number of elements returned, so asking for a small fixed
number of entries per call is 0(1). 0(M), where M is the total number of
entries scanned when used with the IDLE filter. When the command returns just
the summary and the list of consumers is small, it runs in 0(1) time;

```



```

otherwise, an additional  $O(N)$  time for iterating every
consumer.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XPENDING_History, XP
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XPENDING_Args},
{"xrange", "Return a range of elements in a stream, with IDs matching the
specified IDs interval", "0(N) with N being the number of elements being
returned. If N is constant (e.g. always asking for the first 10 elements with
COUNT), you can consider it
0(1).", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XRANGE_History, XRANGE_t
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XRANGE_Args},
{"xread", "Return never seen elements in multiple streams, with IDs greater than
the ones reported by the caller for each stream. Can block.", "For each stream
mentioned: 0(N) with N being the number of elements being returned, it means
that XREAD-ing with a fixed COUNT is 0(1). Note that when the BLOCK option is
used, XADD will pay 0(M) time in order to serve the M clients blocked on the
stream getting new
data.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XREAD_History, XREAD_tip
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_KEYWORD, .bs.keyword=
{"STREAMS", 1}, KSPEC_FK_RANGE, .fk.range=
{-1,1,2}}}, xreadGetKeys, .args=XREAD_Args},
{"xreadgroup", "Return new entries from a stream using a consumer group, or
access the history of the pending entries for a given consumer. Can
block.", "For each stream mentioned: 0(M) with M being the number of elements
returned. If M is constant (e.g. always asking for the first 10 elements with
COUNT), you can consider it 0(1). On the other side when XREADGROUP blocks,
XADD will pay the 0(N) time in order to serve the N clients blocked on the
stream getting new
data.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XREADGROUP_History, XREA
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_KEYWORD, .bs.keyword=
{"STREAMS", 4}, KSPEC_FK_RANGE, .fk.range=
{-1,1,2}}}, xreadGetKeys, .args=XREADGROUP_Args},
{"xrevrange", "Return a range of elements in a stream, with IDs matching the
specified IDs interval, in reverse order (from greater to smaller IDs) compared
to XRANGE", "0(N) with N being the number of elements returned. If N is constant
(e.g. always asking for the first 10 elements with COUNT), you can consider it
0(1).", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XREVRANGE_History, XREVR
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XREVRANGE_Args},
{"xsetid", "An internal command for replicating stream
values", "0(1)", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XSETID_History,
{{NULL, CMD_KEY_RW|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XSETID_Args},
{"xtrim", "Trims the stream to (approximately if '~' is passed) a certain
size", "0(N), with N being the number of evicted entries. Constant times are
very small however, since entries are organized in macro nodes containing
multiple entries that can be released with a single
deallocation.", "5.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STREAM, XTRIM_History, X
{{NULL, CMD_KEY_RW|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=XTRIM_Args},
/* string */

```

```

{"append","Append a value to a key","0(1). The amortized time complexity is
0(1) assuming the appended value is small and the already present value is of
any size, since the dynamic string library used by Redis will double the free
space available on every
reallocation.", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, APPEND_History,
{{NULL, CMD_KEY_RW|CMD_KEY_INSERT, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=APPEND_Args},
{"decr","Decrement the integer value of a key by
one","0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, DECR_History, DECR_
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=DECR_Args},
{"decrby","Decrement the integer value of a key by the given
number","0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, DECRBY_History,
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=DECRBY_Args},
{"get","Get the value of a
key","0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, GET_History, GET_ti
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GET_Args},
{"getdel","Get the value of a key and delete the
key","0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, GETDEL_History, GET
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_DELETE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GETDEL_Args},
{"getex","Get the value of a key and optionally set its
expiration","0(1)", "6.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, GETEX_Histo
{"RW and UPDATE because it changes the
TTL", CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GETEX_Args},
{"getrange","Get a substring of the string stored at a key", "0(N) where N is
the length of the returned string. The complexity is ultimately determined by
the returned length, but because creating a substring from an existing string
is very cheap, it can be considered 0(1) for small
strings.", "2.4.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, GETRANGE_History, GET
{{NULL, CMD_KEY_RO|CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GETRANGE_Args},
{"getset","Set the string value of a key and return its old
value","0(1)", "1.0.0", CMD_DOC_DEPRECATED, "`SET` with the `!GET`
argument", "6.2.0", COMMAND_GROUP_STRING, GETSET_History, GETSET_tips, getsetCommand, 3,
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=GETSET_Args},
{"incr","Increment the integer value of a key by
one","0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, INCR_History, INCR_
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=INCR_Args},
{"incrby","Increment the integer value of a key by the given
amount","0(1)", "1.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, INCRBY_History,
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=INCRBY_Args},
{"incrbyfloat","Increment the float value of a key by the given
amount","0(1)", "2.6.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_STRING, INCRBYFLOAT_His
{{NULL, CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE, KSPEC_BS_INDEX, .bs.index=

```

```

{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=INCRBYFLOAT_Args},
{"lcs","Find longest common substring","0(N*M) where N and M are the lengths of
s1 and s2,
respectively","7.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,LCS_History,LCS_
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={1,1,0}}},.args=LCS_Args},
{"mget","Get the values of all the given keys","0(N) where N is the number of
keys to
retrieve.","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,MGET_History,MGET_t
{{NULL,CMD_KEY_RO|CMD_KEY_ACCESS,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={-1,1,0}}},.args=MGET_Args},
{"mset","Set multiple keys to multiple values","0(N) where N is the number of
keys to
set.","1.0.1",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,MSET_History,MSET_tips,m
{{NULL,CMD_KEY_OW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={-1,2,0}}},.args=MSET_Args},
{"msetnx","Set multiple keys to multiple values, only if none of the keys
exist","0(N) where N is the number of keys to
set.","1.0.1",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,MSETNX_History,MSETNX_ti
{{NULL,CMD_KEY_OW|CMD_KEY_INSERT,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={-1,2,0}}},.args=MSETNX_Args},
{"psetex","Set the value and expiration in milliseconds of a
key","0(1)","2.6.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,PSETEX_History,PSE
{{NULL,CMD_KEY_OW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=PSETEX_Args},
{"set","Set the string value of a
key","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,SET_History,SET_ti
{{"RW and ACCESS due to the optional `GET`
argument",CMD_KEY_RW|CMD_KEY_ACCESS|CMD_KEY_UPDATE|CMD_KEY_VARIABLE_FLAGS,KSPEC_BS
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},setGetKeys,.args=SET_Args},
{"setex","Set the value and expiration of a
key","0(1)","2.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,SETEX_History,SETE
{{NULL,CMD_KEY_OW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=SETEX_Args},
{"setnx","Set the value of a key, only if the key does not
exist","0(1)","1.0.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,SETNX_History,SE
{{NULL,CMD_KEY_OW|CMD_KEY_INSERT,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=SETNX_Args},
{"setrange","Overwrite part of a string at key starting at the specified
offset","0(1), not counting the time taken to copy the new string in place.
Usually, this string is very small so the amortized complexity is O(1).
Otherwise, complexity is O(M) with M being the length of the value
argument.","2.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,SETRANGE_History,SE
{{NULL,CMD_KEY_RW|CMD_KEY_UPDATE,KSPEC_BS_INDEX,.bs.index=
{1},KSPEC_FK_RANGE,.fk.range={0,1,0}}},.args=SETRANGE_Args},
{"strlen","Get the length of the value stored in a
key","0(1)","2.2.0",CMD_DOC_NONE,NULL,NULL,COMMAND_GROUP_STRING,STRLEN_History,STR
{{NULL,CMD_KEY_RO,KSPEC_BS_INDEX,.bs.index={1},KSPEC_FK_RANGE,.fk.range=
{0,1,0}}},.args=STRLEN_Args},
{"substr","Get a substring of the string stored at a key","0(N) where N is the
length of the returned string. The complexity is ultimately determined by the

```



```

returned length, but because creating a substring from an existing string is
very cheap, it can be considered O(1) for small
strings.", "1.0.0", CMD_DOC_DEPRECATED, "`GETRANGE` ", "2.0.0", COMMAND_GROUP_STRING, SUB
{{NULL, CMD_KEY_RO | CMD_KEY_ACCESS, KSPEC_BS_INDEX, .bs.index=
{1}, KSPEC_FK_RANGE, .fk.range={0,1,0}}}, .args=SUBSTR_Args},
/* transactions */
{"discard", "Discard all commands issued after MULTI", "O(N), when N is the
number of queued
commands", "2.0.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_TRANSACTIONS, DISCARD_Histor
{"exec", "Execute all commands issued after MULTI", "Depends on commands in the
transaction", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_TRANSACTIONS, EXEC_Histor
{"multi", "Mark the start of a transaction
block", "O(1)", "1.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_TRANSACTIONS, MULTI_Hist
{"unwatch", "Forget about all watched
keys", "O(1)", "2.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_TRANSACTIONS, UNWATCH_His
{"watch", "Watch the given keys to determine execution of the MULTI/EXEC
block", "O(1) for every
key.", "2.2.0", CMD_DOC_NONE, NULL, NULL, COMMAND_GROUP_TRANSACTIONS, WATCH_History, WATC
{{NULL, 0, KSPEC_BS_INDEX, .bs.index={1}, KSPEC_FK_RANGE, .fk.range=
{-1,1,0}}}, .args=WATCH_Args},
{0}
};

```

/config.c

[to top](#)

```

/* Configuration file parsing and CONFIG GET/SET commands implementation.
 *
 * Copyright (c) 2009-2012, Salvatore Sanfilippo <antirez at gmail dot com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of Redis nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include "server.h"
#include "cluster.h"

#include <fcntl.h>
#include <sys/stat.h>
#include <glob.h>
#include <string.h>

/*-----
 * Config file name-value maps.
 *-----
 */

typedef struct deprecatedConfig {
    const char *name;
    const int argc_min;
    const int argc_max;
} deprecatedConfig;

configEnum maxmemory_policy_enum[] = {

```

```
    {"volatile-lru", MAXMEMORY_VOLATILE_LRU},
    {"volatile-lfu", MAXMEMORY_VOLATILE_LFU},
    {"volatile-random", MAXMEMORY_VOLATILE_RANDOM},
    {"volatile-ttl", MAXMEMORY_VOLATILE_TTL},
    {"allkeys-lru", MAXMEMORY_ALLKEYS_LRU},
    {"allkeys-lfu", MAXMEMORY_ALLKEYS_LFU},
    {"allkeys-random", MAXMEMORY_ALLKEYS_RANDOM},
    {"noeviction", MAXMEMORY_NO_EVICTION},
    {NULL, 0}
};
```

```
configEnum syslog_facility_enum[] = {
    {"user", LOG_USER},
    {"local0", LOG_LOCAL0},
    {"local1", LOG_LOCAL1},
    {"local2", LOG_LOCAL2},
    {"local3", LOG_LOCAL3},
    {"local4", LOG_LOCAL4},
    {"local5", LOG_LOCAL5},
    {"local6", LOG_LOCAL6},
    {"local7", LOG_LOCAL7},
    {NULL, 0}
};
```

```
configEnum loglevel_enum[] = {
    {"debug", LL_DEBUG},
    {"verbose", LL_VERBOSE},
    {"notice", LL_NOTICE},
    {"warning", LL_WARNING},
    {NULL, 0}
};
```

```
configEnum supervised_mode_enum[] = {
    {"upstart", SUPERVISED_UPSTART},
    {"systemd", SUPERVISED_SYSTEMD},
    {"auto", SUPERVISED_AUTODETECT},
    {"no", SUPERVISED_NONE},
    {NULL, 0}
};
```

```
configEnum aof_fsync_enum[] = {
    {"everysec", AOF_FSYNC_EVERYSEC},
    {"always", AOF_FSYNC_ALWAYS},
    {"no", AOF_FSYNC_NO},
    {NULL, 0}
};
```

```
configEnum shutdown_on_sig_enum[] = {
    {"default", 0},
    {"save", SHUTDOWN_SAVE},
    {"nosave", SHUTDOWN_NOSAVE},
};
```

```

        {"now", SHUTDOWN_NOW},
        {"force", SHUTDOWN_FORCE},
        {NULL, 0}
};

configEnum repl_diskless_load_enum[] = {
    {"disabled", REPL_DISKLESS_LOAD_DISABLED},
    {"on-empty-db", REPL_DISKLESS_LOAD_WHEN_DB_EMPTY},
    {"swapdb", REPL_DISKLESS_LOAD_SWAPDB},
    {NULL, 0}
};

configEnum tls_auth_clients_enum[] = {
    {"no", TLS_CLIENT_AUTH_NO},
    {"yes", TLS_CLIENT_AUTH_YES},
    {"optional", TLS_CLIENT_AUTH_OPTIONAL},
    {NULL, 0}
};

configEnum oom_score_adj_enum[] = {
    {"no", OOM_SCORE_ADJ_NO},
    {"yes", OOM_SCORE_RELATIVE},
    {"relative", OOM_SCORE_RELATIVE},
    {"absolute", OOM_SCORE_ADJ_ABSOLUTE},
    {NULL, 0}
};

configEnum acl_pubsub_default_enum[] = {
    {"allchannels", SELECTOR_FLAG_ALLCHANNELS},
    {"resetchannels", 0},
    {NULL, 0}
};

configEnum sanitize_dump_payload_enum[] = {
    {"no", SANITIZE_DUMP_NO},
    {"yes", SANITIZE_DUMP_YES},
    {"clients", SANITIZE_DUMP_CLIENTS},
    {NULL, 0}
};

configEnum protected_action_enum[] = {
    {"no", PROTECTED_ACTION_ALLOWED_NO},
    {"yes", PROTECTED_ACTION_ALLOWED_YES},
    {"local", PROTECTED_ACTION_ALLOWED_LOCAL},
    {NULL, 0}
};

configEnum cluster_preferred_endpoint_type_enum[] = {
    {"ip", CLUSTER_ENDPOINT_TYPE_IP},
    {"hostname", CLUSTER_ENDPOINT_TYPE_HOSTNAME},
    {"unknown-endpoint", CLUSTER_ENDPOINT_TYPE_UNKNOWN_ENDPOINT},

```

```

    {NULL, 0}
};

configEnum propagation_error_behavior_enum[] = {
    {"ignore", PROPAGATION_ERR_BEHAVIOR_IGNORE},
    {"panic", PROPAGATION_ERR_BEHAVIOR_PANIC},
    {"panic-on-replicas", PROPAGATION_ERR_BEHAVIOR_PANIC_ON_REPLICAS},
    {NULL, 0}
};

/* Output buffer limits presets. */
clientBufferLimitsConfig clientBufferLimitsDefaults[CLIENT_TYPE_OBUF_COUNT] = {
    {0, 0, 0}, /* normal */
    {1024*1024*256, 1024*1024*64, 60}, /* slave */
    {1024*1024*32, 1024*1024*8, 60} /* pubsub */
};

/* OOM Score defaults */
int configOOMScoreAdjValuesDefaults[CONFIG_OOM_COUNT] = { 0, 200, 800 };

/* Generic config infrastructure function pointers
 * int is_valid_fn(val, err)
 *     Return 1 when val is valid, and 0 when invalid.
 *     Optionally set err to a static error string.
 */

/* Configuration values that require no special handling to set, get, load or
 * rewrite. */
typedef struct boolConfigData {
    int *config; /* The pointer to the server config this value is stored in */
    int default_value; /* The default value of the config on rewrite */
    int (*is_valid_fn)(int val, const char **err); /* Optional function to
check validity of new value (generic doc above) */
} boolConfigData;

typedef struct stringConfigData {
    char **config; /* Pointer to the server config this value is stored in. */
    const char *default_value; /* Default value of the config on rewrite. */
    int (*is_valid_fn)(char* val, const char **err); /* Optional function to
check validity of new value (generic doc above) */
    int convert_empty_to_null; /* Boolean indicating if empty strings should
be stored as a NULL value. */
} stringConfigData;

typedef struct sdsConfigData {
    sds *config; /* Pointer to the server config this value is stored in. */
    char *default_value; /* Default value of the config on rewrite. */
    int (*is_valid_fn)(sds val, const char **err); /* Optional function to
check validity of new value (generic doc above) */
    int convert_empty_to_null; /* Boolean indicating if empty SDS strings
should

```

```

                                be stored as a NULL value. */
} sdsConfigData;

typedef struct enumConfigData {
    int *config; /* The pointer to the server config this value is stored in */
    configEnum *enum_value; /* The underlying enum type this data represents */
    int default_value; /* The default value of the config on rewrite */
    int (*is_valid_fn)(int val, const char **err); /* Optional function to
check validity of new value (generic doc above) */
} enumConfigData;

typedef enum numericType {
    NUMERIC_TYPE_INT,
    NUMERIC_TYPE_UINT,
    NUMERIC_TYPE_LONG,
    NUMERIC_TYPE_ULONG,
    NUMERIC_TYPE_LONG_LONG,
    NUMERIC_TYPE_ULONG_LONG,
    NUMERIC_TYPE_SIZE_T,
    NUMERIC_TYPE_SSIZE_T,
    NUMERIC_TYPE_OFF_T,
    NUMERIC_TYPE_TIME_T,
} numericType;

typedef struct numericConfigData {
    union {
        int *i;
        unsigned int *ui;
        long *l;
        unsigned long *ul;
        long long *ll;
        unsigned long long *ull;
        size_t *st;
        ssize_t *sst;
        off_t *ot;
        time_t *tt;
    } config; /* The pointer to the numeric config this value is stored in */
    unsigned int flags;
    numericType numeric_type; /* An enum indicating the type of this value */
    long long lower_bound; /* The lower bound of this numeric value */
    long long upper_bound; /* The upper bound of this numeric value */
    long long default_value; /* The default value of the config on rewrite */
    int (*is_valid_fn)(long long val, const char **err); /* Optional function
to check validity of new value (generic doc above) */
} numericConfigData;

typedef union typeData {
    boolConfigData yesno;
    stringConfigData string;
    sdsConfigData sds;
    enumConfigData enumd;

```

```

    numericConfigData numeric;
} typeData;

typedef struct standardConfig standardConfig;

typedef int (*apply_fn)(const char **err);
typedef struct typeInterface {
    /* Called on server start, to init the server with default value */
    void (*init)(standardConfig *config);
    /* Called on server startup and CONFIG SET, returns 1 on success,
     * 2 meaning no actual change done, 0 on error and can set a verbose err
     * string */
    int (*set)(standardConfig *config, sds *argv, int argc, const char **err);
    /* Optional: called after `set()` to apply the config change. Used only in
     * the context of CONFIG SET. Returns 1 on success, 0 on failure.
     * Optionally set err to a static error string. */
    apply_fn apply;
    /* Called on CONFIG GET, returns sds to be used in reply */
    sds (*get)(standardConfig *config);
    /* Called on CONFIG REWRITE, required to rewrite the config state */
    void (*rewrite)(standardConfig *config, const char *name, struct
rewriteConfigState *state);
} typeInterface;

struct standardConfig {
    const char *name; /* The user visible name of this config */
    const char *alias; /* An alias that can also be used for this config */
    unsigned int flags; /* Flags for this specific config */
    typeInterface interface; /* The function pointers that define the type
interface */
    typeData data; /* The type specific data exposed used by the interface */
    configType type; /* The type of config this is. */
    void *privdata; /* privdata for this config, for module configs this is a
ModuleConfig struct */
};

dict *configs = NULL; /* Runtime config values */

/* Lookup a config by the provided sds string name, or return NULL
 * if the config does not exist */
static standardConfig *lookupConfig(sds name) {
    dictEntry *de = dictFind(configs, name);
    return de ? dictGetVal(de) : NULL;
}

/*-----
 * Enum access functions
 *-----
*/

/* Get enum value from name. If there is no match INT_MIN is returned. */

```

```

int configEnumGetValue(configEnum *ce, sds *argv, int argc, int bitflags) {
    if (argc == 0 || (!bitflags && argc != 1)) return INT_MIN;
    int values = 0;
    for (int i = 0; i < argc; i++) {
        int matched = 0;
        for (configEnum *ceItem = ce; ceItem->name != NULL; ceItem++) {
            if (!strcasecmp(argv[i],ceItem->name)) {
                values |= ceItem->val;
                matched = 1;
            }
        }
        if (!matched) return INT_MIN;
    }
    return values;
}

/* Get enum name/s from value. If no matches are found "unknown" is returned.
*/
static sds configEnumGetName(configEnum *ce, int values, int bitflags) {
    sds names = NULL;
    int matches = 0;
    for( ; ce->name != NULL; ce++) {
        if (values == ce->val) { /* Short path for perfect match */
            sdsfree(names);
            return sdsnew(ce->name);
        }
        if (bitflags && (values & ce->val)) {
            names = names ? sdscatfmt(names, " %s", ce->name) : sdsnew(ce->name);
            matches |= ce->val;
        }
    }
    if (!names || values != matches) {
        sdsfree(names);
        return sdsnew("unknown");
    }
    return names;
}

/* Used for INFO generation. */
const char *evictPolicyToString(void) {
    for (configEnum *ce = maxmemory_policy_enum; ce->name != NULL; ce++) {
        if (server.maxmemory_policy == ce->val)
            return ce->name;
    }
    serverPanic("unknown eviction policy");
}

/*-----
* Config file parsing
*-----

```



```

*/

int yesnotoi(char *s) {
    if (!strcasecmp(s,"yes")) return 1;
    else if (!strcasecmp(s,"no")) return 0;
    else return -1;
}

void appendServerSaveParams(time_t seconds, int changes) {
    server.saveparams = zrealloc(server.saveparams,sizeof(struct saveparam)*
(server.saveparamslen+1));
    server.saveparams[server.saveparamslen].seconds = seconds;
    server.saveparams[server.saveparamslen].changes = changes;
    server.saveparamslen++;
}

void resetServerSaveParams(void) {
    zfree(server.saveparams);
    server.saveparams = NULL;
    server.saveparamslen = 0;
}

void queueLoadModule(sds path, sds *argv, int argc) {
    int i;
    struct moduleLoadQueueEntry *loadmod;

    loadmod = zmalloc(sizeof(struct moduleLoadQueueEntry));
    loadmod->argv = argc ? zmalloc(sizeof(robj*)*argc) : NULL;
    loadmod->path = sdsnew(path);
    loadmod->argc = argc;
    for (i = 0; i < argc; i++) {
        loadmod->argv[i] = createRawStringObject(argv[i],sdslen(argv[i]));
    }
    listAddNodeTail(server.loadmodule_queue,loadmod);
}

/* Parse an array of `arg_len` sds strings, validate and populate
 * server.client_obuf_limits if valid.
 * Used in CONFIG SET and configuration file parsing. */
static int updateClientOutputBufferLimit(sds *args, int arg_len, const char
**err) {
    int j;
    int class;
    unsigned long long hard, soft;
    int hard_err, soft_err;
    int soft_seconds;
    char *soft_seconds_eptr;
    clientBufferLimitsConfig values[CLIENT_TYPE_OBUF_COUNT];
    int classes[CLIENT_TYPE_OBUF_COUNT] = {0};

    /* We need a multiple of 4: <class> <hard> <soft> <soft_seconds> */

```

```

if (arg_len % 4) {
    if (err) *err = "Wrong number of arguments in "
                  "buffer limit configuration.";
    return 0;
}

/* Sanity check of single arguments, so that we either refuse the
 * whole configuration string or accept it all, even if a single
 * error in a single client class is present. */
for (j = 0; j < arg_len; j += 4) {
    class = getClientTypeByName(args[j]);
    if (class == -1 || class == CLIENT_TYPE_MASTER) {
        if (err) *err = "Invalid client class specified in "
                      "buffer limit configuration.";
        return 0;
    }

    hard = memtoul(args[j+1], &hard_err);
    soft = memtoul(args[j+2], &soft_err);
    soft_seconds = strtoll(args[j+3], &soft_seconds_eptr, 10);
    if (hard_err || soft_err ||
        soft_seconds < 0 || *soft_seconds_eptr != '\0')
    {
        if (err) *err = "Error in hard, soft or soft_seconds setting in "
                      "buffer limit configuration.";
        return 0;
    }

    values[class].hard_limit_bytes = hard;
    values[class].soft_limit_bytes = soft;
    values[class].soft_limit_seconds = soft_seconds;
    classes[class] = 1;
}

/* Finally set the new config. */
for (j = 0; j < CLIENT_TYPE_OBUF_COUNT; j++) {
    if (classes[j]) server.client_obuf_limits[j] = values[j];
}

return 1;
}

/* Note this is here to support detecting we're running a config set from
 * within conf file parsing. This is only needed to support the deprecated
 * abnormal aggregate `save T C` functionality. Remove in the future. */
static int reading_config_file;

void loadServerConfigFromString(char *config) {
    deprecatedConfig deprecated_configs[] = {
        {"list-max-ziplist-entries", 2, 2},
        {"list-max-ziplist-value", 2, 2},
    }

```

```

        {"lua-replicate-commands", 2, 2},
        {NULL, 0},
    };
    char buf[1024];
    const char *err = NULL;
    int linenum = 0, totlines, i;
    sds *lines;

    reading_config_file = 1;
    lines = sdssplitlen(config, strlen(config), "\n", 1, &totlines);

    for (i = 0; i < totlines; i++) {
        sds *argv;
        int argc;

        linenum = i+1;
        lines[i] = sdstrim(lines[i], " \t\r\n");

        /* Skip comments and blank lines */
        if (lines[i][0] == '#' || lines[i][0] == '\0') continue;

        /* Split into arguments */
        argv = sdssplitargs(lines[i], &argc);
        if (argv == NULL) {
            err = "Unbalanced quotes in configuration line";
            goto loaderr;
        }

        /* Skip this line if the resulting command vector is empty. */
        if (argc == 0) {
            sdsfreesplitres(argv, argc);
            continue;
        }
        sdstolower(argv[0]);

        /* Iterate the configs that are standard */
        standardConfig *config = lookupConfig(argv[0]);
        if (config) {
            /* For normal single arg configs enforce we have a single argument.
             * Note that MULTI_ARG_CONFIGs need to validate arg count on their
            own */
            if (!(config->flags & MULTI_ARG_CONFIG) && argc != 2) {
                err = "wrong number of arguments";
                goto loaderr;
            }

            /* Set config using all arguments that follows */
            if (!config->interface.set(config, &argv[1], argc-1, &err)) {
                goto loaderr;
            }

            sdsfreesplitres(argv, argc);

```

```

        continue;
    } else {
        int match = 0;
        for (deprecatedConfig *config = deprecated_configs; config->name !=
NULL; config++) {
            if (!strcasecmp(argv[0], config->name) &&
                config->argc_min <= argc &&
                argc <= config->argc_max)
            {
                match = 1;
                break;
            }
        }
        if (match) {
            sdsfreesplitres(argv,argc);
            continue;
        }
    }

    /* Execute config directives */
    if (!strcasecmp(argv[0],"include") && argc == 2) {
        loadServerConfig(argv[1], 0, NULL);
    } else if (!strcasecmp(argv[0],"rename-command") && argc == 3) {
        struct redisCommand *cmd = lookupCommandBySds(argv[1]);
        int retval;

        if (!cmd) {
            err = "No such command in rename-command";
            goto loaderr;
        }

        /* If the target command name is the empty string we just
         * remove it from the command table. */
        retval = dictDelete(server.commands, argv[1]);
        serverAssert(retval == DICT_OK);

        /* Otherwise we re-add the command under a different name. */
        if (sdslen(argv[2]) != 0) {
            sds copy = sdsdup(argv[2]);

            retval = dictAdd(server.commands, copy, cmd);
            if (retval != DICT_OK) {
                sdsfree(copy);
                err = "Target command name already exists"; goto loaderr;
            }
        }
    } else if (!strcasecmp(argv[0],"user") && argc >= 2) {
        int argc_err;
        if (ACLAppendUserForLoading(argv,argc,&argc_err) == C_ERR) {
            const char *errmsg = ACLSetUserStringError();
            snprintf(buf,sizeof(buf),"Error in user declaration '%s': %s",

```

```

        argv[argc_err],errmsg);
        err = buf;
        goto loaderr;
    }
} else if (!strcasecmp(argv[0],"loadmodule") && argc >= 2) {
    queueLoadModule(argv[1],&argv[2],argc-2);
} else if (strchr(argv[0], '.')) {
    if (argc < 2) {
        err = "Module config specified without value";
        goto loaderr;
    }
    sds name = sdsdup(argv[0]);
    sds val = sdsdup(argv[1]);
    for (int i = 2; i < argc; i++)
        val = sdscatfmt(val, " %S", argv[i]);
    if (!dictReplace(server.module_configs_queue, name, val))
sdsfree(name);
} else if (!strcasecmp(argv[0],"sentinel")) {
    /* argc == 1 is handled by main() as we need to enter the sentinel
     * mode ASAP. */
    if (argc != 1) {
        if (!server.sentinel_mode) {
            err = "sentinel directive while not in sentinel mode";
            goto loaderr;
        }
        queueSentinelConfig(argv+1,argc-1,linenum,lines[i]);
    }
} else {
    err = "Bad directive or wrong number of arguments"; goto loaderr;
}
sdsfreesplitres(argv,argc);
}

if (server.logfile[0] != '\0') {
    FILE *logfp;

    /* Test if we are able to open the file. The server will not
     * be able to abort just for this problem later... */
    logfp = fopen(server.logfile,"a");
    if (logfp == NULL) {
        err = sdscatprintf(sdsempty(),
                           "Can't open the log file: %s", strerror(errno));
        goto loaderr;
    }
    fclose(logfp);
}

/* Sanity checks. */
if (server.cluster_enabled && server.masterhost) {
    err = "replicaof directive not allowed in cluster mode";
    goto loaderr;
}

```

```

}

/* To ensure backward compatibility and work while hz is out of range */
if (server.config_hz < CONFIG_MIN_HZ) server.config_hz = CONFIG_MIN_HZ;
if (server.config_hz > CONFIG_MAX_HZ) server.config_hz = CONFIG_MAX_HZ;

sdsfreesplitres(lines,totlines);
reading_config_file = 0;
return;

loaderr:
    fprintf(stderr, "\n*** FATAL CONFIG FILE ERROR (Redis %s) ***\n",
        REDIS_VERSION);
    if (i < totlines) {
        fprintf(stderr, "Reading the configuration file, at line %d\n",
linenum);
        fprintf(stderr, ">>> '%s'\n", lines[i]);
    }
    fprintf(stderr, "%s\n", err);
    exit(1);
}

/* Load the server configuration from the specified filename.
 * The function appends the additional configuration directives stored
 * in the 'options' string to the config file before loading.
 *
 * Both filename and options can be NULL, in such a case are considered
 * empty. This way loadServerConfig can be used to just load a file or
 * just load a string. */
#define CONFIG_READ_LEN 1024
void loadServerConfig(char *filename, char config_from_stdin, char *options) {
    sds config = sdsempty();
    char buf[CONFIG_READ_LEN+1];
    FILE *fp;
    glob_t globbuf;

    /* Load the file content */
    if (filename) {

        /* The logic for handling wildcards has slightly different behavior in
        cases where
            * there is a failure to locate the included file.
            * Whether or not a wildcard is specified, we should ALWAYS log errors
        when attempting
            * to open included config files.
            *
            * However, we desire a behavioral difference between instances where a
        wildcard was
            * specified and those where it hasn't:
            *     no wildcards : attempt to open the specified file and fail
        with a logged error

```

```

        *                               if the file cannot be found and opened.
        *       with wildcards : attempt to glob the specified pattern; if no
files match the
        *                               pattern, then gracefully continue on to the
next entry in the
        *                               config file, as if the current entry was never
encountered.
        *                               This will allow for empty conf.d directories
to be included. */

    if (strchr(filename, '*') || strchr(filename, '?') || strchr(filename,
[''])) {
        /* A wildcard character detected in filename, so let us use glob */
        if (glob(filename, 0, NULL, &globbuf) == 0) {

            for (size_t i = 0; i < globbuf.gl_pathc; i++) {
                if ((fp = fopen(globbuf.gl_pathv[i], "r")) == NULL) {
                    serverLog(LL_WARNING,
                        "Fatal error, can't open config file '%s':
%s",
                        globbuf.gl_pathv[i], strerror(errno));
                    exit(1);
                }
                while(fgets(buf, CONFIG_READ_LEN+1, fp) != NULL)
                    config = sdscat(config, buf);
                fclose(fp);
            }

            globfree(&globbuf);
        } else {
            /* No wildcard in filename means we can use the original logic to
read and
            * potentially fail traditionally */
            if ((fp = fopen(filename, "r")) == NULL) {
                serverLog(LL_WARNING,
                    "Fatal error, can't open config file '%s': %s",
                    filename, strerror(errno));
                exit(1);
            }
            while(fgets(buf, CONFIG_READ_LEN+1, fp) != NULL)
                config = sdscat(config, buf);
            fclose(fp);
        }
    }

    /* Append content from stdin */
    if (config_from_stdin) {
        serverLog(LL_WARNING, "Reading config from stdin");
        fp = stdin;
        while(fgets(buf, CONFIG_READ_LEN+1, fp) != NULL)

```

```

        config = sdscat(config,buf);
    }

    /* Append the additional options */
    if (options) {
        config = sdscat(config,"\n");
        config = sdscat(config,options);
    }
    loadServerConfigFromString(config);
    sdsfree(config);
}

static int performInterfaceSet(standardConfig *config, sds value, const char
**errstr) {
    sds *argv;
    int argc, res;

    if (config->flags & MULTI_ARG_CONFIG) {
        argv = sdssplitlen(value, sdslen(value), " ", 1, &argc);
    } else {
        argv = (char**)&value;
        argc = 1;
    }

    /* Set the config */
    res = config->interface.set(config, argv, argc, errstr);
    if (config->flags & MULTI_ARG_CONFIG) sdsfreesplitres(argv, argc);
    return res;
}

/* Find the config by name and attempt to set it to value. */
int performModuleConfigSetName(sds name, sds value, const char **err) {
    standardConfig *config = lookupConfig(name);
    if (!config || !(config->flags & MODULE_CONFIG)) {
        *err = "Config name not found";
        return 0;
    }
    return performInterfaceSet(config, value, err);
}

/* Find config by name and attempt to set it to its default value. */
int performModuleConfigSetDefaultFromName(sds name, const char **err) {
    standardConfig *config = lookupConfig(name);
    serverAssert(config);
    if (!(config->flags & MODULE_CONFIG)) {
        *err = "Config name not found";
        return 0;
    }
    switch (config->type) {
        case BOOL_CONFIG:
            return setModuleBoolConfig(config->privdata, config->

```



```

>data.yesno.default_value, err);
    case SDS_CONFIG:
        return setModuleStringConfig(config->privdata, config-
>data.sds.default_value, err);
    case NUMERIC_CONFIG:
        return setModuleNumericConfig(config->privdata, config-
>data.numeric.default_value, err);
    case ENUM_CONFIG:
        return setModuleEnumConfig(config->privdata, config-
>data.enumd.default_value, err);
    default:
        serverPanic("Config type of module config is not allowed.");
}
return 0;
}

static void restoreBackupConfig(standardConfig **set_configs, sds *old_values,
int count, apply_fn *apply_fns, list *module_configs) {
    int i;
    const char *errstr = "unknown error";
    /* Set all backup values */
    for (i = 0; i < count; i++) {
        if (!performInterfaceSet(set_configs[i], old_values[i], &errstr))
            serverLog(LL_WARNING, "Failed restoring failed CONFIG SET command.
Error setting %s to '%s': %s",
                set_configs[i]->name, old_values[i], errstr);
    }
    /* Apply backup */
    if (apply_fns) {
        for (i = 0; i < count && apply_fns[i] != NULL; i++) {
            if (!apply_fns[i](&errstr))
                serverLog(LL_WARNING, "Failed applying restored failed CONFIG
SET command: %s", errstr);
        }
    }
    if (module_configs) {
        if (!moduleConfigApplyConfig(module_configs, &errstr, NULL))
            serverLog(LL_WARNING, "Failed applying restored failed CONFIG SET
command: %s", errstr);
    }
}

/*-----
 * CONFIG SET implementation
 *-----
*/

void configSetCommand(client *c) {
    const char *errstr = NULL;
    const char *invalid_arg_name = NULL;
    const char *err_arg_name = NULL;

```

```

    standardConfig **set_configs; /* TODO: make this a dict for better
performance */
    list *module_configs_apply;
    const char **config_names;
    sds *new_values;
    sds *old_values = NULL;
    apply_fn *apply_fns; /* TODO: make this a set for better performance */
    int config_count, i, j;
    int invalid_args = 0, deny_loading_error = 0;
    int *config_map_fns;

    /* Make sure we have an even number of arguments: conf-val pairs */
    if (c->argc & 1) {
        addReplyErrorObject(c, shared.syntaxerr);
        return;
    }
    config_count = (c->argc - 2) / 2;

    module_configs_apply = listCreate();
    set_configs = zcalloc(sizeof(standardConfig)*config_count);
    config_names = zcalloc(sizeof(char*)*config_count);
    new_values = zmalloc(sizeof(sds)*config_count);
    old_values = zcalloc(sizeof(sds)*config_count);
    apply_fns = zcalloc(sizeof(apply_fn)*config_count);
    config_map_fns = zmalloc(sizeof(int)*config_count);

    /* Find all relevant configs */
    for (i = 0; i < config_count; i++) {
        standardConfig *config = lookupConfig(c->argv[2+i*2]->ptr);
        /* Fail if we couldn't find this config */
        if (!config) {
            if (!invalid_args) {
                invalid_arg_name = c->argv[2+i*2]->ptr;
                invalid_args = 1;
            }
            continue;
        }

        /* Note: it's important we run over ALL passed configs and check if we
need to call `redactClientCommandArgument()`.
        * This is in order to avoid anyone using this command for a
log/slowlog/monitor/etc. displaying sensitive info.
        * So even if we encounter an error we still continue running over the
remaining arguments. */
        if (config->flags & SENSITIVE_CONFIG) {
            redactClientCommandArgument(c,2+i*2+1);
        }

        /* We continue to make sure we redact all the configs */
        if (invalid_args) continue;
    }

```

```

        if (config->flags & IMMUTABLE_CONFIG ||
            (config->flags & PROTECTED_CONFIG &&
!allowProtectedAction(server.enable_protected_configs, c)))
        {
            /* Note: we don't abort the loop since we still want to handle
redacting sensitive configs (above) */
            errstr = (config->flags & IMMUTABLE_CONFIG) ? "can't set immutable
config" : "can't set protected config";
            err_arg_name = c->argv[2+i*2]->ptr;
            invalid_args = 1;
            continue;
        }

        if (server.loading && config->flags & DENY_LOADING_CONFIG) {
            /* Note: we don't abort the loop since we still want to handle
redacting sensitive configs (above) */
            deny_loading_error = 1;
            invalid_args = 1;
            continue;
        }

        /* If this config appears twice then fail */
        for (j = 0; j < i; j++) {
            if (set_configs[j] == config) {
                /* Note: we don't abort the loop since we still want to handle
redacting sensitive configs (above) */
                errstr = "duplicate parameter";
                err_arg_name = c->argv[2+i*2]->ptr;
                invalid_args = 1;
                break;
            }
        }
        set_configs[i] = config;
        config_names[i] = config->name;
        new_values[i] = c->argv[2+i*2+1]->ptr;
    }

    if (invalid_args) goto err;

    /* Backup old values before setting new ones */
    for (i = 0; i < config_count; i++)
        old_values[i] = set_configs[i]->interface.get(set_configs[i]);

    /* Set all new values (don't apply yet) */
    for (i = 0; i < config_count; i++) {
        int res = performInterfaceSet(set_configs[i], new_values[i], &errstr);
        if (!res) {
            restoreBackupConfig(set_configs, old_values, i+1, NULL, NULL);
            err_arg_name = set_configs[i]->name;
            goto err;
        } else if (res == 1) {

```

```

        /* A new value was set, if this config has an apply function then
store it for execution later */
        if (set_configs[i]->flags & MODULE_CONFIG) {
            addModuleConfigApply(module_configs_apply, set_configs[i]-
>privdata);
        } else if (set_configs[i]->interface.apply) {
            /* Check if this apply function is already stored */
            int exists = 0;
            for (j = 0; apply_fns[j] != NULL && j <= i; j++) {
                if (apply_fns[j] == set_configs[i]->interface.apply) {
                    exists = 1;
                    break;
                }
            }
            /* Apply function not stored, store it */
            if (!exists) {
                apply_fns[j] = set_configs[i]->interface.apply;
                config_map_fns[j] = i;
            }
        }
    }
}

/* Apply all configs after being set */
for (i = 0; i < config_count && apply_fns[i] != NULL; i++) {
    if (!apply_fns[i](&errstr)) {
        serverLog(LL_WARNING, "Failed applying new configuration. Possibly
related to new %s setting. Restoring previous settings.",
set_configs[config_map_fns[i]]->name);
        restoreBackupConfig(set_configs, old_values, config_count,
apply_fns, NULL);
        err_arg_name = set_configs[config_map_fns[i]]->name;
        goto err;
    }
}

/* Apply all module configs that were set. */
if (!moduleConfigApplyConfig(module_configs_apply, &errstr, &err_arg_name))
{
    serverLogRaw(LL_WARNING, "Failed applying new module configuration.
Restoring previous settings.");
    restoreBackupConfig(set_configs, old_values, config_count, apply_fns,
module_configs_apply);
    goto err;
}

RedisModuleConfigChangeV1 cc = {.num_changes = config_count, .config_names
= config_names};
moduleFireServerEvent(REDISMODULE_EVENT_CONFIG,
REDISMODULE_SUBEVENT_CONFIG_CHANGE, &cc);
addReply(c,shared.ok);
goto end;

```

```

err:
    if (deny_loading_error) {
        /* We give the loading error precedence because it may be handled by
clients differently, unlike a plain -ERR. */
        addReplyErrorObject(c,shared.loadingerr);
    } else if (invalid_arg_name) {
        addReplyErrorFormat(c,"Unknown option or number of arguments for CONFIG
SET - '%s'", invalid_arg_name);
    } else if (errstr) {
        addReplyErrorFormat(c,"CONFIG SET failed (possibly related to argument
'%s') - %s", err_arg_name, errstr);
    } else {
        addReplyErrorFormat(c,"CONFIG SET failed (possibly related to argument
'%s')", err_arg_name);
    }
end:
    zfree(set_configs);
    zfree(config_names);
    zfree(new_values);
    for (i = 0; i < config_count; i++)
        sdsfree(old_values[i]);
    zfree(old_values);
    zfree(apply_fns);
    zfree(config_map_fns);
    listRelease(module_configs_apply);
}

/*-----
 * CONFIG GET implementation
 *-----
*/

void configGetCommand(client *c) {
    int i;
    dictEntry *de;
    dictIterator *di;
    /* Create a dictionary to store the matched configs */
    dict *matches = dictCreate(&externalStringType);
    for (i = 0; i < c->argc - 2; i++) {
        robj *o = c->argv[2+i];
        sds name = o->ptr;

        /* If the string doesn't contain glob patterns, just directly
 * look up the key in the dictionary. */
        if (!strpbrk(name, "[*?")) {
            if (dictFind(matches, name)) continue;
            standardConfig *config = lookupConfig(name);

            if (config) {
                dictAdd(matches, name, config);
            }
        }
    }
}

```

```

    }
    continue;
}

/* Otherwise, do a match against all items in the dictionary. */
di = dictGetIterator(configs);

while ((de = dictNext(di)) != NULL) {
    standardConfig *config = dictGetVal(de);
    /* Note that hidden configs require an exact match (not a pattern)
*/
    if (config->flags & HIDDEN_CONFIG) continue;
    if (dictFind(matches, config->name)) continue;
    if (stringmatch(name, de->key, 1)) {
        dictAdd(matches, de->key, config);
    }
}
dictReleaseIterator(di);
}

di = dictGetIterator(matches);
addReplyMapLen(c, dictSize(matches));
while ((de = dictNext(di)) != NULL) {
    standardConfig *config = (standardConfig *) dictGetVal(de);
    addReplyBulkCString(c, de->key);
    addReplyBulkSds(c, config->interface.get(config));
}
dictReleaseIterator(di);
dictRelease(matches);
}

/*-----
 * CONFIG REWRITE implementation
 *-----
*/

#define REDIS_CONFIG_REWRITE_SIGNATURE "# Generated by CONFIG REWRITE"

/* We use the following dictionary type to store where a configuration
 * option is mentioned in the old configuration file, so it's
 * like "maxmemory" -> list of line numbers (first line is zero). */
void dictListDestructor(dict *d, void *val);

/* Sentinel config rewriting is implemented inside sentinel.c by
 * rewriteConfigSentinelOption(). */
void rewriteConfigSentinelOption(struct rewriteConfigState *state);

dictType optionToLineDictType = {
    dictSdsCaseHash,          /* hash function */
    NULL,                    /* key dup */
    NULL,                    /* val dup */

```

```

    dictSdsKeyCaseCompare,      /* key compare */
    dictSdsDestructor,          /* key destructor */
    dictListDestructor,         /* val destructor */
    NULL                        /* allow to expand */
};

dictType optionSetDictType = {
    dictSdsCaseHash,           /* hash function */
    NULL,                      /* key dup */
    NULL,                      /* val dup */
    dictSdsKeyCaseCompare,     /* key compare */
    dictSdsDestructor,         /* key destructor */
    NULL,                      /* val destructor */
    NULL                       /* allow to expand */
};

/* The config rewrite state. */
struct rewriteConfigState {
    dict *option_to_line; /* Option -> list of config file lines map */
    dict *rewritten;      /* Dictionary of already processed options */
    int numlines;         /* Number of lines in current config */
    sds *lines;           /* Current lines as an array of sds strings */
    int needs_signature;  /* True if we need to append the rewrite
                           signature. */
    int force_write;      /* True if we want all keywords to be force
                           written. Currently only used for testing
                           and debug information. */
};

/* Free the configuration rewrite state. */
void rewriteConfigReleaseState(struct rewriteConfigState *state) {
    sdsfreesplitres(state->lines, state->numlines);
    dictRelease(state->option_to_line);
    dictRelease(state->rewritten);
    zfree(state);
}

/* Create the configuration rewrite state */
struct rewriteConfigState *rewriteConfigCreateState() {
    struct rewriteConfigState *state = zmalloc(sizeof(*state));
    state->option_to_line = dictCreate(&optionToLineDictType);
    state->rewritten = dictCreate(&optionSetDictType);
    state->numlines = 0;
    state->lines = NULL;
    state->needs_signature = 1;
    state->force_write = 0;
    return state;
}

/* Append the new line to the current configuration state. */
void rewriteConfigAppendLine(struct rewriteConfigState *state, sds line) {

```

```

    state->lines = zrealloc(state->lines, sizeof(char*) * (state->numlines+1));
    state->lines[state->numlines++] = line;
}

/* Populate the option -> list of line numbers map. */
void rewriteConfigAddLineNumberToOption(struct rewriteConfigState *state, sds
option, int linenum) {
    list *l = dictFetchValue(state->option_to_line,option);

    if (l == NULL) {
        l = listCreate();
        dictAdd(state->option_to_line,sdsdup(option),l);
    }
    listAddNodeTail(l,(void*)(long)linenum);
}

/* Add the specified option to the set of processed options.
 * This is useful as only unused lines of processed options will be blanked
 * in the config file, while options the rewrite process does not understand
 * remain untouched. */
void rewriteConfigMarkAsProcessed(struct rewriteConfigState *state, const char
*option) {
    sds opt = sdsnew(option);

    if (dictAdd(state->rewritten,opt,NULL) != DICT_OK) sdsfree(opt);
}

/* Read the old file, split it into lines to populate a newly created
 * config rewrite state, and return it to the caller.
 *
 * If it is impossible to read the old file, NULL is returned.
 * If the old file does not exist at all, an empty state is returned. */
struct rewriteConfigState *rewriteConfigReadOldFile(char *path) {
    FILE *fp = fopen(path,"r");
    if (fp == NULL && errno != ENOENT) return NULL;

    struct redis_stat sb;
    if (fp && redis_fstat(fileno(fp),&sb) == -1) return NULL;

    int linenum = -1;
    struct rewriteConfigState *state = rewriteConfigCreateState();

    if (fp == NULL || sb.st_size == 0) return state;

    /* Load the file content */
    sds config = sdsnewlen(SDS_NOINIT,sb.st_size);
    if (fread(config,1,sb.st_size,fp) == 0) {
        sdsfree(config);
        rewriteConfigReleaseState(state);
        fclose(fp);
        return NULL;
    }

```



```

}

int i, totlines;
sds *lines = sdssplitlen(config,sdslen(config),"\n",1,&totlines);

/* Read the old content line by line, populate the state. */
for (i = 0; i < totlines; i++) {
    int argc;
    sds *argv;
    sds line = sdstrim(lines[i],"\r\n\t ");
    lines[i] = NULL;

    linenum++; /* Zero based, so we init at -1 */

    /* Handle comments and empty lines. */
    if (line[0] == '#' || line[0] == '\0') {
        if (state->needs_signature &&
!strcmp(line,REDIS_CONFIG_REWRITE_SIGNATURE))
            state->needs_signature = 0;
        rewriteConfigAppendLine(state,line);
        continue;
    }

    /* Not a comment, split into arguments. */
    argv = sdssplitargs(line,&argc);
    if (argv == NULL || (!server.sentinel_mode && !lookupConfig(argv[0])))
    {
        /* Apparently the line is unparsable for some reason, for
        * instance it may have unbalanced quotes, or may contain a
        * config that doesn't exist anymore. Load it as a comment. */
        sds aux = sdsnew("# ??? ");
        aux = sdscatsds(aux,line);
        if (argv) sdsfreesplitres(argv, argc);
        sdsfree(line);
        rewriteConfigAppendLine(state,aux);
        continue;
    }

    sdstolower(argv[0]); /* We only want lowercase config directives. */

    /* Now we populate the state according to the content of this line.
    * Append the line and populate the option -> line numbers map. */
    rewriteConfigAppendLine(state,line);

    /* Translate options using the word "slave" to the corresponding name
    * "replica", before adding such option to the config name -> lines
    * mapping. */
    char *p = strstr(argv[0],"slave");
    if (p) {
        sds alt = sdsempty();
        alt = sdscatlen(alt,argv[0],p-argv[0]);

```

```

        alt = sdscatlen(alt,"replica",7);
        alt = sdscatlen(alt,p+5,strlen(p+5));
        sdsfree(argv[0]);
        argv[0] = alt;
    }
    /* If this is sentinel config, we use sentinel "sentinel <config>" as
option
        to avoid messing up the sequence. */
    if (server.sentinel_mode && argc > 1 &&
!strcasecmp(argv[0],"sentinel")) {
        sds sentinelOption = sdsempty();
        sentinelOption = sdscatfmt(sentinelOption,"%S %S",argv[0],argv[1]);
        rewriteConfigAddLineNumberToOption(state,sentinelOption,linenum);
        sdsfree(sentinelOption);
    } else {
        rewriteConfigAddLineNumberToOption(state,argv[0],linenum);
    }
    sdsfreesplitres(argv,argc);
}
fclose(fp);
sdsfreesplitres(lines,totlines);
sdsfree(config);
return state;
}

/* Rewrite the specified configuration option with the new "line".
 * It progressively uses lines of the file that were already used for the same
 * configuration option in the old version of the file, removing that line from
 * the map of options -> line numbers.
 *
 * If there are lines associated with a given configuration option and
 * "force" is non-zero, the line is appended to the configuration file.
 * Usually "force" is true when an option has not its default value, so it
 * must be rewritten even if not present previously.
 *
 * The first time a line is appended into a configuration file, a comment
 * is added to show that starting from that point the config file was generated
 * by CONFIG REWRITE.
 *
 * "line" is either used, or freed, so the caller does not need to free it
 * in any way. */
void rewriteConfigRewriteLine(struct rewriteConfigState *state, const char
*option, sds line, int force) {
    sds o = sdsnew(option);
    list *l = dictFetchValue(state->option_to_line,o);

    rewriteConfigMarkAsProcessed(state,option);

    if (!l && !force && !state->force_write) {
        /* Option not used previously, and we are not forced to use it. */
        sdsfree(line);
    }

```

```

        sdsfree(o);
        return;
    }

    if (l) {
        listNode *ln = listFirst(l);
        int linenum = (long) ln->value;

        /* There are still lines in the old configuration file we can reuse
         * for this option. Replace the line with the new one. */
        listDelNode(l,ln);
        if (listLength(l) == 0) dictDelete(state->option_to_line,o);
        sdsfree(state->lines[linenum]);
        state->lines[linenum] = line;
    } else {
        /* Append a new line. */
        if (state->needs_signature) {
            rewriteConfigAppendLine(state,
                sdsnew(Redis_CONFIG_REWRITE_SIGNATURE));
            state->needs_signature = 0;
        }
        rewriteConfigAppendLine(state,line);
    }
    sdsfree(o);
}

/* Write the long long 'bytes' value as a string in a way that is parsable
 * inside redis.conf. If possible uses the GB, MB, KB notation. */
int rewriteConfigFormatMemory(char *buf, size_t len, long long bytes) {
    int gb = 1024*1024*1024;
    int mb = 1024*1024;
    int kb = 1024;

    if (bytes && (bytes % gb) == 0) {
        return snprintf(buf,len,"%lldgb",bytes/gb);
    } else if (bytes && (bytes % mb) == 0) {
        return snprintf(buf,len,"%lldmb",bytes/mb);
    } else if (bytes && (bytes % kb) == 0) {
        return snprintf(buf,len,"%lldkb",bytes/kb);
    } else {
        return snprintf(buf,len,"%lld",bytes);
    }
}

/* Rewrite a simple "option-name <bytes>" configuration option. */
void rewriteConfigBytesOption(struct rewriteConfigState *state, const char
*option, long long value, long long defvalue) {
    char buf[64];
    int force = value != defvalue;
    sds line;

```

```

rewriteConfigFormatMemory(buf,sizeof(buf),value);
line = sdscatprintf(sdsempty(),"%s %s",option,buf);
rewriteConfigRewriteLine(state,option,line,force);
}

/* Rewrite a simple "option-name n%" configuration option. */
void rewriteConfigPercentOption(struct rewriteConfigState *state, const char
*option, long long value, long long defvalue) {
    int force = value != defvalue;
    sds line = sdscatprintf(sdsempty(),"%s %lld%",option,value);

    rewriteConfigRewriteLine(state,option,line,force);
}

/* Rewrite a yes/no option. */
void rewriteConfigYesNoOption(struct rewriteConfigState *state, const char
*option, int value, int defvalue) {
    int force = value != defvalue;
    sds line = sdscatprintf(sdsempty(),"%s %s",option,
        value ? "yes" : "no");

    rewriteConfigRewriteLine(state,option,line,force);
}

/* Rewrite a string option. */
void rewriteConfigStringOption(struct rewriteConfigState *state, const char
*option, char *value, const char *defvalue) {
    int force = 1;
    sds line;

    /* String options set to NULL need to be not present at all in the
     * configuration file to be set to NULL again at the next reboot. */
    if (value == NULL) {
        rewriteConfigMarkAsProcessed(state,option);
        return;
    }

    /* Set force to zero if the value is set to its default. */
    if (defvalue && strcmp(value,defvalue) == 0) force = 0;

    line = sdsnew(option);
    line = sdscatlen(line, " ", 1);
    line = sdscatrepr(line, value, strlen(value));

    rewriteConfigRewriteLine(state,option,line,force);
}

/* Rewrite a SDS string option. */
void rewriteConfigSdsOption(struct rewriteConfigState *state, const char
*option, sds value, const char *defvalue) {
    int force = 1;

```

```

sds line;

/* If there is no value set, we don't want the SDS option
 * to be present in the configuration at all. */
if (value == NULL) {
    rewriteConfigMarkAsProcessed(state, option);
    return;
}

/* Set force to zero if the value is set to its default. */
if (defvalue && strcmp(value, defvalue) == 0) force = 0;

line = sdsnew(option);
line = sdscatlen(line, " ", 1);
line = sdscatrepr(line, value, sdslen(value));

rewriteConfigRewriteLine(state, option, line, force);
}

/* Rewrite a numerical (long long range) option. */
void rewriteConfigNumericalOption(struct rewriteConfigState *state, const char
*option, long long value, long long defvalue) {
    int force = value != defvalue;
    sds line = sdscatprintf(sdsempty(), "%s %lld", option, value);

    rewriteConfigRewriteLine(state, option, line, force);
}

/* Rewrite an octal option. */
void rewriteConfigOctalOption(struct rewriteConfigState *state, const char
*option, long long value, long long defvalue) {
    int force = value != defvalue;
    sds line = sdscatprintf(sdsempty(), "%s %llo", option, value);

    rewriteConfigRewriteLine(state, option, line, force);
}

/* Rewrite an enumeration option. It takes as usually state and option name,
 * and in addition the enumeration array and the default value for the
 * option. */
void rewriteConfigEnumOption(struct rewriteConfigState *state, const char
*option, int value, standardConfig *config) {
    int multiarg = config->flags & MULTI_ARG_CONFIG;
    sds names = configEnumGetName(config-
>data.enumd.enum_value, value, multiarg);
    sds line = sdscatfmt(sdsempty(), "%s %s", option, names);
    sdsfree(names);
    int force = value != config->data.enumd.default_value;

    rewriteConfigRewriteLine(state, option, line, force);
}

```

```

/* Rewrite the save option. */
void rewriteConfigSaveOption(standardConfig *config, const char *name, struct
rewriteConfigState *state) {
    UNUSED(config);
    int j;
    sds line;

    /* In Sentinel mode we don't need to rewrite the save parameters */
    if (server.sentinel_mode) {
        rewriteConfigMarkAsProcessed(state,name);
        return;
    }

    /* Rewrite save parameters, or an empty 'save ""' line to avoid the
     * defaults from being used.
     */
    if (!server.saveparamslen) {
        rewriteConfigRewriteLine(state,name,sdsnew("save \"\""),1);
    } else {
        for (j = 0; j < server.saveparamslen; j++) {
            line = sdscatprintf(sdsempy(),"save %ld %d",
                (long) server.saveparams[j].seconds,
server.saveparams[j].changes);
            rewriteConfigRewriteLine(state,name,line,1);
        }
    }

    /* Mark "save" as processed in case server.saveparamslen is zero. */
    rewriteConfigMarkAsProcessed(state,name);
}

/* Rewrite the user option. */
void rewriteConfigUserOption(struct rewriteConfigState *state) {
    /* If there is a user file defined we just mark this configuration
     * directive as processed, so that all the lines containing users
     * inside the config file gets discarded. */
    if (server.acl_filename[0] != '\0') {
        rewriteConfigMarkAsProcessed(state,"user");
        return;
    }

    /* Otherwise scan the list of users and rewrite every line. Note that
     * in case the list here is empty, the effect will just be to comment
     * all the users directive inside the config file. */
    raxIterator ri;
    raxStart(&ri,Users);
    raxSeek(&ri,"^",NULL,0);
    while(raxNext(&ri)) {
        user *u = ri.data;
        sds line = sdsnew("user ");
    }
}

```

```

        line = sdscatsds(line,u->name);
        line = sdscatlen(line," ",1);
        sds descr = ACLDescribeUser(u);
        line = sdscatsds(line,descr);
        sdsfree(descr);
        rewriteConfigRewriteLine(state,"user",line,1);
    }
    raxStop(&ri);

    /* Mark "user" as processed in case there are no defined users. */
    rewriteConfigMarkAsProcessed(state,"user");
}

/* Rewrite the dir option, always using absolute paths.*/
void rewriteConfigDirOption(standardConfig *config, const char *name, struct
rewriteConfigState *state) {
    UNUSED(config);
    char cwd[1024];

    if (getcwd(cwd,sizeof(cwd)) == NULL) {
        rewriteConfigMarkAsProcessed(state,name);
        return; /* no rewrite on error. */
    }
    rewriteConfigStringOption(state,name,cwd,NULL);
}

/* Rewrite the slaveof option. */
void rewriteConfigReplicaOfOption(standardConfig *config, const char *name,
struct rewriteConfigState *state) {
    UNUSED(config);
    sds line;

    /* If this is a master, we want all the slaveof config options
     * in the file to be removed. Note that if this is a cluster instance
     * we don't want a slaveof directive inside redis.conf. */
    if (server.cluster_enabled || server.masterhost == NULL) {
        rewriteConfigMarkAsProcessed(state, name);
        return;
    }
    line = sdscatprintf(sdsempty(),"%s %s %d", name,
        server.masterhost, server.masterport);
    rewriteConfigRewriteLine(state,name,line,1);
}

/* Rewrite the notify-keyspace-events option. */
void rewriteConfigNotifyKeyspaceEventsOption(standardConfig *config, const char
*name, struct rewriteConfigState *state) {
    UNUSED(config);
    int force = server.notify_keyspace_events != 0;
    sds line, flags;

```

```

    flags = keyspaceEventsFlagsToString(server.notify_keyspace_events);
    line = sdsnew(name);
    line = sdscatlen(line, " ", 1);
    line = sdscatrepr(line, flags, sdslen(flags));
    sdsfree(flags);
    rewriteConfigRewriteLine(state,name,line,force);
}

/* Rewrite the client-output-buffer-limit option. */
void rewriteConfigClientOutputBufferLimitOption(standardConfig *config, const
char *name, struct rewriteConfigState *state) {
    UNUSED(config);
    int j;
    for (j = 0; j < CLIENT_TYPE_OBUF_COUNT; j++) {
        int force = (server.client_obuf_limits[j].hard_limit_bytes !=
            clientBufferLimitsDefaults[j].hard_limit_bytes) ||
            (server.client_obuf_limits[j].soft_limit_bytes !=
            clientBufferLimitsDefaults[j].soft_limit_bytes) ||
            (server.client_obuf_limits[j].soft_limit_seconds !=
            clientBufferLimitsDefaults[j].soft_limit_seconds);

        sds line;
        char hard[64], soft[64];

        rewriteConfigFormatMemory(hard,sizeof(hard),
            server.client_obuf_limits[j].hard_limit_bytes);
        rewriteConfigFormatMemory(soft,sizeof(soft),
            server.client_obuf_limits[j].soft_limit_bytes);

        char *typename = getClientTypeName(j);
        if (!strcmp(typename,"slave")) typename = "replica";
        line = sdscatprintf(sdsempty(),"%s %s %s %s %ld",
            name, typename, hard, soft,
            (long) server.client_obuf_limits[j].soft_limit_seconds);
        rewriteConfigRewriteLine(state,name,line,force);
    }
}

/* Rewrite the oom-score-adj-values option. */
void rewriteConfigOOMScoreAdjValuesOption(standardConfig *config, const char
*name, struct rewriteConfigState *state) {
    UNUSED(config);
    int force = 0;
    int j;
    sds line;

    line = sdsnew(name);
    line = sdscatlen(line, " ", 1);
    for (j = 0; j < CONFIG_OOM_COUNT; j++) {
        if (server.oom_score_adj_values[j] !=
            configOOMScoreAdjValuesDefaults[j])
            force = 1;
    }
}

```



```

        line = sdscatprintf(line, "%d", server.oom_score_adj_values[j]);
        if (j+1 != CONFIG_OOM_COUNT)
            line = sdscatlen(line, " ", 1);
    }
    rewriteConfigRewriteLine(state,name,line,force);
}

/* Rewrite the bind option. */
void rewriteConfigBindOption(standardConfig *config, const char *name, struct
rewriteConfigState *state) {
    UNUSED(config);
    int force = 1;
    sds line, addresses;
    int is_default = 0;

    /* Compare server.bindaddr with CONFIG_DEFAULT_BINDADDR */
    if (server.bindaddr_count == CONFIG_DEFAULT_BINDADDR_COUNT) {
        is_default = 1;
        char *default_bindaddr[CONFIG_DEFAULT_BINDADDR_COUNT] =
CONFIG_DEFAULT_BINDADDR;
        for (int j = 0; j < CONFIG_DEFAULT_BINDADDR_COUNT; j++) {
            if (strcmp(server.bindaddr[j], default_bindaddr[j]) != 0) {
                is_default = 0;
                break;
            }
        }
    }

    if (is_default) {
        rewriteConfigMarkAsProcessed(state,name);
        return;
    }

    /* Rewrite as bind <addr1> <addr2> ... <addrN> */
    if (server.bindaddr_count > 0)
        addresses = sdsjoin(server.bindaddr,server.bindaddr_count," ");
    else
        addresses = sdsnew("\\"");
    line = sdsnew(name);
    line = sdscatlen(line, " ", 1);
    line = sdscatsds(line, addresses);
    sdsfree(addresses);

    rewriteConfigRewriteLine(state,name,line,force);
}

/* Rewrite the loadmodule option. */
void rewriteConfigLoadmoduleOption(struct rewriteConfigState *state) {
    sds line;

```

```

dictIterator *di = dictGetIterator(modules);
dictEntry *de;
while ((de = dictNext(di)) != NULL) {
    struct RedisModule *module = dictGetVal(de);
    line = sdsnew("loadmodule ");
    line = sdscatsds(line, module->loadmod->path);
    for (int i = 0; i < module->loadmod->argc; i++) {
        line = sdscatlen(line, " ", 1);
        line = sdscatsds(line, module->loadmod->argv[i]->ptr);
    }
    rewriteConfigRewriteLine(state,"loadmodule",line,1);
}
dictReleaseIterator(di);
/* Mark "loadmodule" as processed in case modules is empty. */
rewriteConfigMarkAsProcessed(state,"loadmodule");
}

/* Glue together the configuration lines in the current configuration
 * rewrite state into a single string, stripping multiple empty lines. */
sds rewriteConfigGetContentFromState(struct rewriteConfigState *state) {
    sds content = sdsempty();
    int j, was_empty = 0;

    for (j = 0; j < state->numlines; j++) {
        /* Every cluster of empty lines is turned into a single empty line. */
        if (sdslen(state->lines[j]) == 0) {
            if (was_empty) continue;
            was_empty = 1;
        } else {
            was_empty = 0;
        }
        content = sdscatsds(content,state->lines[j]);
        content = sdscatlen(content,"\n",1);
    }
    return content;
}

/* At the end of the rewrite process the state contains the remaining
 * map between "option name" => "lines in the original config file".
 * Lines used by the rewrite process were removed by the function
 * rewriteConfigRewriteLine(), all the other lines are "orphaned" and
 * should be replaced by empty lines.
 *
 * This function does just this, iterating all the option names and
 * blanking all the lines still associated. */
void rewriteConfigRemoveOrphaned(struct rewriteConfigState *state) {
    dictIterator *di = dictGetIterator(state->option_to_line);
    dictEntry *de;

    while((de = dictNext(di)) != NULL) {
        list *l = dictGetVal(de);

```

```

    sds option = dictGetKey(de);

    /* Don't blank lines about options the rewrite process
     * don't understand. */
    if (dictFind(state->rewritten,option) == NULL) {
        serverLog(LL_DEBUG,"Not rewritten option: %s", option);
        continue;
    }

    while(listLength(l)) {
        listNode *ln = listFirst(l);
        int linenum = (long) ln->value;

        sdsfree(state->lines[linenum]);
        state->lines[linenum] = sdsempty();
        listDelNode(l,ln);
    }
}
dictReleaseIterator(di);
}

/* This function returns a string representation of all the config options
 * marked with DEBUG_CONFIG, which can be used to help with debugging. */
sds getConfigDebugInfo() {
    struct rewriteConfigState *state = rewriteConfigCreateState();
    state->force_write = 1; /* Force the output */
    state->needs_signature = 0; /* Omit the rewrite signature */

    /* Iterate the configs and "rewrite" the ones that have
     * the debug flag. */
    dictIterator *di = dictGetIterator(configs);
    dictEntry *de;
    while ((de = dictNext(di)) != NULL) {
        standardConfig *config = dictGetVal(de);
        if (!(config->flags & DEBUG_CONFIG)) continue;
        config->interface.rewrite(config, config->name, state);
    }
    dictReleaseIterator(di);
    sds info = rewriteConfigGetContentFromState(state);
    rewriteConfigReleaseState(state);
    return info;
}

/* This function replaces the old configuration file with the new content
 * in an atomic manner.
 *
 * The function returns 0 on success, otherwise -1 is returned and errno
 * is set accordingly. */
int rewriteConfigOverwriteFile(char *configfile, sds content) {
    int fd = -1;
    int retval = -1;

```

```

char tmp_conffile[PATH_MAX];
const char *tmp_suffix = ".XXXXXX";
size_t offset = 0;
ssize_t written_bytes = 0;

int tmp_path_len = snprintf(tmp_conffile, sizeof(tmp_conffile), "%s%s",
configfile, tmp_suffix);
if (tmp_path_len <= 0 || (unsigned int)tmp_path_len >=
sizeof(tmp_conffile)) {
    serverLog(LL_WARNING, "Config file full path is too long");
    errno = ENAMETOOLONG;
    return retval;
}

#ifdef _GNU_SOURCE
    fd = mkostemp(tmp_conffile, O_CLOEXEC);
#else
    /* There's a theoretical chance here to leak the FD if a module thread
forks & execv in the middle */
    fd = mkstemp(tmp_conffile);
#endif

    if (fd == -1) {
        serverLog(LL_WARNING, "Could not create tmp config file (%s)",
strerror(errno));
        return retval;
    }

    while (offset < sdslen(content)) {
        written_bytes = write(fd, content + offset, sdslen(content) - offset);
        if (written_bytes <= 0) {
            if (errno == EINTR) continue; /* FD is blocking, no other
retryable errors */
            serverLog(LL_WARNING, "Failed after writing (%zd) bytes to tmp
config file (%s)", offset, strerror(errno));
            goto cleanup;
        }
        offset+=written_bytes;
    }

    if (fsync(fd))
        serverLog(LL_WARNING, "Could not sync tmp config file to disk (%s)",
strerror(errno));
    else if (fchmod(fd, 0644 & ~server.umask) == -1)
        serverLog(LL_WARNING, "Could not chmod config file (%s)",
strerror(errno));
    else if (rename(tmp_conffile, configfile) == -1)
        serverLog(LL_WARNING, "Could not rename tmp config file (%s)",
strerror(errno));
    else {
        retval = 0;
    }

```

```

        serverLog(LL_DEBUG, "Rewritten config file (%s) successfully",
configfile);
    }

cleanup:
    close(fd);
    if (retval) unlink(tmp_conffile);
    return retval;
}

/* Rewrite the configuration file at "path".
 * If the configuration file already exists, we try at best to retain comments
 * and overall structure.
 *
 * Configuration parameters that are at their default value, unless already
 * explicitly included in the old configuration file, are not rewritten.
 * The force_write flag overrides this behavior and forces everything to be
 * written. This is currently only used for testing purposes.
 *
 * On error -1 is returned and errno is set accordingly, otherwise 0. */
int rewriteConfig(char *path, int force_write) {
    struct rewriteConfigState *state;
    sds newcontent;
    int retval;

    /* Step 1: read the old config into our rewrite state. */
    if ((state = rewriteConfigReadOldFile(path)) == NULL) return -1;
    if (force_write) state->force_write = 1;

    /* Step 2: rewrite every single option, replacing or appending it inside
     * the rewrite state. */

    /* Iterate the configs that are standard */
    dictIterator *di = dictGetIterator(configs);
    dictEntry *de;
    while ((de = dictNext(di)) != NULL) {
        standardConfig *config = dictGetVal(de);
        /* Only rewrite the primary names */
        if (config->flags & ALIAS_CONFIG) continue;
        if (config->interface.rewrite) config->interface.rewrite(config, de->key, state);
    }
    dictReleaseIterator(di);

    rewriteConfigUserOption(state);
    rewriteConfigLoadmoduleOption(state);

    /* Rewrite Sentinel config if in Sentinel mode. */
    if (server.sentinel_mode) rewriteConfigSentinelOption(state);

    /* Step 3: remove all the orphaned lines in the old file, that is, lines

```

```

    * that were used by a config option and are no longer used, like in case
    * of multiple "save" options or duplicated options. */
rewriteConfigRemoveOrphaned(state);

/* Step 4: generate a new configuration file from the modified state
 * and write it into the original file. */
newcontent = rewriteConfigGetContentFromState(state);
retval = rewriteConfigOverwriteFile(server.configfile,newcontent);

sdsfree(newcontent);
rewriteConfigReleaseState(state);
return retval;
}

/*-----
 * Configs that fit one of the major types and require no special handling
 *-----
*/
#define LOADBUF_SIZE 256
static char loadbuf[LOADBUF_SIZE];

#define embedCommonConfig(config_name, config_alias, config_flags) \
    .name = (config_name), \
    .alias = (config_alias), \
    .flags = (config_flags),

#define embedConfigInterface(initfn, setfn, getfn, rewritefn, applyfn)
    .interface = { \
        .init = (initfn), \
        .set = (setfn), \
        .get = (getfn), \
        .rewrite = (rewritefn), \
        .apply = (applyfn) \
    },

/* What follows is the generic config types that are supported. To add a new
 * config with one of these types, add it to the standardConfig table with
 * the creation macro for each type.
 *
 * Each type contains the following:
 * * A function defining how to load this type on startup.
 * * A function defining how to update this type on CONFIG SET.
 * * A function defining how to serialize this type on CONFIG SET.
 * * A function defining how to rewrite this type on CONFIG REWRITE.
 * * A Macro defining how to create this type.
 */

/* Bool Configs */
static void boolConfigInit(standardConfig *config) {
    *config->data.yesno.config = config->data.yesno.default_value;
}

```

```

static int boolConfigSet(standardConfig *config, sds *argv, int argc, const
char **err) {
    UNUSED(argc);
    int yn = yesnotoi(argv[0]);
    if (yn == -1) {
        *err = "argument must be 'yes' or 'no'";
        return 0;
    }
    if (config->data.yesno.is_valid_fn && !config->data.yesno.is_valid_fn(yn,
err))
        return 0;
    int prev = config->flags & MODULE_CONFIG ? getModuleBoolConfig(config->
privdata) : *(config->data.yesno.config);
    if (prev != yn) {
        if (config->flags & MODULE_CONFIG) {
            return setModuleBoolConfig(config->privdata, yn, err);
        }
        *(config->data.yesno.config) = yn;
        return 1;
    }
    return 2;
}

static sds boolConfigGet(standardConfig *config) {
    if (config->flags & MODULE_CONFIG) {
        return sdsnew(getModuleBoolConfig(config->privdata) ? "yes" : "no");
    }
    return sdsnew(*(config->data.yesno.config) ? "yes" : "no");
}

static void boolConfigRewrite(standardConfig *config, const char *name, struct
rewriteConfigState *state) {
    int val = config->flags & MODULE_CONFIG ? getModuleBoolConfig(config->
privdata) : *(config->data.yesno.config);
    rewriteConfigYesNoOption(state, name, val, config->
data.yesno.default_value);
}

#define createBoolConfig(name, alias, flags, config_addr, default, is_valid,
apply) { \
    embedCommonConfig(name, alias, flags) \
    embedConfigInterface(boolConfigInit, boolConfigSet, boolConfigGet,
boolConfigRewrite, apply) \
    .type = BOOL_CONFIG, \
    .data.yesno = { \
        .config = &(config_addr), \
        .default_value = (default), \
        .is_valid_fn = (is_valid), \
    } \
}

```

```

/* String Configs */
static void stringConfigInit(standardConfig *config) {
    *config->data.string.config = (config->data.string.convert_empty_to_null &&
!config->data.string.default_value) ? NULL : zstrdup(config->
data.string.default_value);
}

static int stringConfigSet(standardConfig *config, sds *argv, int argc, const
char **err) {
    UNUSED(argc);
    if (config->data.string.is_valid_fn && !config->
data.string.is_valid_fn(argv[0], err))
        return 0;
    char *prev = *config->data.string.config;
    char *new = (config->data.string.convert_empty_to_null && !argv[0][0]) ?
NULL : argv[0];
    if (new != prev && (new == NULL || prev == NULL || strcmp(prev, new))) {
        *config->data.string.config = new != NULL ? zstrdup(new) : NULL;
        zfree(prev);
        return 1;
    }
    return 2;
}

static sds stringConfigGet(standardConfig *config) {
    return sdsnew(*config->data.string.config ? *config->data.string.config :
"" );
}

static void stringConfigRewrite(standardConfig *config, const char *name,
struct rewriteConfigState *state) {
    rewriteConfigStringOption(state, name, *(config->data.string.config),
config->data.string.default_value);
}

/* SDS Configs */
static void sdsConfigInit(standardConfig *config) {
    *config->data.sds.config = (config->data.sds.convert_empty_to_null &&
!config->data.sds.default_value) ? NULL : sdsnew(config->
data.sds.default_value);
}

static int sdsConfigSet(standardConfig *config, sds *argv, int argc, const char
**err) {
    UNUSED(argc);
    if (config->data.sds.is_valid_fn && !config->data.sds.is_valid_fn(argv[0],
err))
        return 0;

    sds prev = config->flags & MODULE_CONFIG ? getModuleStringConfig(config->

```



```

>privdata) : *config->data.sds.config;
    sds new = (config->data.string.convert_empty_to_null && (sdslen(argv[0]) ==
0)) ? NULL : argv[0];

    /* if prev and new configuration are not equal, set the new one */
    if (new != prev && (new == NULL || prev == NULL || sdscmp(prev, new))) {
        /* If MODULE_CONFIG flag is set, then free temporary prev
getModuleStringConfig returned.
        * Otherwise, free the actual previous config value Redis held (Same
action, different reasons) */
        sdsfree(prev);

        if (config->flags & MODULE_CONFIG) {
            return setModuleStringConfig(config->privdata, new, err);
        }
        *config->data.sds.config = new != NULL ? sdsdup(new) : NULL;
        return 1;
    }
    if (config->flags & MODULE_CONFIG && prev) sdsfree(prev);
    return 2;
}

static sds sdsConfigGet(standardConfig *config) {
    sds val = config->flags & MODULE_CONFIG ? getModuleStringConfig(config->
>privdata) : *config->data.sds.config;
    if (val) {
        if (config->flags & MODULE_CONFIG) return val;
        return sdsdup(val);
    } else {
        return sdsnew("");
    }
}

static void sdsConfigRewrite(standardConfig *config, const char *name, struct
rewriteConfigState *state) {
    sds val = config->flags & MODULE_CONFIG ? getModuleStringConfig(config->
>privdata) : *config->data.sds.config;
    rewriteConfigSdsOption(state, name, val, config->data.sds.default_value);
    if ((val) && (config->flags & MODULE_CONFIG)) sdsfree(val);
}

#define ALLOW_EMPTY_STRING 0
#define EMPTY_STRING_IS_NULL 1

#define createStringConfig(name, alias, flags, empty_to_null, config_addr,
default, is_valid, apply) { \
    embedCommonConfig(name, alias, flags) \
    embedConfigInterface(stringConfigInit, stringConfigSet, stringConfigGet,
stringConfigRewrite, apply) \
    .type = STRING_CONFIG, \

```

```

        .data.string = { \
            .config = &(config_addr), \
            .default_value = (default), \
            .is_valid_fn = (is_valid), \
            .convert_empty_to_null = (empty_to_null), \
        } \
    }

#define createSDSConfig(name, alias, flags, empty_to_null, config_addr,
default, is_valid, apply) { \
    embedCommonConfig(name, alias, flags) \
    embedConfigInterface(sdsConfigInit, sdsConfigSet, sdsConfigGet,
sdsConfigRewrite, apply) \
    .type = SDS_CONFIG, \
    .data.sds = { \
        .config = &(config_addr), \
        .default_value = (default), \
        .is_valid_fn = (is_valid), \
        .convert_empty_to_null = (empty_to_null), \
    } \
}

/* Enum configs */
static void enumConfigInit(standardConfig *config) {
    *config->data.enumd.config = config->data.enumd.default_value;
}

static int enumConfigSet(standardConfig *config, sds *argv, int argc, const
char **err) {
    int enumval;
    int bitflags = !(config->flags & MULTI_ARG_CONFIG);
    enumval = configEnumGetValue(config->data.enumd.enum_value, argv, argc,
bitflags);

    if (enumval == INT_MIN) {
        sds enumerr = sdsnew("argument(s) must be one of the following: ");
        configEnum *enumNode = config->data.enumd.enum_value;
        while(enumNode->name != NULL) {
            enumerr = sdscatlen(enumerr, enumNode->name,
                                strlen(enumNode->name));
            enumerr = sdscatlen(enumerr, ", ", 2);
            enumNode++;
        }
        sdsrange(enumerr,0,-3); /* Remove final ", ". */

        strncpy(loadbuf, enumerr, LOADBUF_SIZE);
        loadbuf[LOADBUF_SIZE - 1] = '\0';

        sdsfree(enumerr);
        *err = loadbuf;
        return 0;
    }
}

```

```

    }
    if (config->data.enumd.is_valid_fn && !config->data.enumd.is_valid_fn(enumval, err))
        return 0;
    int prev = config->flags & MODULE_CONFIG ? getModuleEnumConfig(config->privdata) : *(config->data.enumd.config);
    if (prev != enumval) {
        if (config->flags & MODULE_CONFIG)
            return setModuleEnumConfig(config->privdata, enumval, err);
        *(config->data.enumd.config) = enumval;
        return 1;
    }
    return 2;
}

static sds enumConfigGet(standardConfig *config) {
    int val = config->flags & MODULE_CONFIG ? getModuleEnumConfig(config->privdata) : *(config->data.enumd.config);
    int bitflags = !(config->flags & MULTI_ARG_CONFIG);
    return configEnumGetName(config->data.enumd.enum_value, val, bitflags);
}

static void enumConfigRewrite(standardConfig *config, const char *name, struct rewriteConfigState *state) {
    int val = config->flags & MODULE_CONFIG ? getModuleEnumConfig(config->privdata) : *(config->data.enumd.config);
    rewriteConfigEnumOption(state, name, val, config);
}

#define createEnumConfig(name, alias, flags, enum, config_addr, default, is_valid, apply) { \
    embedCommonConfig(name, alias, flags) \
    embedConfigInterface(enumConfigInit, enumConfigSet, enumConfigGet, enumConfigRewrite, apply) \
    .type = ENUM_CONFIG, \
    .data.enumd = { \
        .config = &(config_addr), \
        .default_value = (default), \
        .is_valid_fn = (is_valid), \
        .enum_value = (enum), \
    } \
}

/* Gets a 'long long val' and sets it into the union, using a macro to get
 * compile time type check. */
int setNumericType(standardConfig *config, long long val, const char **err) {
    if (config->data.numeric.numeric_type == NUMERIC_TYPE_INT) {
        *(config->data.numeric.config.i) = (int) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_UINT) {
        *(config->data.numeric.config.ui) = (unsigned int) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_LONG) {

```

```

        *(config->data.numeric.config.l) = (long) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_ULONG) {
        *(config->data.numeric.config.ul) = (unsigned long) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_LONG_LONG) {
        if (config->flags & MODULE_CONFIG)
            return setModuleNumericConfig(config->privdata, val, err);
        else *(config->data.numeric.config.ll) = (long long) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_ULONG_LONG) {
        *(config->data.numeric.config.ull) = (unsigned long long) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_SIZE_T) {
        *(config->data.numeric.config.st) = (size_t) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_SSIZE_T) {
        *(config->data.numeric.config.sst) = (ssize_t) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_OFF_T) {
        *(config->data.numeric.config.ot) = (off_t) val;
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_TIME_T) {
        *(config->data.numeric.config.tt) = (time_t) val;
    }
    return 1;
}

/* Gets a 'long long val' and sets it with the value from the union, using a
 * macro to get compile time type check. */
#define GET_NUMERIC_TYPE(val) \
    if (config->data.numeric.numeric_type == NUMERIC_TYPE_INT) { \
        val = *(config->data.numeric.config.i); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_UINT) { \
        val = *(config->data.numeric.config.ui); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_LONG) { \
        val = *(config->data.numeric.config.l); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_ULONG) { \
        val = *(config->data.numeric.config.ul); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_LONG_LONG) { \
        if (config->flags & MODULE_CONFIG) val = getModuleNumericConfig(config->privdata); \
        else val = *(config->data.numeric.config.ll); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_ULONG_LONG) { \
        \
        val = *(config->data.numeric.config.ull); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_SIZE_T) { \
        val = *(config->data.numeric.config.st); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_SSIZE_T) { \
        val = *(config->data.numeric.config.sst); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_OFF_T) { \
        val = *(config->data.numeric.config.ot); \
    } else if (config->data.numeric.numeric_type == NUMERIC_TYPE_TIME_T) { \
        val = *(config->data.numeric.config.tt); \
    }
}

/* Numeric configs */
static void numericConfigInit(standardConfig *config) {

```

```

    setNumericType(config, config->data.numeric.default_value, NULL);
}

static int numericBoundaryCheck(standardConfig *config, long long ll, const
char **err) {
    if (config->data.numeric.numeric_type == NUMERIC_TYPE_ULONG_LONG ||
        config->data.numeric.numeric_type == NUMERIC_TYPE_UINT ||
        config->data.numeric.numeric_type == NUMERIC_TYPE_SIZE_T) {
        /* Boundary check for unsigned types */
        unsigned long long ull = ll;
        unsigned long long upper_bound = config->data.numeric.upper_bound;
        unsigned long long lower_bound = config->data.numeric.lower_bound;
        if (ull > upper_bound || ull < lower_bound) {
            if (config->data.numeric.flags & OCTAL_CONFIG) {
                snprintf(loadbuf, LOADBUF_SIZE,
                    "argument must be between %llo and %llo inclusive",
                    lower_bound,
                    upper_bound);
            } else {
                snprintf(loadbuf, LOADBUF_SIZE,
                    "argument must be between %llu and %llu inclusive",
                    lower_bound,
                    upper_bound);
            }
            *err = loadbuf;
            return 0;
        }
    } else {
        /* Boundary check for percentages */
        if (config->data.numeric.flags & PERCENT_CONFIG && ll < 0) {
            if (ll < config->data.numeric.lower_bound) {
                snprintf(loadbuf, LOADBUF_SIZE,
                    "percentage argument must be less or equal to %lld",
                    -config->data.numeric.lower_bound);
                *err = loadbuf;
                return 0;
            }
        }
        /* Boundary check for signed types */
        else if (ll > config->data.numeric.upper_bound || ll < config-
>data.numeric.lower_bound) {
            snprintf(loadbuf, LOADBUF_SIZE,
                "argument must be between %lld and %lld inclusive",
                config->data.numeric.lower_bound,
                config->data.numeric.upper_bound);
            *err = loadbuf;
            return 0;
        }
    }
    return 1;
}

```

```

static int numericParseString(standardConfig *config, sds value, const char
**err, long long *res) {
    /* First try to parse as memory */
    if (config->data.numeric.flags & MEMORY_CONFIG) {
        int memerr;
        *res = memtoll(value, &memerr);
        if (!memerr)
            return 1;
    }

    /* Attempt to parse as percent */
    if (config->data.numeric.flags & PERCENT_CONFIG &&
        sdslen(value) > 1 && value[sdslen(value)-1] == '%' &&
        string2ll(value, sdslen(value)-1, res) &&
        *res >= 0) {
        /* We store percentage as negative value */
        *res = -*res;
        return 1;
    }

    /* Attempt to parse as an octal number */
    if (config->data.numeric.flags & OCTAL_CONFIG) {
        char *endptr;
        errno = 0;
        *res = strtoll(value, &endptr, 8);
        if (errno == 0 && *endptr == '\0')
            return 1; /* No overflow or invalid characters */
    }

    /* Attempt a simple number (no special flags set) */
    if (!config->data.numeric.flags && string2ll(value, sdslen(value), res))
        return 1;

    /* Select appropriate error string */
    if (config->data.numeric.flags & MEMORY_CONFIG &&
        config->data.numeric.flags & PERCENT_CONFIG)
        *err = "argument must be a memory or percent value" ;
    else if (config->data.numeric.flags & MEMORY_CONFIG)
        *err = "argument must be a memory value";
    else if (config->data.numeric.flags & OCTAL_CONFIG)
        *err = "argument couldn't be parsed as an octal number";
    else
        *err = "argument couldn't be parsed into an integer";
    return 0;
}

static int numericConfigSet(standardConfig *config, sds *argv, int argc, const
char **err) {
    UNUSED(argc);
    long long ll, prev = 0;

```

```

    if (!numericParseString(config, argv[0], err, &ll))
        return 0;

    if (!numericBoundaryCheck(config, ll, err))
        return 0;

    if (config->data.numeric.is_valid_fn && !config->data.numeric.is_valid_fn(ll, err))
        return 0;

    GET_NUMERIC_TYPE(prev)
    if (prev != ll) {
        return setNumericType(config, ll, err);
    }

    return 2;
}

static sds numericConfigGet(standardConfig *config) {
    char buf[128];

    long long value = 0;
    GET_NUMERIC_TYPE(value)

    if (config->data.numeric.flags & PERCENT_CONFIG && value < 0) {
        int len = ll2string(buf, sizeof(buf), -value);
        buf[len] = '%';
        buf[len+1] = '\\0';
    }
    else if (config->data.numeric.flags & MEMORY_CONFIG) {
        ull2string(buf, sizeof(buf), value);
    } else if (config->data.numeric.flags & OCTAL_CONFIG) {
        snprintf(buf, sizeof(buf), "%llo", value);
    } else {
        ll2string(buf, sizeof(buf), value);
    }
    return sdsnew(buf);
}

static void numericConfigRewrite(standardConfig *config, const char *name,
struct rewriteConfigState *state) {
    long long value = 0;

    GET_NUMERIC_TYPE(value)

    if (config->data.numeric.flags & PERCENT_CONFIG && value < 0) {
        rewriteConfigPercentOption(state, name, -value, config->data.numeric.default_value);
    } else if (config->data.numeric.flags & MEMORY_CONFIG) {
        rewriteConfigBytesOption(state, name, value, config->data.numeric.default_value);
    }
}

```

```

>data.numeric.default_value);
    } else if (config->data.numeric.flags & OCTAL_CONFIG) {
        rewriteConfigOctalOption(state, name, value, config->data.numeric.default_value);
    } else {
        rewriteConfigNumericalOption(state, name, value, config->data.numeric.default_value);
    }
}

#define embedCommonNumericalConfig(name, alias, _flags, lower, upper,
config_addr, default, num_conf_flags, is_valid, apply) { \
    embedCommonConfig(name, alias, _flags) \
    embedConfigInterface(numericConfigInit, numericConfigSet, numericConfigGet,
numericConfigRewrite, apply) \
    .type = NUMERIC_CONFIG, \
    .data.numeric = { \
        .lower_bound = (lower), \
        .upper_bound = (upper), \
        .default_value = (default), \
        .is_valid_fn = (is_valid), \
        .flags = (num_conf_flags),

#define createIntConfig(name, alias, flags, lower, upper, config_addr, default,
num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_INT, \
    .config.i = &(config_addr) \
} \
}

#define createUIntConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_UINT, \
    .config.ui = &(config_addr) \
} \
}

#define createLongConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_LONG, \
    .config.l = &(config_addr) \
} \
}

#define createULongConfig(name, alias, flags, lower, upper, config_addr,

```



```

default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_ULONG, \
    .config.ul = &(config_addr) \
} \
}

#define createLongLongConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_LONG_LONG, \
    .config.ll = &(config_addr) \
} \
}

#define createULongLongConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_ULONG_LONG, \
    .config.ull = &(config_addr) \
} \
}

#define createSizeTConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_SIZE_T, \
    .config.st = &(config_addr) \
} \
}

#define createSSizeTConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_SSIZE_T, \
    .config.sst = &(config_addr) \
} \
}

#define createTimeTConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    .numeric_type = NUMERIC_TYPE_TIME_T, \
    .config.tt = &(config_addr) \
} \
}

```

```

}

#define createOffTConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
    embedCommonNumericalConfig(name, alias, flags, lower, upper, config_addr,
default, num_conf_flags, is_valid, apply) \
        .numeric_type = NUMERIC_TYPE_OFF_T, \
        .config.ot = &(config_addr) \
    } \
}

#define createSpecialConfig(name, alias, modifiable, setfn, getfn, rewritefn,
applyfn) { \
    .type = SPECIAL_CONFIG, \
    embedCommonConfig(name, alias, modifiable) \
    embedConfigInterface(NULL, setfn, getfn, rewritefn, applyfn) \
}

static int isValidActiveDefrag(int val, const char **err) {
#ifdef HAVE_DEFRAG
    if (val) {
        *err = "Active defragmentation cannot be enabled: it "
            "requires a Redis server compiled with a modified Jemalloc "
            "like the one shipped by default with the Redis source "
            "distribution";
        return 0;
    }
#else
    UNUSED(val);
    UNUSED(err);
#endif
    return 1;
}

static int isValidDBfilename(char *val, const char **err) {
    if (!pathIsBaseName(val)) {
        *err = "dbfilename can't be a path, just a filename";
        return 0;
    }
    return 1;
}

static int isValidAOFfilename(char *val, const char **err) {
    if (!strcmp(val, "")) {
        *err = "appendfilename can't be empty";
        return 0;
    }
    if (!pathIsBaseName(val)) {
        *err = "appendfilename can't be a path, just a filename";
        return 0;
    }
}

```

```

    return 1;
}

static int isValidAOFdirname(char *val, const char **err) {
    if (!strcmp(val, "")) {
        *err = "appenddirname can't be empty";
        return 0;
    }
    if (!pathIsBaseName(val)) {
        *err = "appenddirname can't be a path, just a dirname";
        return 0;
    }
    return 1;
}

static int isValidShutdownOnSigFlags(int val, const char **err) {
    /* Individual arguments are validated by createEnumConfig logic.
     * We just need to ensure valid combinations here. */
    if (val & SHUTDOWN_NOSAVE && val & SHUTDOWN_SAVE) {
        *err = "shutdown options SAVE and NOSAVE can't be used simultaneously";
        return 0;
    }
    return 1;
}

static int isValidAnnouncedHostname(char *val, const char **err) {
    if (strlen(val) >= NET_HOST_STR_LEN) {
        *err = "Hostnames must be less than "
            STRINGIFY(NET_HOST_STR_LEN) " characters";
        return 0;
    }

    int i = 0;
    char c;
    while ((c = val[i])) {
        /* We just validate the character set to make sure that everything
         * is parsed and handled correctly. */
        if (!((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')
            || (c >= '0' && c <= '9') || (c == '-') || (c == '.')))
        {
            *err = "Hostnames may only contain alphanumeric characters, "
                "hyphens or dots";
            return 0;
        }
        c = val[i++];
    }
    return 1;
}

/* Validate specified string is a valid proc-title-template */
static int isValidProcTitleTemplate(char *val, const char **err) {

```

```

    if (!validateProcTitleTemplate(val)) {
        *err = "template format is invalid or contains unknown variables";
        return 0;
    }
    return 1;
}

static int updateProcTitleTemplate(const char **err) {
    if (redisSetProcTitle(NULL) == C_ERR) {
        *err = "failed to set process title";
        return 0;
    }
    return 1;
}

static int updateHZ(const char **err) {
    UNUSED(err);
    /* HZ is more a hint from the user, so we accept values out of range
     * but cap them to reasonable values. */
    if (server.config_hz < CONFIG_MIN_HZ) server.config_hz = CONFIG_MIN_HZ;
    if (server.config_hz > CONFIG_MAX_HZ) server.config_hz = CONFIG_MAX_HZ;
    server.hz = server.config_hz;
    return 1;
}

static int updatePort(const char **err) {
    if (changeListenPort(server.port, &server.ipfd, acceptTcpHandler) == C_ERR)
    {
        *err = "Unable to listen on this port. Check server logs.";
        return 0;
    }

    return 1;
}

static int updateJemallocBgThread(const char **err) {
    UNUSED(err);
    set_jemalloc_bg_thread(server.jemalloc_bg_thread);
    return 1;
}

static int updateReplBacklogSize(const char **err) {
    UNUSED(err);
    resizeReplicationBacklog();
    return 1;
}

static int updateMaxmemory(const char **err) {
    UNUSED(err);
    if (server.maxmemory) {
        size_t used = zmalloc_used_memory() - freeMemoryGetNotCountedMemory();

```

```

        if (server.maxmemory < used) {
            serverLog(LL_WARNING,"WARNING: the new maxmemory value set via
CONFIG SET (%llu) is smaller than the current memory usage (%zu). This will
result in key eviction and/or the inability to accept new write commands
depending on the maxmemory-policy.", server.maxmemory, used);
        }
        startEvictionTimeProc();
    }
    return 1;
}

static int updateGoodSlaves(const char **err) {
    UNUSED(err);
    refreshGoodSlavesCount();
    return 1;
}

static int updateWatchdogPeriod(const char **err) {
    UNUSED(err);
    applyWatchdogPeriod();
    return 1;
}

static int updateAppendonly(const char **err) {
    if (!server.aof_enabled && server.aof_state != AOF_OFF) {
        stopAppendOnly();
    } else if (server.aof_enabled && server.aof_state == AOF_OFF) {
        if (startAppendOnly() == C_ERR) {
            *err = "Unable to turn on AOF. Check server logs.";
            return 0;
        }
    }
    return 1;
}

static int updateAofAutoGCEnabled(const char **err) {
    UNUSED(err);
    if (!server.aof_disable_auto_gc) {
        aofDelHistoryFiles();
    }

    return 1;
}

static int updateSighandlerEnabled(const char **err) {
    UNUSED(err);
    if (server.crashlog_enabled)
        setupSignalHandlers();
    else
        removeSignalHandlers();
    return 1;
}

```

```

}

static int updateMaxclients(const char **err) {
    unsigned int new_maxclients = server.maxclients;
    adjustOpenFilesLimit();
    if (server.maxclients != new_maxclients) {
        static char msg[128];
        sprintf(msg, "The operating system is not able to handle the specified
number of clients, try with %d", server.maxclients);
        *err = msg;
        return 0;
    }
    if ((unsigned int) aeGetSetSize(server.el) <
        server.maxclients + CONFIG_FDSET_INCR)
    {
        if (aeResizeSetSize(server.el,
            server.maxclients + CONFIG_FDSET_INCR) == AE_ERR)
        {
            *err = "The event loop API used by Redis is not able to handle the
specified number of clients";
            return 0;
        }
    }
    return 1;
}

static int updateOOMScoreAdj(const char **err) {
    if (setOOMScoreAdj(-1) == C_ERR) {
        *err = "Failed to set current oom_score_adj. Check server logs.";
        return 0;
    }

    return 1;
}

int updateRequirePass(const char **err) {
    UNUSED(err);
    /* The old "requirepass" directive just translates to setting
    * a password to the default user. The only thing we do
    * additionally is to remember the cleartext password in this
    * case, for backward compatibility with Redis <= 5. */
    ACLUpdateDefaultUserPassword(server.requirepass);
    return 1;
}

static int applyBind(const char **err) {
    if (changeBindAddr() == C_ERR) {
        *err = "Failed to bind to specified addresses.";
        return 0;
    }
}

```

```

        return 1;
    }

    int updateClusterFlags(const char **err) {
        UNUSED(err);
        clusterUpdateMyselfFlags();
        return 1;
    }

    static int updateClusterIp(const char **err) {
        UNUSED(err);
        clusterUpdateMyselfIp();
        return 1;
    }

    int updateClusterHostname(const char **err) {
        UNUSED(err);
        clusterUpdateMyselfHostname();
        return 1;
    }

#ifdef USE_OPENSSL
    static int applyTlsCfg(const char **err) {
        UNUSED(err);

        /* If TLS is enabled, try to configure OpenSSL. */
        if ((server.tls_port || server.tls_replication || server.tls_cluster)
            && tlsConfigure(&server.tls_ctx_config) == C_ERR) {
            *err = "Unable to update TLS configuration. Check server logs.";
            return 0;
        }
        return 1;
    }

    static int applyTLSPort(const char **err) {
        /* Configure TLS in case it wasn't enabled */
        if (!isTlsConfigured() && tlsConfigure(&server.tls_ctx_config) == C_ERR) {
            *err = "Unable to update TLS configuration. Check server logs.";
            return 0;
        }

        if (changeListenPort(server.tls_port, &server.tlsfd, acceptTLSHandler) ==
            C_ERR) {
            *err = "Unable to listen on this port. Check server logs.";
            return 0;
        }

        return 1;
    }

#endif /* USE_OPENSSL */

```

```

static int setConfigDirOption(standardConfig *config, sds *argv, int argc,
const char **err) {
    UNUSED(config);
    if (argc != 1) {
        *err = "wrong number of arguments";
        return 0;
    }
    if (chdir(argv[0]) == -1) {
        *err = strerror(errno);
        return 0;
    }
    return 1;
}

static sds getConfigDirOption(standardConfig *config) {
    UNUSED(config);
    char buf[1024];

    if (getcwd(buf, sizeof(buf)) == NULL)
        buf[0] = '\0';

    return sdsnew(buf);
}

static int setConfigSaveOption(standardConfig *config, sds *argv, int argc,
const char **err) {
    UNUSED(config);
    int j;

    /* Special case: treat single arg "" as zero args indicating empty save
configuration */
    if (argc == 1 && !strcasecmp(argv[0], ""))
        argc = 0;

    /* Perform sanity check before setting the new config:
* - Even number of args
* - Seconds >= 1, changes >= 0 */
    if (argc & 1) {
        *err = "Invalid save parameters";
        return 0;
    }
    for (j = 0; j < argc; j++) {
        char *eptr;
        long val;

        val = strtoll(argv[j], &eptr, 10);
        if (eptr[0] != '\0' ||
            ((j & 1) == 0 && val < 1) ||
            ((j & 1) == 1 && val < 0)) {
            *err = "Invalid save parameters";

```



```

        return 0;
    }
}
/* Finally set the new config */
if (!reading_config_file) {
    resetServerSaveParams();
} else {
    /* We don't reset save params before loading, because if they're not
part
    * of the file the defaults should be used.
    */
    static int save_loaded = 0;
    if (!save_loaded) {
        save_loaded = 1;
        resetServerSaveParams();
    }
}

for (j = 0; j < argc; j += 2) {
    time_t seconds;
    int changes;

    seconds = strtoll(argv[j],NULL,10);
    changes = strtoll(argv[j+1],NULL,10);
    appendServerSaveParams(seconds, changes);
}

return 1;
}

static sds getConfigSaveOption(standardConfig *config) {
    UNUSED(config);
    sds buf = sdsempty();
    int j;

    for (j = 0; j < server.saveparamslen; j++) {
        buf = sdscatprintf(buf,"%jd %d",
                           (intmax_t)server.saveparams[j].seconds,
                           server.saveparams[j].changes);
        if (j != server.saveparamslen-1)
            buf = sdscatlen(buf," ",1);
    }

    return buf;
}

static int setConfigClientOutputBufferLimitOption(standardConfig *config, sds
*argv, int argc, const char **err) {
    UNUSED(config);
    return updateClientOutputBufferLimit(argv, argc, err);
}

```

```

static sds getConfigClientOutputBufferLimitOption(standardConfig *config) {
    UNUSED(config);
    sds buf = sdsempty();
    int j;
    for (j = 0; j < CLIENT_TYPE_OBUF_COUNT; j++) {
        buf = sdscatprintf(buf, "%s %llu %llu %ld",
                           getClientTypeName(j),
                           server.client_obuf_limits[j].hard_limit_bytes,
                           server.client_obuf_limits[j].soft_limit_bytes,
                           (long)
server.client_obuf_limits[j].soft_limit_seconds);
        if (j != CLIENT_TYPE_OBUF_COUNT-1)
            buf = sdscatlen(buf, " ", 1);
    }
    return buf;
}

/* Parse an array of CONFIG_OOM_COUNT sds strings, validate and populate
 * server.oom_score_adj_values if valid.
 */
static int setConfigOOMScoreAdjValuesOption(standardConfig *config, sds *argv,
int argc, const char **err) {
    int i;
    int values[CONFIG_OOM_COUNT];
    int change = 0;
    UNUSED(config);

    if (argc != CONFIG_OOM_COUNT) {
        *err = "wrong number of arguments";
        return 0;
    }

    for (i = 0; i < CONFIG_OOM_COUNT; i++) {
        char *eptr;
        long long val = strtoll(argv[i], &eptr, 10);

        if (*eptr != '\0' || val < -2000 || val > 2000) {
            if (err) *err = "Invalid oom-score-adj-values, elements must be
between -2000 and 2000.";
            return 0;
        }

        values[i] = val;
    }

    /* Verify that the values make sense. If they don't omit a warning but
     * keep the configuration, which may still be valid for privileged
     * processes.
     */
}

```

```

    if (values[CONFIG_OOM_REPLICA] < values[CONFIG_OOM_MASTER] ||
        values[CONFIG_OOM_BGCHILD] < values[CONFIG_OOM_REPLICA])
    {
        serverLog(LL_WARNING,
            "The oom-score-adj-values configuration may not work for non-
privileged processes! "
            "Please consult the documentation.");
    }

    for (i = 0; i < CONFIG_OOM_COUNT; i++) {
        if (server.oom_score_adj_values[i] != values[i]) {
            server.oom_score_adj_values[i] = values[i];
            change = 1;
        }
    }

    return change ? 1 : 2;
}

static sds getConfigOOMScoreAdjValuesOption(standardConfig *config) {
    UNUSED(config);
    sds buf = sdsempty();
    int j;

    for (j = 0; j < CONFIG_OOM_COUNT; j++) {
        buf = sdscatprintf(buf, "%d", server.oom_score_adj_values[j]);
        if (j != CONFIG_OOM_COUNT-1)
            buf = sdscatlen(buf, " ", 1);
    }

    return buf;
}

static int setConfigNotifyKeyspaceEventsOption(standardConfig *config, sds
*argv, int argc, const char **err) {
    UNUSED(config);
    if (argc != 1) {
        *err = "wrong number of arguments";
        return 0;
    }
    int flags = keyspaceEventsStringToFlags(argv[0]);
    if (flags == -1) {
        *err = "Invalid event class character. Use 'Ag$lshzxeKEtmdn'.";
        return 0;
    }
    server.notify_keyspace_events = flags;
    return 1;
}

static sds getConfigNotifyKeyspaceEventsOption(standardConfig *config) {
    UNUSED(config);

```

```

    return keyspaceEventsFlagsToString(server.notify_keyspace_events);
}

static int setConfigBindOption(standardConfig *config, sds* argv, int argc,
const char **err) {
    UNUSED(config);
    int j;

    if (argc > CONFIG_BINDADDR_MAX) {
        *err = "Too many bind addresses specified.";
        return 0;
    }

    /* A single empty argument is treated as a zero bindaddr count */
    if (argc == 1 && sdslen(argv[0]) == 0) argc = 0;

    /* Free old bind addresses */
    for (j = 0; j < server.bindaddr_count; j++) {
        zfree(server.bindaddr[j]);
    }
    for (j = 0; j < argc; j++)
        server.bindaddr[j] = zstrdup(argv[j]);
    server.bindaddr_count = argc;

    return 1;
}

static int setConfigReplicaOfOption(standardConfig *config, sds* argv, int
argc, const char **err) {
    UNUSED(config);

    if (argc != 2) {
        *err = "wrong number of arguments";
        return 0;
    }

    sdsfree(server.masterhost);
    server.masterhost = NULL;
    if (!strcasecmp(argv[0], "no") && !strcasecmp(argv[1], "one")) {
        return 1;
    }
    char *ptr;
    server.masterport = strtol(argv[1], &ptr, 10);
    if (server.masterport < 0 || server.masterport > 65535 || *ptr != '\0') {
        *err = "Invalid master port";
        return 0;
    }
    server.masterhost = sdsnew(argv[0]);
    server.repl_state = REPL_STATE_CONNECT;
    return 1;
}

```

```

static sds getConfigBindOption(standardConfig *config) {
    UNUSED(config);
    return sdsjoin(server.bindaddr,server.bindaddr_count," ");
}

static sds getConfigReplicaOfOption(standardConfig *config) {
    UNUSED(config);
    char buf[256];
    if (server.masterhost)
        snprintf(buf,sizeof(buf),"%s %d",
                 server.masterhost, server.masterport);
    else
        buf[0] = '\0';
    return sdsnew(buf);
}

int allowProtectedAction(int config, client *c) {
    return (config == PROTECTED_ACTION_ALLOWED_YES) ||
        (config == PROTECTED_ACTION_ALLOWED_LOCAL && islocalClient(c));
}

static int setConfigLatencyTrackingInfoPercentilesOutputOption(standardConfig
*config, sds *argv, int argc, const char **err) {
    UNUSED(config);
    zfree(server.latency_tracking_info_percentiles);
    server.latency_tracking_info_percentiles = NULL;
    server.latency_tracking_info_percentiles_len = argc;

    /* Special case: treat single arg "" as zero args indicating empty
    percentile configuration */
    if (argc == 1 && sdslen(argv[0]) == 0)
        server.latency_tracking_info_percentiles_len = 0;
    else
        server.latency_tracking_info_percentiles =
zmalloc(sizeof(double)*argc);

    for (int j = 0; j < server.latency_tracking_info_percentiles_len; j++) {
        double percentile;
        if (!string2d(argv[j], sdslen(argv[j]), &percentile)) {
            *err = "Invalid latency-tracking-info-percentiles parameters";
            goto configerr;
        }
        if (percentile > 100.0 || percentile < 0.0) {
            *err = "latency-tracking-info-percentiles parameters should sit
between [0.0,100.0]";
            goto configerr;
        }
        server.latency_tracking_info_percentiles[j] = percentile;
    }
}

```

```

    return 1;
configerr:
    zfree(server.latency_tracking_info_percentiles);
    server.latency_tracking_info_percentiles = NULL;
    server.latency_tracking_info_percentiles_len = 0;
    return 0;
}

static sds getConfigLatencyTrackingInfoPercentilesOutputOption(standardConfig
*config) {
    UNUSED(config);
    sds buf = sdsempty();
    for (int j = 0; j < server.latency_tracking_info_percentiles_len; j++) {
        char fbuf[128];
        size_t len = sprintf(fbuf, "%f",
server.latency_tracking_info_percentiles[j]);
        len = trimDoubleString(fbuf, len);
        buf = sdscatlen(buf, fbuf, len);
        if (j != server.latency_tracking_info_percentiles_len-1)
            buf = sdscatlen(buf, " ",1);
    }
    return buf;
}

/* Rewrite the latency-tracking-info-percentiles option. */
void rewriteConfigLatencyTrackingInfoPercentilesOutputOption(standardConfig
*config, const char *name, struct rewriteConfigState *state) {
    UNUSED(config);
    sds line = sdsnew(name);
    /* Rewrite latency-tracking-info-percentiles parameters,
     * or an empty 'latency-tracking-info-percentiles ""' line to avoid the
     * defaults from being used.
     */
    if (!server.latency_tracking_info_percentiles_len) {
        line = sdscat(line, " \\\"\\\"");
    } else {
        for (int j = 0; j < server.latency_tracking_info_percentiles_len; j++)
        {
            char fbuf[128];
            size_t len = sprintf(fbuf, " %f",
server.latency_tracking_info_percentiles[j]);
            len = trimDoubleString(fbuf, len);
            line = sdscatlen(line, fbuf, len);
        }
        rewriteConfigRewriteLine(state,name,line,1);
    }
}

standardConfig static_configs[] = {
    /* Bool configs */

```

```

    createBoolConfig("rdbchecksum", NULL, IMMUTABLE_CONFIG,
server.rdb_checksum, 1, NULL, NULL),
    createBoolConfig("daemonize", NULL, IMMUTABLE_CONFIG, server.daemonize, 0,
NULL, NULL),
    createBoolConfig("io-threads-do-reads", NULL, DEBUG_CONFIG |
IMMUTABLE_CONFIG, server.io_threads_do_reads, 0, NULL, NULL), /* Read + parse
from threads? */
    createBoolConfig("always-show-logo", NULL, IMMUTABLE_CONFIG,
server.always_show_logo, 0, NULL, NULL),
    createBoolConfig("protected-mode", NULL, MODIFIABLE_CONFIG,
server.protected_mode, 1, NULL, NULL),
    createBoolConfig("rdbcompression", NULL, MODIFIABLE_CONFIG,
server.rdb_compression, 1, NULL, NULL),
    createBoolConfig("rdb-del-sync-files", NULL, MODIFIABLE_CONFIG,
server.rdb_del_sync_files, 0, NULL, NULL),
    createBoolConfig("activeresharding", NULL, MODIFIABLE_CONFIG,
server.activeresharding, 1, NULL, NULL),
    createBoolConfig("stop-writes-on-bgsave-error", NULL, MODIFIABLE_CONFIG,
server.stop_writes_on_bgsave_err, 1, NULL, NULL),
    createBoolConfig("set-proc-title", NULL, IMMUTABLE_CONFIG,
server.set_proc_title, 1, NULL, NULL), /* Should setproctitle be used? */
    createBoolConfig("dynamic-hz", NULL, MODIFIABLE_CONFIG, server.dynamic_hz,
1, NULL, NULL), /* Adapt hz to # of clients.*/
    createBoolConfig("lazyfree-lazy-eviction", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, server.lazyfree_lazy_eviction, 0, NULL, NULL),
    createBoolConfig("lazyfree-lazy-expire", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, server.lazyfree_lazy_expire, 0, NULL, NULL),
    createBoolConfig("lazyfree-lazy-server-del", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, server.lazyfree_lazy_server_del, 0, NULL, NULL),
    createBoolConfig("lazyfree-lazy-user-del", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, server.lazyfree_lazy_user_del , 0, NULL, NULL),
    createBoolConfig("lazyfree-lazy-user-flush", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, server.lazyfree_lazy_user_flush , 0, NULL, NULL),
    createBoolConfig("repl-disable-tcp-nodelay", NULL, MODIFIABLE_CONFIG,
server.repl_disable_tcp_nodelay, 0, NULL, NULL),
    createBoolConfig("repl-diskless-sync", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, server.repl_diskless_sync, 1, NULL, NULL),
    createBoolConfig("aof-rewrite-incremental-fsync", NULL, MODIFIABLE_CONFIG,
server.aof_rewrite_incremental_fsync, 1, NULL, NULL),
    createBoolConfig("no-appendfsync-on-rewrite", NULL, MODIFIABLE_CONFIG,
server.aof_no_fsync_on_rewrite, 0, NULL, NULL),
    createBoolConfig("cluster-require-full-coverage", NULL, MODIFIABLE_CONFIG,
server.cluster_require_full_coverage, 1, NULL, NULL),
    createBoolConfig("rdb-save-incremental-fsync", NULL, MODIFIABLE_CONFIG,
server.rdb_save_incremental_fsync, 1, NULL, NULL),
    createBoolConfig("aof-load-truncated", NULL, MODIFIABLE_CONFIG,
server.aof_load_truncated, 1, NULL, NULL),
    createBoolConfig("aof-use-rdb-preamble", NULL, MODIFIABLE_CONFIG,
server.aof_use_rdb_preamble, 1, NULL, NULL),
    createBoolConfig("aof-timestamp-enabled", NULL, MODIFIABLE_CONFIG,
server.aof_timestamp_enabled, 0, NULL, NULL),

```

```

    createBoolConfig("cluster-replica-no-failover", "cluster-slave-no-
failover", MODIFIABLE_CONFIG, server.cluster_slave_no_failover, 0, NULL,
updateClusterFlags), /* Failover by default. */
    createBoolConfig("replica-lazy-flush", "slave-lazy-flush",
MODIFIABLE_CONFIG, server.repl_slave_lazy_flush, 0, NULL, NULL),
    createBoolConfig("replica-serve-stale-data", "slave-serve-stale-data",
MODIFIABLE_CONFIG, server.repl_serve_stale_data, 1, NULL, NULL),
    createBoolConfig("replica-read-only", "slave-read-only", DEBUG_CONFIG |
MODIFIABLE_CONFIG, server.repl_slave_ro, 1, NULL, NULL),
    createBoolConfig("replica-ignore-maxmemory", "slave-ignore-maxmemory",
MODIFIABLE_CONFIG, server.repl_slave_ignore_maxmemory, 1, NULL, NULL),
    createBoolConfig("jemalloc-bg-thread", NULL, MODIFIABLE_CONFIG,
server.jemalloc_bg_thread, 1, NULL, updateJemallocBgThread),
    createBoolConfig("activedefrag", NULL, DEBUG_CONFIG | MODIFIABLE_CONFIG,
server.active_defrag_enabled, 0, isValidActiveDefrag, NULL),
    createBoolConfig("syslog-enabled", NULL, IMMUTABLE_CONFIG,
server.syslog_enabled, 0, NULL, NULL),
    createBoolConfig("cluster-enabled", NULL, IMMUTABLE_CONFIG,
server.cluster_enabled, 0, NULL, NULL),
    createBoolConfig("appendonly", NULL, MODIFIABLE_CONFIG |
DENY_LOADING_CONFIG, server.aof_enabled, 0, NULL, updateAppendonly),
    createBoolConfig("cluster-allow-reads-when-down", NULL, MODIFIABLE_CONFIG,
server.cluster_allow_reads_when_down, 0, NULL, NULL),
    createBoolConfig("cluster-allow-pubsubshard-when-down", NULL,
MODIFIABLE_CONFIG, server.cluster_allow_pubsubshard_when_down, 1, NULL, NULL),
    createBoolConfig("crash-log-enabled", NULL, MODIFIABLE_CONFIG,
server.crashlog_enabled, 1, NULL, updateSigHandlerEnabled),
    createBoolConfig("crash-memcheck-enabled", NULL, MODIFIABLE_CONFIG,
server.memcheck_enabled, 1, NULL, NULL),
    createBoolConfig("use-exit-on-panic", NULL, MODIFIABLE_CONFIG |
HIDDEN_CONFIG, server.use_exit_on_panic, 0, NULL, NULL),
    createBoolConfig("disable-thp", NULL, IMMUTABLE_CONFIG, server.disable_thp,
1, NULL, NULL),
    createBoolConfig("cluster-allow-replica-migration", NULL,
MODIFIABLE_CONFIG, server.cluster_allow_replica_migration, 1, NULL, NULL),
    createBoolConfig("replica-announced", NULL, MODIFIABLE_CONFIG,
server.replica_announced, 1, NULL, NULL),
    createBoolConfig("latency-tracking", NULL, MODIFIABLE_CONFIG,
server.latency_tracking_enabled, 1, NULL, NULL),
    createBoolConfig("aof-disable-auto-gc", NULL, MODIFIABLE_CONFIG,
server.aof_disable_auto_gc, 0, NULL, updateAofAutoGCEnabled),
    createBoolConfig("replica-ignore-disk-write-errors", NULL,
MODIFIABLE_CONFIG, server.repl_ignore_disk_write_error, 0, NULL, NULL),

/* String Configs */
    createStringConfig("aclfile", NULL, IMMUTABLE_CONFIG, ALLOW_EMPTY_STRING,
server.acl_filename, "", NULL, NULL),
    createStringConfig("unixsocket", NULL, IMMUTABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.unixsocket, NULL, NULL, NULL),
    createStringConfig("pidfile", NULL, IMMUTABLE_CONFIG, EMPTY_STRING_IS_NULL,
server.pidfile, NULL, NULL, NULL),

```



```

    createStringConfig("replica-announce-ip", "slave-announce-ip",
MODIFIABLE_CONFIG, EMPTY_STRING_IS_NULL, server.slave_announce_ip, NULL, NULL,
NULL),
    createStringConfig("masteruser", NULL, MODIFIABLE_CONFIG |
SENSITIVE_CONFIG, EMPTY_STRING_IS_NULL, server.masteruser, NULL, NULL, NULL),
    createStringConfig("cluster-announce-ip", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.cluster_announce_ip, NULL, NULL, updateClusterIp),
    createStringConfig("cluster-config-file", NULL, IMMUTABLE_CONFIG,
ALLOW_EMPTY_STRING, server.cluster_configfile, "nodes.conf", NULL, NULL),
    createStringConfig("cluster-announce-hostname", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.cluster_announce_hostname, NULL,
isValidAnnouncedHostname, updateClusterHostname),
    createStringConfig("syslog-ident", NULL, IMMUTABLE_CONFIG,
ALLOW_EMPTY_STRING, server.syslog_ident, "redis", NULL, NULL),
    createStringConfig("dbfilename", NULL, MODIFIABLE_CONFIG |
PROTECTED_CONFIG, ALLOW_EMPTY_STRING, server.rdb_filename, "dump.rdb",
isValidDBfilename, NULL),
    createStringConfig("appendfilename", NULL, IMMUTABLE_CONFIG,
ALLOW_EMPTY_STRING, server.aof_filename, "appendonly.aof", isValidAOFfilename,
NULL),
    createStringConfig("appenddirname", NULL, IMMUTABLE_CONFIG,
ALLOW_EMPTY_STRING, server.aof_dirname, "appendonlydir", isValidAOFdirname,
NULL),
    createStringConfig("server_cpulist", NULL, IMMUTABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.server_cpulist, NULL, NULL, NULL),
    createStringConfig("bio_cpulist", NULL, IMMUTABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.bio_cpulist, NULL, NULL, NULL),
    createStringConfig("aof_rewrite_cpulist", NULL, IMMUTABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.aof_rewrite_cpulist, NULL, NULL, NULL),
    createStringConfig("bgsave_cpulist", NULL, IMMUTABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.bgsave_cpulist, NULL, NULL, NULL),
    createStringConfig("ignore-warnings", NULL, MODIFIABLE_CONFIG,
ALLOW_EMPTY_STRING, server.ignore_warnings, "", NULL, NULL),
    createStringConfig("proc-title-template", NULL, MODIFIABLE_CONFIG,
ALLOW_EMPTY_STRING, server.proc_title_template,
CONFIG_DEFAULT_PROC_TITLE_TEMPLATE, isValidProcTitleTemplate,
updateProcTitleTemplate),
    createStringConfig("bind-source-addr", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.bind_source_addr, NULL, NULL, NULL),
    createStringConfig("logfile", NULL, IMMUTABLE_CONFIG, ALLOW_EMPTY_STRING,
server.logfile, "", NULL, NULL),

/* SDS Configs */
    createSDSConfig("masterauth", NULL, MODIFIABLE_CONFIG | SENSITIVE_CONFIG,
EMPTY_STRING_IS_NULL, server.masterauth, NULL, NULL, NULL),
    createSDSConfig("requirepass", NULL, MODIFIABLE_CONFIG | SENSITIVE_CONFIG,
EMPTY_STRING_IS_NULL, server.requirepass, NULL, NULL, updateRequirePass),

/* Enum Configs */
    createEnumConfig("supervised", NULL, IMMUTABLE_CONFIG,
supervised_mode_enum, server.supervised_mode, SUPERVISED_NONE, NULL, NULL),

```

```

    createEnumConfig("syslog-facility", NULL, IMMUTABLE_CONFIG,
syslog_facility_enum, server.syslog_facility, LOG_LOCAL0, NULL, NULL),
    createEnumConfig("repl-diskless-load", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG | DENY_LOADING_CONFIG, repl_diskless_load_enum,
server.repl_diskless_load, REPL_DISKLESS_LOAD_DISABLED, NULL, NULL),
    createEnumConfig("loglevel", NULL, MODIFIABLE_CONFIG, loglevel_enum,
server.verbosity, LL_NOTICE, NULL, NULL),
    createEnumConfig("maxmemory-policy", NULL, MODIFIABLE_CONFIG,
maxmemory_policy_enum, server.maxmemory_policy, MAXMEMORY_NO_EVICTION, NULL,
NULL),
    createEnumConfig("appendfsync", NULL, MODIFIABLE_CONFIG, aof_fsync_enum,
server.aof_fsync, AOF_FSYNC_EVERYSEC, NULL, NULL),
    createEnumConfig("oom-score-adj", NULL, MODIFIABLE_CONFIG,
oom_score_adj_enum, server.oom_score_adj, OOM_SCORE_ADJ_NO, NULL,
updateOOMScoreAdj),
    createEnumConfig("acl-pubsub-default", NULL, MODIFIABLE_CONFIG,
acl_pubsub_default_enum, server.acl_pubsub_default, 0, NULL, NULL),
    createEnumConfig("sanitize-dump-payload", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, sanitize_dump_payload_enum, server.sanitize_dump_payload,
SANITIZE_DUMP_NO, NULL, NULL),
    createEnumConfig("enable-protected-configs", NULL, IMMUTABLE_CONFIG,
protected_action_enum, server.enable_protected_configs,
PROTECTED_ACTION_ALLOWED_NO, NULL, NULL),
    createEnumConfig("enable-debug-command", NULL, IMMUTABLE_CONFIG,
protected_action_enum, server.enable_debug_cmd, PROTECTED_ACTION_ALLOWED_NO,
NULL, NULL),
    createEnumConfig("enable-module-command", NULL, IMMUTABLE_CONFIG,
protected_action_enum, server.enable_module_cmd, PROTECTED_ACTION_ALLOWED_NO,
NULL, NULL),
    createEnumConfig("cluster-preferred-endpoint-type", NULL,
MODIFIABLE_CONFIG, cluster_preferred_endpoint_type_enum,
server.cluster_preferred_endpoint_type, CLUSTER_ENDPOINT_TYPE_IP, NULL, NULL),
    createEnumConfig("propagation-error-behavior", NULL, MODIFIABLE_CONFIG,
propagation_error_behavior_enum, server.propagation_error_behavior,
PROPAGATION_ERR_BEHAVIOR_IGNORE, NULL, NULL),
    createEnumConfig("shutdown-on-sigint", NULL, MODIFIABLE_CONFIG |
MULTI_ARG_CONFIG, shutdown_on_sig_enum, server.shutdown_on_sigint, 0,
isValidShutdownOnSigFlags, NULL),
    createEnumConfig("shutdown-on-sigterm", NULL, MODIFIABLE_CONFIG |
MULTI_ARG_CONFIG, shutdown_on_sig_enum, server.shutdown_on_sigterm, 0,
isValidShutdownOnSigFlags, NULL),

    /* Integer configs */
    createIntConfig("databases", NULL, IMMUTABLE_CONFIG, 1, INT_MAX,
server.dbnum, 16, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("port", NULL, MODIFIABLE_CONFIG, 0, 65535, server.port,
6379, INTEGER_CONFIG, NULL, updatePort), /* TCP port. */
    createIntConfig("io-threads", NULL, DEBUG_CONFIG | IMMUTABLE_CONFIG, 1,
128, server.io_threads_num, 1, INTEGER_CONFIG, NULL, NULL), /* Single threaded
by default */
    createIntConfig("auto-aof-rewrite-percentage", NULL, MODIFIABLE_CONFIG, 0,

```

```

INT_MAX, server.aof_rewrite_perc, 100, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("cluster-replica-validity-factor", "cluster-slave-validity-
factor", MODIFIABLE_CONFIG, 0, INT_MAX, server.cluster_slave_validity_factor,
10, INTEGER_CONFIG, NULL, NULL), /* Slave max data age factor. */
    createIntConfig("list-max-listpack-size", "list-max-ziplist-size",
MODIFIABLE_CONFIG, INT_MIN, INT_MAX, server.list_max_listpack_size, -2,
INTEGER_CONFIG, NULL, NULL),
    createIntConfig("tcp-keepalive", NULL, MODIFIABLE_CONFIG, 0, INT_MAX,
server.tcpkeepalive, 300, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("cluster-migration-barrier", NULL, MODIFIABLE_CONFIG, 0,
INT_MAX, server.cluster_migration_barrier, 1, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("active-defrag-cycle-min", NULL, MODIFIABLE_CONFIG, 1, 99,
server.active_defrag_cycle_min, 1, INTEGER_CONFIG, NULL, NULL), /* Default: 1%
CPU min (at lower threshold) */
    createIntConfig("active-defrag-cycle-max", NULL, MODIFIABLE_CONFIG, 1, 99,
server.active_defrag_cycle_max, 25, INTEGER_CONFIG, NULL, NULL), /* Default:
25% CPU max (at upper threshold) */
    createIntConfig("active-defrag-threshold-lower", NULL, MODIFIABLE_CONFIG,
0, 1000, server.active_defrag_threshold_lower, 10, INTEGER_CONFIG, NULL, NULL),
/* Default: don't defrag when fragmentation is below 10% */
    createIntConfig("active-defrag-threshold-upper", NULL, MODIFIABLE_CONFIG,
0, 1000, server.active_defrag_threshold_upper, 100, INTEGER_CONFIG, NULL,
NULL), /* Default: maximum defrag force at 100% fragmentation */
    createIntConfig("lfu-log-factor", NULL, MODIFIABLE_CONFIG, 0, INT_MAX,
server.lfu_log_factor, 10, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("lfu-decay-time", NULL, MODIFIABLE_CONFIG, 0, INT_MAX,
server.lfu_decay_time, 1, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("replica-priority", "slave-priority", MODIFIABLE_CONFIG, 0,
INT_MAX, server.slave_priority, 100, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("repl-diskless-sync-delay", NULL, MODIFIABLE_CONFIG, 0,
INT_MAX, server.repl_diskless_sync_delay, 5, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("maxmemory-samples", NULL, MODIFIABLE_CONFIG, 1, INT_MAX,
server.maxmemory_samples, 5, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("maxmemory-eviction-tenacity", NULL, MODIFIABLE_CONFIG, 0,
100, server.maxmemory_eviction_tenacity, 10, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("timeout", NULL, MODIFIABLE_CONFIG, 0, INT_MAX,
server.maxidletime, 0, INTEGER_CONFIG, NULL, NULL), /* Default client timeout:
infinite */
    createIntConfig("replica-announce-port", "slave-announce-port",
MODIFIABLE_CONFIG, 0, 65535, server.slave_announce_port, 0, INTEGER_CONFIG,
NULL, NULL),
    createIntConfig("tcp-backlog", NULL, IMMUTABLE_CONFIG, 0, INT_MAX,
server.tcp_backlog, 511, INTEGER_CONFIG, NULL, NULL), /* TCP listen backlog. */
    createIntConfig("cluster-port", NULL, IMMUTABLE_CONFIG, 0, 65535,
server.cluster_port, 0, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("cluster-announce-bus-port", NULL, MODIFIABLE_CONFIG, 0,
65535, server.cluster_announce_bus_port, 0, INTEGER_CONFIG, NULL, NULL), /*
Default: Use +10000 offset. */
    createIntConfig("cluster-announce-port", NULL, MODIFIABLE_CONFIG, 0, 65535,
server.cluster_announce_port, 0, INTEGER_CONFIG, NULL, NULL), /* Use
server.port */

```

```

    createIntConfig("cluster-announce-tls-port", NULL, MODIFIABLE_CONFIG, 0,
65535, server.cluster_announce_tls_port, 0, INTEGER_CONFIG, NULL, NULL), /* Use
server.tls_port */
    createIntConfig("repl-timeout", NULL, MODIFIABLE_CONFIG, 1, INT_MAX,
server.repl_timeout, 60, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("repl-ping-replica-period", "repl-ping-slave-period",
MODIFIABLE_CONFIG, 1, INT_MAX, server.repl_ping_slave_period, 10,
INTEGER_CONFIG, NULL, NULL),
    createIntConfig("list-compress-depth", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, 0, INT_MAX, server.list_compress_depth, 0, INTEGER_CONFIG,
NULL, NULL),
    createIntConfig("rdb-key-save-delay", NULL, MODIFIABLE_CONFIG |
HIDDEN_CONFIG, INT_MIN, INT_MAX, server.rdb_key_save_delay, 0, INTEGER_CONFIG,
NULL, NULL),
    createIntConfig("key-load-delay", NULL, MODIFIABLE_CONFIG | HIDDEN_CONFIG,
INT_MIN, INT_MAX, server.key_load_delay, 0, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("active-expire-effort", NULL, MODIFIABLE_CONFIG, 1, 10,
server.active_expire_effort, 1, INTEGER_CONFIG, NULL, NULL), /* From 1 to 10.
*/
    createIntConfig("hz", NULL, MODIFIABLE_CONFIG, 0, INT_MAX,
server.config_hz, CONFIG_DEFAULT_HZ, INTEGER_CONFIG, NULL, updateHZ),
    createIntConfig("min-replicas-to-write", "min-slaves-to-write",
MODIFIABLE_CONFIG, 0, INT_MAX, server.repl_min_slaves_to_write, 0,
INTEGER_CONFIG, NULL, updateGoodSlaves),
    createIntConfig("min-replicas-max-lag", "min-slaves-max-lag",
MODIFIABLE_CONFIG, 0, INT_MAX, server.repl_min_slaves_max_lag, 10,
INTEGER_CONFIG, NULL, updateGoodSlaves),
    createIntConfig("watchdog-period", NULL, MODIFIABLE_CONFIG | HIDDEN_CONFIG,
0, INT_MAX, server.watchdog_period, 0, INTEGER_CONFIG, NULL,
updateWatchdogPeriod),
    createIntConfig("shutdown-timeout", NULL, MODIFIABLE_CONFIG, 0, INT_MAX,
server.shutdown_timeout, 10, INTEGER_CONFIG, NULL, NULL),
    createIntConfig("repl-diskless-sync-max-replicas", NULL, MODIFIABLE_CONFIG,
0, INT_MAX, server.repl_diskless_sync_max_replicas, 0, INTEGER_CONFIG, NULL,
NULL),

    /* Unsigned int configs */
    createUIntConfig("maxclients", NULL, MODIFIABLE_CONFIG, 1, UINT_MAX,
server.maxclients, 10000, INTEGER_CONFIG, NULL, updateMaxclients),
    createUIntConfig("unixsocketperm", NULL, IMMUTABLE_CONFIG, 0, 0777,
server.unixsocketperm, 0, OCTAL_CONFIG, NULL, NULL),
    createUIntConfig("socket-mark-id", NULL, IMMUTABLE_CONFIG, 0, UINT_MAX,
server.socket_mark_id, 0, INTEGER_CONFIG, NULL, NULL),

    /* Unsigned Long configs */
    createULongConfig("active-defrag-max-scan-fields", NULL, MODIFIABLE_CONFIG,
1, LONG_MAX, server.active_defrag_max_scan_fields, 1000, INTEGER_CONFIG, NULL,
NULL), /* Default: keys with more than 1000 fields will be processed separately
*/
    createULongConfig("slowlog-max-len", NULL, MODIFIABLE_CONFIG, 0, LONG_MAX,
server.slowlog_max_len, 128, INTEGER_CONFIG, NULL, NULL),

```

```

    createUlongConfig("aclog-max-len", NULL, MODIFIABLE_CONFIG, 0, LONG_MAX,
server.aclog_max_len, 128, INTEGER_CONFIG, NULL, NULL),

    /* Long Long configs */
    createLongLongConfig("busy-reply-threshold", "lua-time-limit",
MODIFIABLE_CONFIG, 0, LONG_MAX, server.busy_reply_threshold, 5000,
INTEGER_CONFIG, NULL, NULL), /* milliseconds */
    createLongLongConfig("cluster-node-timeout", NULL, MODIFIABLE_CONFIG, 0,
LLONG_MAX, server.cluster_node_timeout, 15000, INTEGER_CONFIG, NULL, NULL),
    createLongLongConfig("slowlog-log-slower-than", NULL, MODIFIABLE_CONFIG,
-1, LLONG_MAX, server.slowlog_log_slower_than, 10000, INTEGER_CONFIG, NULL,
NULL),
    createLongLongConfig("latency-monitor-threshold", NULL, MODIFIABLE_CONFIG,
0, LLONG_MAX, server.latency_monitor_threshold, 0, INTEGER_CONFIG, NULL, NULL),
    createLongLongConfig("proto-max-bulk-len", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, 1024*1024, LONG_MAX, server.proto_max_bulk_len,
512ll*1024*1024, MEMORY_CONFIG, NULL, NULL), /* Bulk request max size */
    createLongLongConfig("stream-node-max-entries", NULL, MODIFIABLE_CONFIG, 0,
LLONG_MAX, server.stream_node_max_entries, 100, INTEGER_CONFIG, NULL, NULL),
    createLongLongConfig("repl-backlog-size", NULL, MODIFIABLE_CONFIG, 1,
LLONG_MAX, server.repl_backlog_size, 1024*1024, MEMORY_CONFIG, NULL,
updateReplBacklogSize), /* Default: 1mb */

    /* Unsigned Long Long configs */
    createUlongLongConfig("maxmemory", NULL, MODIFIABLE_CONFIG, 0, ULLONG_MAX,
server.maxmemory, 0, MEMORY_CONFIG, NULL, updateMaxmemory),
    createUlongLongConfig("cluster-link-sendbuf-limit", NULL,
MODIFIABLE_CONFIG, 0, ULLONG_MAX, server.cluster_link_sendbuf_limit_bytes, 0,
MEMORY_CONFIG, NULL, NULL),

    /* Size_t configs */
    createSizeTConfig("hash-max-listpack-entries", "hash-max-ziplist-entries",
MODIFIABLE_CONFIG, 0, LONG_MAX, server.hash_max_listpack_entries, 512,
INTEGER_CONFIG, NULL, NULL),
    createSizeTConfig("set-max-intset-entries", NULL, MODIFIABLE_CONFIG, 0,
LONG_MAX, server.set_max_intset_entries, 512, INTEGER_CONFIG, NULL, NULL),
    createSizeTConfig("zset-max-listpack-entries", "zset-max-ziplist-entries",
MODIFIABLE_CONFIG, 0, LONG_MAX, server.zset_max_listpack_entries, 128,
INTEGER_CONFIG, NULL, NULL),
    createSizeTConfig("active-defrag-ignore-bytes", NULL, MODIFIABLE_CONFIG, 1,
LLONG_MAX, server.active_defrag_ignore_bytes, 100<20, MEMORY_CONFIG, NULL,
NULL), /* Default: don't defrag if frag overhead is below 100mb */
    createSizeTConfig("hash-max-listpack-value", "hash-max-ziplist-value",
MODIFIABLE_CONFIG, 0, LONG_MAX, server.hash_max_listpack_value, 64,
MEMORY_CONFIG, NULL, NULL),
    createSizeTConfig("stream-node-max-bytes", NULL, MODIFIABLE_CONFIG, 0,
LONG_MAX, server.stream_node_max_bytes, 4096, MEMORY_CONFIG, NULL, NULL),
    createSizeTConfig("zset-max-listpack-value", "zset-max-ziplist-value",
MODIFIABLE_CONFIG, 0, LONG_MAX, server.zset_max_listpack_value, 64,
MEMORY_CONFIG, NULL, NULL),
    createSizeTConfig("hll-sparse-max-bytes", NULL, MODIFIABLE_CONFIG, 0,

```



```

LONG_MAX, server.hll_sparse_max_bytes, 3000, MEMORY_CONFIG, NULL, NULL),
    createSizeTConfig("tracking-table-max-keys", NULL, MODIFIABLE_CONFIG, 0,
LONG_MAX, server.tracking_table_max_keys, 1000000, INTEGER_CONFIG, NULL, NULL),
/* Default: 1 million keys max. */
    createSizeTConfig("client-query-buffer-limit", NULL, DEBUG_CONFIG |
MODIFIABLE_CONFIG, 1024*1024, LONG_MAX, server.client_max_querybuf_len,
1024*1024*1024, MEMORY_CONFIG, NULL, NULL), /* Default: 1GB max query buffer.
*/
    createSSizeTConfig("maxmemory-clients", NULL, MODIFIABLE_CONFIG, -100,
SSIZE_MAX, server.maxmemory_clients, 0, MEMORY_CONFIG | PERCENT_CONFIG, NULL,
NULL),

    /* Other configs */
    createTimeTConfig("repl-backlog-ttl", NULL, MODIFIABLE_CONFIG, 0, LONG_MAX,
server.repl_backlog_time_limit, 60*60, INTEGER_CONFIG, NULL, NULL), /* Default:
1 hour */
    createOfftConfig("auto-aof-rewrite-min-size", NULL, MODIFIABLE_CONFIG, 0,
LLONG_MAX, server.aof_rewrite_min_size, 64*1024*1024, MEMORY_CONFIG, NULL,
NULL),
    createOfftConfig("loading-process-events-interval-bytes", NULL,
MODIFIABLE_CONFIG | HIDDEN_CONFIG, 1024, INT_MAX,
server.loading_process_events_interval_bytes, 1024*1024*2, INTEGER_CONFIG,
NULL, NULL),

#ifdef USE_OPENSSL
    createIntConfig("tls-port", NULL, MODIFIABLE_CONFIG, 0, 65535,
server.tls_port, 0, INTEGER_CONFIG, NULL, applyTlsPort), /* TCP port. */
    createIntConfig("tls-session-cache-size", NULL, MODIFIABLE_CONFIG, 0,
INT_MAX, server.tls_ctx_config.session_cache_size, 20*1024, INTEGER_CONFIG,
NULL, applyTlsCfg),
    createIntConfig("tls-session-cache-timeout", NULL, MODIFIABLE_CONFIG, 0,
INT_MAX, server.tls_ctx_config.session_cache_timeout, 300, INTEGER_CONFIG,
NULL, applyTlsCfg),
    createBoolConfig("tls-cluster", NULL, MODIFIABLE_CONFIG,
server.tls_cluster, 0, NULL, applyTlsCfg),
    createBoolConfig("tls-replication", NULL, MODIFIABLE_CONFIG,
server.tls_replication, 0, NULL, applyTlsCfg),
    createEnumConfig("tls-auth-clients", NULL, MODIFIABLE_CONFIG,
tls_auth_clients_enum, server.tls_auth_clients, TLS_CLIENT_AUTH_YES, NULL,
NULL),
    createBoolConfig("tls-prefer-server-ciphers", NULL, MODIFIABLE_CONFIG,
server.tls_ctx_config.prefer_server_ciphers, 0, NULL, applyTlsCfg),
    createBoolConfig("tls-session-caching", NULL, MODIFIABLE_CONFIG,
server.tls_ctx_config.session_caching, 1, NULL, applyTlsCfg),
    createStringConfig("tls-cert-file", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.cert_file, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-key-file", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.key_file, NULL, NULL, applyTlsCfg),
    createStringConfig("tls-key-file-pass", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.key_file_pass, NULL, NULL,

```

```

applyTlsCfg),
    createStringConfig("tls-client-cert-file", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.client_cert_file, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-client-key-file", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.client_key_file, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-client-key-file-pass", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.client_key_file_pass, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-dh-params-file", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.dh_params_file, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-ca-cert-file", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.ca_cert_file, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-ca-cert-dir", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.ca_cert_dir, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-protocols", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.protocols, NULL, NULL,
applyTlsCfg),
    createStringConfig("tls-ciphers", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.ciphers, NULL, NULL, applyTlsCfg),
    createStringConfig("tls-ciphersuites", NULL, MODIFIABLE_CONFIG,
EMPTY_STRING_IS_NULL, server.tls_ctx_config.ciphersuites, NULL, NULL,
applyTlsCfg),
#endif

/* Special configs */
createSpecialConfig("dir", NULL, MODIFIABLE_CONFIG | PROTECTED_CONFIG |
DENY_LOADING_CONFIG, setConfigDirOption, getConfigDirOption,
rewriteConfigDirOption, NULL),
createSpecialConfig("save", NULL, MODIFIABLE_CONFIG | MULTI_ARG_CONFIG,
setConfigSaveOption, getConfigSaveOption, rewriteConfigSaveOption, NULL),
createSpecialConfig("client-output-buffer-limit", NULL, MODIFIABLE_CONFIG |
MULTI_ARG_CONFIG, setConfigClientOutputBufferLimitOption,
getConfigClientOutputBufferLimitOption,
rewriteConfigClientOutputBufferLimitOption, NULL),
createSpecialConfig("oom-score-adj-values", NULL, MODIFIABLE_CONFIG |
MULTI_ARG_CONFIG, setConfigOOMScoreAdjValuesOption,
getConfigOOMScoreAdjValuesOption, rewriteConfigOOMScoreAdjValuesOption,
updateOOMScoreAdj),
createSpecialConfig("notify-keyspace-events", NULL, MODIFIABLE_CONFIG,
setConfigNotifyKeyspaceEventsOption, getConfigNotifyKeyspaceEventsOption,
rewriteConfigNotifyKeyspaceEventsOption, NULL),
createSpecialConfig("bind", NULL, MODIFIABLE_CONFIG | MULTI_ARG_CONFIG,
setConfigBindOption, getConfigBindOption, rewriteConfigBindOption, applyBind),
createSpecialConfig("replicaof", "slaveof", IMMUTABLE_CONFIG |
MULTI_ARG_CONFIG, setConfigReplicaOfOption, getConfigReplicaOfOption,
rewriteConfigReplicaOfOption, NULL),

```

```

    createSpecialConfig("latency-tracking-info-percentiles", NULL,
MODIFIABLE_CONFIG | MULTI_ARG_CONFIG,
setConfigLatencyTrackingInfoPercentilesOutputOption,
getConfigLatencyTrackingInfoPercentilesOutputOption,
rewriteConfigLatencyTrackingInfoPercentilesOutputOption, NULL),

    /* NULL Terminator, this is dropped when we convert to the runtime array.
*/
    {NULL}
};

/* Create a new config by copying the passed in config. Returns 1 on success
 * or 0 when there was already a config with the same name.. */
int registerConfigValue(const char *name, const standardConfig *config, int
alias) {
    standardConfig *new = zmalloc(sizeof(standardConfig));
    memcpy(new, config, sizeof(standardConfig));
    if (alias) {
        new->flags |= ALIAS_CONFIG;
        new->name = config->alias;
        new->alias = config->name;
    }

    return dictAdd(configs, sdsnew(name), new) == DICT_OK;
}

/* Initialize configs to their default values and create and populate the
 * runtime configuration dictionary. */
void initConfigValues() {
    configs = dictCreate(&sdsHashDictType);
    dictExpand(configs, sizeof(static_configs) / sizeof(standardConfig));
    for (standardConfig *config = static_configs; config->name != NULL;
config++) {
        if (config->interface.init) config->interface.init(config);
        /* Add the primary config to the dictionary. */
        int ret = registerConfigValue(config->name, config, 0);
        serverAssert(ret);

        /* Aliases are the same as their primary counter parts, but they
         * also have a flag indicating they are the alias. */
        if (config->alias) {
            int ret = registerConfigValue(config->alias, config, ALIAS_CONFIG);
            serverAssert(ret);
        }
    }
}

/* Remove a config by name from the configs dict. */
void removeConfig(sds name) {
    standardConfig *config = lookupConfig(name);
    if (!config) return;

```



```

    if (config->flags & MODULE_CONFIG) {
        sdsfree((sds) config->name);
        if (config->type == ENUM_CONFIG) {
            configEnum *enumNode = config->data.enumd.enum_value;
            while(enumNode->name != NULL) {
                zfree(enumNode->name);
                enumNode++;
            }
            zfree(config->data.enumd.enum_value);
        } else if (config->type == SDS_CONFIG) {
            if (config->data.sds.default_value) sdsfree((sds)config->
data.sds.default_value);
        }
    }
    dictDelete(configs, name);
}

/*-----
 * Module Config
 *-----
*/

/* Create a bool/string/enum/numeric standardConfig for a module config in the
configs dictionary */
void addModuleBoolConfig(const char *module_name, const char *name, int flags,
void *privdata, int default_val) {
    sds config_name = sdscatfmt(sdsempty(), "%s.%s", module_name, name);
    int config_dummy_address;
    standardConfig module_config = createBoolConfig(config_name, NULL, flags |
MODULE_CONFIG, config_dummy_address, default_val, NULL, NULL);
    module_config.data.yesno.config = NULL;
    module_config.privdata = privdata;
    registerConfigValue(config_name, &module_config, 0);
}

void addModuleStringConfig(const char *module_name, const char *name, int
flags, void *privdata, sds default_val) {
    sds config_name = sdscatfmt(sdsempty(), "%s.%s", module_name, name);
    sds config_dummy_address;
    standardConfig module_config = createSDSConfig(config_name, NULL, flags |
MODULE_CONFIG, 0, config_dummy_address, default_val, NULL, NULL);
    module_config.data.sds.config = NULL;
    module_config.privdata = privdata;
    registerConfigValue(config_name, &module_config, 0);
}

void addModuleEnumConfig(const char *module_name, const char *name, int flags,
void *privdata, int default_val, configEnum *enum_vals) {
    sds config_name = sdscatfmt(sdsempty(), "%s.%s", module_name, name);
    int config_dummy_address;
    standardConfig module_config = createEnumConfig(config_name, NULL, flags |

```

```

MODULE_CONFIG, enum_vals, config_dummy_address, default_val, NULL, NULL);
    module_config.data.enumd.config = NULL;
    module_config.privdata = privdata;
    registerConfigValue(config_name, &module_config, 0);
}

void addModuleNumericConfig(const char *module_name, const char *name, int
flags, void *privdata, long long default_val, int conf_flags, long long lower,
long long upper) {
    sds config_name = sdscatfmt(sdsempty(), "%s.%s", module_name, name);
    long long config_dummy_address;
    standardConfig module_config = createLongLongConfig(config_name, NULL,
flags | MODULE_CONFIG, lower, upper, config_dummy_address, default_val,
conf_flags, NULL, NULL);
    module_config.data.numeric.config.ll = NULL;
    module_config.privdata = privdata;
    registerConfigValue(config_name, &module_config, 0);
}

/*-----
 * CONFIG HELP
 *-----
*/

void configHelpCommand(client *c) {
    const char *help[] = {
"GET <pattern>",
"    Return parameters matching the glob-like <pattern> and their values.",
"SET <directive> <value>",
"    Set the configuration <directive> to <value>.",
"RESETSTAT",
"    Reset statistics reported by the INFO command.",
"REWRITE",
"    Rewrite the configuration file.",
NULL
    };

    addReplyHelp(c, help);
}

/*-----
 * CONFIG RESETSTAT
 *-----
*/

void configResetStatCommand(client *c) {
    resetServerStats();
    resetCommandTableStats(server.commands);
    resetErrorTableStats();
    addReply(c, shared.ok);
}

```

```

/*-----
 * CONFIG REWRITE
 *-----
*/

void configRewriteCommand(client *c) {
    if (server.configfile == NULL) {
        addReplyError(c,"The server is running without a config file");
        return;
    }
    if (rewriteConfig(server.configfile, 0) == -1) {
        /* save errno in case of being tainted. */
        int err = errno;
        serverLog(LL_WARNING,"CONFIG REWRITE failed: %s", strerror(err));
        addReplyErrorFormat(c,"Rewriting config file: %s", strerror(err));
    } else {
        serverLog(LL_WARNING,"CONFIG REWRITE executed with success.");
        addReply(c,shared.ok);
    }
}

```