../raft/CHANGELOG.md
../raft/README.md
../raft/api.go
../raft/commands.go
../raft/commitment.go
../raft/commitment_test.go
../raft/config.go
../raft/configuration.go
../raft/configuration_test.go
../raft/discard_snapshot.go
../raft/discard_snapshot_test.go
../raft/file_snapshot.go
../raft/file_snapshot_test.go
../raft/fsm.go
../raft/future.go
../raft/future_test.go
../raft/inmem_snapshot.go
../raft/inmem_snapshot_test.go
../raft/inmem_store.go
../raft/inmem_transport.go
../raft/inmem_transport_test.go
../raft/integ_test.go
../raft/log.go
../raft/log_cache.go
../raft/log_cache_test.go
../raft/membership.md
../raft/net_transport.go
../raft/net_transport_test.go
../raft/observer.go
../raft/peersjson.go
../raft/peersjson_test.go
../raft/raft.go
../raft/raft_test.go
../raft/replication.go
../raft/snapshot.go
../raft/stable.go
../raft/state.go
../raft/tcp_transport.go
../raft/tcp_transport_test.go
../raft/testing.go

../raft/testing_batch.go
../raft/transport.go
../raft/transport_test.go
../raft/util.go
../raft/util_test.go
../raft/bench/bench.go
../raft/fuzzy/apply_src.go
../raft/fuzzy/cluster.go
../raft/fuzzy/fsm.go
../raft/fuzzy/fsm_batch.go
../raft/fuzzy/leadershiptransfer_test.go
../raft/fuzzy/membership_test.go
../raft/fuzzy/node.go
../raft/fuzzy/partition_test.go
../raft/fuzzy/readme.md
../raft/fuzzy/resolve.go
../raft/fuzzy/simple_test.go
../raft/fuzzy/slowvoter_test.go
../raft/fuzzy/transport.go
../raft/fuzzy/verifier.go

../raft/CHANGELOG.md

UNRELEASED

FEATURES

- Improve FSM apply performance through batching. Implementing the `BatchingFSM` interface enables this new feature [GH-364 (https://github.com/hashicorp/raft/pull/364)]
- Add ability to obtain Raft configuration before Raft starts with GetConfiguration [GH-369 (https://github.com/hashicorp/raft/pull/369)]

IMPROVEMENTS

- Remove lint violations and add a `make` rule for running the linter.
- Replace logger with hclog [GH-360 (https://github.com/hashicorp/raft/pull/360)]

BUG FIXES

- Export the leader field in LeaderObservation [GH-357 (https://github.com/hashicorp/raft/pull/357)]
- Fix snapshot to not attempt to truncate a negative range [GH-358 (https://github.com/hashicorp/raft/pull/358)]

1.1.1 (July 23rd, 2019)

FEATURES

- Add support for extensions to be sent on log entries [GH-353 (https://github.com/hashicorp/raft/pull/353)]
- Add config option to skip snapshot restore on startup [GH-340 (https://github.com/hashicorp/raft/pull/340)]
- Add optional configuration store interface [GH-339 (https://github.com/hashicorp/raft/pull/339)]

IMPROVEMENTS

- Break out of group commit early when no logs are present [GH-341 (https://github.com/hashicorp/raft/pull/341)]

BUGFIXES

- Fix 64-bit counters on 32-bit platforms [GH-344 (https://github.com/hashicorp/raft/pull/344)]
- Don't defer closing source in recover/restore operations since it's in a loop [GH-337 (https://github.com/hashicorp/raft/pull/337)]

1.1.0 (May 23rd, 2019)

FEATURES

- Add transfer leadership extension [GH-306 (https://github.com/hashicorp/raft/pull/306)]

IMPROVEMENTS

- Move to `go mod` [GH-323 (https://github.com/hashicorp/consul/pull/323)]
- Leveled log [GH-321 (https://github.com/hashicorp/consul/pull/321)]
- Add peer changes to observations [GH-326 (https://github.com/hashicorp/consul/pull/326)]

BUGFIXES

- Copy the contents of an InmemSnapshotStore when opening a snapshot [GH-270 (https://github.com/hashicorp/consul/pull/270)]
- Fix logging panic when converting parameters to strings [GH-332 (https://github.com/hashicorp/consul/pull/332)]

1.0.1 (April 12th, 2019)

IMPROVEMENTS

- InMemTransport: Add timeout for sending a message [GH-313 (https://github.com/hashicorp/raft/pull/313)]
- ensure 'make deps' downloads test dependencies like testify [GH-310 (https://github.com/hashicorp/raft/pull/310)]
- Clarifies function of CommitTimeout [GH-309 (https://github.com/hashicorp/raft/pull/309)]
- Add additional metrics regarding log dispatching and committal [GH-316 (https://github.com/hashicorp/raft/pull/316)]

1.0.0 (October 3rd, 2017)

v1.0.0 takes the changes that were staged in the library-v2-stage-one branch. This version manages server identities using a UUID, so introduces some breaking API changes. It also versions the Raft protocol, and requires some special steps when interoperating with Raft servers running older versions of the library (see the detailed comment in config.go about version compatibility). You can reference https://github.com/hashicorp/consul/pull/2222 for an idea of what was required to port Consul to these new interfaces.

0.1.0 (September 29th, 2017)

v0.1.0 is the original stable version of the library that was in master and has been maintained with no breaking API changes. This was in use by Consul prior to version 0.7.0.

../raft/README.md

raft build failing (https://travis-ci.org/hashicorp/raft) FAILED (https://circleci.com/gh/hashicorp/raft)

raft is a Go (http://www.golang.org) library that manages a replicated log and can be used with an FSM to manage replicated state machines. It is a library for providing consensus (http://en.wikipedia.org/wiki/Consensus_(computer_science)).

The use cases for such a library are far-reaching, such as replicated state machines which are a key component of many distributed systems. They enable building Consistent, Partition Tolerant (CP) systems, with limited fault tolerance as well.

Building

If you wish to build raft you'll need Go version 1.2+ installed.

Please check your installation with:

```
go version
```

Documentation

For complete documentation, see the associated Godoc (http://godoc.org/github.com/hashicorp/raft).

To prevent complications with cgo, the primary backend `MDBStore` is in a separate repository, called raft-mdb (http://github.com/hashicorp/raft-mdb). That is the recommended implementation for the `LogStore` and `StableStore`.

A pure Go backend using BoltDB (https://github.com/boltdb/bolt) is also available called raft-boltdb (https://github.com/hashicorp/raft-boltdb). It can also be used as a `LogStore` and `StableStore`.

Tagged Releases

As of September 2017, HashiCorp will start using tags for this library to clearly indicate major version updates. We recommend you vendor your application's dependency on this library.

- v0.1.0 is the original stable version of the library that was in master and has been maintained with no breaking API changes. This was in use by Consul prior to version 0.7.0.

- v1.0.0 takes the changes that were staged in the library-v2-stage-one branch. This version manages server identities using a UUID, so introduces some breaking API changes. It also versions the Raft protocol, and requires some special steps when interoperating with Raft servers running

older versions of the library (see the detailed comment in config.go about version compatibility).
You can reference https://github.com/hashicorp/consul/pull/2222 for an idea of what was required
to port Consul to these new interfaces.

This version includes some new features as well, including non voting servers, a new address
provider abstraction in the transport layer, and more resilient snapshots.

Protocol

raft is based on "Raft: In Search of an Understandable Consensus Algorithm" (https://raft.github.io/raft.pdf)

A high level overview of the Raft protocol is described below, but for details please read the full Raft paper (https://raft.github.io/raft.pdf)
followed by the raft source. Any questions about the raft protocol should be sent to the raft-dev mailing list (https://groups.google.com/forum/#!forum/raft-dev).

Protocol Description

Raft nodes are always in one of three states: follower, candidate or leader. All nodes initially start out as a follower. In this state, nodes can accept log entries from a leader and cast votes. If no entries are received for some time, nodes self-promote to the candidate state. In the candidate state nodes request votes from their peers. If a candidate receives a quorum of votes, then it is promoted to a leader. The leader must accept new log entries and replicate to all the other followers. In addition, if stale reads are not acceptable, all queries must also be performed on the leader.

Once a cluster has a leader, it is able to accept new log entries. A client can request that a leader append a new log entry, which is an opaque binary blob to Raft. The leader then writes the entry to durable storage and attempts to replicate to a quorum of followers. Once the log entry is considered *committed*, it can be *applied* to a finite state machine. The finite state machine is application specific, and is implemented using an interface.

An obvious question relates to the unbounded nature of a replicated log. Raft provides a mechanism by which the current state is snapshotted, and the log is compacted. Because of the FSM abstraction, restoring the state of the FSM must result in the same state as a replay of old logs. This allows Raft to capture the FSM state at a point in time, and then remove all the logs that were used to reach that state. This is performed

automatically
without user intervention, and prevents unbounded disk usage as well as minimizing time spent replaying logs.

Lastly, there is the issue of updating the peer set when new servers are joining or existing servers are leaving. As long as a quorum of nodes is available, this is not an issue as Raft provides mechanisms to dynamically update the peer set. If a quorum of nodes is unavailable, then this becomes a very challenging issue. For example, suppose there are only 2 peers, A and B. The quorum size is also 2, meaning both nodes must agree to commit a log entry. If either A or B fails, it is now impossible to reach quorum. This means the cluster is unable to add, or remove a node, or commit any additional log entries. This results in *unavailability*. At this point, manual intervention would be required to remove either A or B, and to restart the remaining node in bootstrap mode.

A Raft cluster of 3 nodes can tolerate a single node failure, while a cluster of 5 can tolerate 2 node failures. The recommended configuration is to either run 3 or 5 raft servers. This maximizes availability without greatly sacrificing performance.

In terms of performance, Raft is comparable to Paxos. Assuming stable leadership, committing a log entry requires a single round trip to half of the cluster. Thus performance is bound by disk I/O and network latency.

../raft/api.go

```
package raft

import (
    "errors"
    "fmt"
    "io"
    "os"
    "strconv"
    "sync"
    "time"

    metrics "github.com/armon/go-metrics"
    hclog "github.com/hashicorp/go-hclog"
)

const (
    // This is the current suggested max size of the data in a raft log entry.
    // This is based on current architecture, default timing, etc. Clients can
    // ignore this value if they want as there is no actual hard checking
    // within the library. As the library is enhanced this value may change
    // over time to reflect current suggested maximums.
    //
    // Increasing beyond this risks RPC IO taking too long and preventing
    // timely heartbeat signals which are sent in serial in current transports,
    // potentially causing leadership instability.
    SuggestedMaxDataSize = 512 * 1024
)

var (
    // ErrLeader is returned when an operation can't be completed on a
    // leader node.
    ErrLeader = errors.New("node is the leader")

    // ErrNotLeader is returned when an operation can't be completed on a
    // follower or candidate node.
    ErrNotLeader = errors.New("node is not the leader")

    // ErrLeadershipLost is returned when a leader fails to commit a log entry
    // because it's been deposed in the process.
    ErrLeadershipLost = errors.New("leadership lost while committing log")

    // ErrAbortedByRestore is returned when a leader fails to commit a log
    // entry because it's been superseded by a user snapshot restore.
    ErrAbortedByRestore = errors.New("snapshot restored while committing log")

    // ErrRaftShutdown is returned when operations are requested against an
    // inactive Raft.
    ErrRaftShutdown = errors.New("raft is already shutdown")

    // ErrEnqueueTimeout is returned when a command fails due to a timeout.
    ErrEnqueueTimeout = errors.New("timed out enqueuing operation")
```

```
    // ErrNothingNewToSnapshot is returned when trying to create a snapshot
    // but there's nothing new committed to the FSM since we started.
    ErrNothingNewToSnapshot = errors.New("nothing new to snapshot")

    // ErrUnsupportedProtocol is returned when an operation is attempted
    // that's not supported by the current protocol version.
    ErrUnsupportedProtocol = errors.New("operation not supported with current
protocol version")

    // ErrCantBootstrap is returned when attempt is made to bootstrap a
    // cluster that already has state present.
    ErrCantBootstrap = errors.New("bootstrap only works on new clusters")

    // ErrLeadershipTransferInProgress is returned when the leader is rejecting
    // client requests because it is attempting to transfer leadership.
    ErrLeadershipTransferInProgress = errors.New("leadership transfer in
progress")
)

// Raft implements a Raft node.
type Raft struct {
    raftState

    // protocolVersion is used to inter-operate with Raft servers running
    // different versions of the library. See comments in config.go for more
    // details.
    protocolVersion ProtocolVersion

    // applyCh is used to async send logs to the main thread to
    // be committed and applied to the FSM.
    applyCh chan *logFuture

    // Configuration provided at Raft initialization
    conf Config

    // FSM is the client state machine to apply commands to
    fsm FSM

    // fsmMutateCh is used to send state-changing updates to the FSM. This
    // receives pointers to commitTuple structures when applying logs or
    // pointers to restoreFuture structures when restoring a snapshot. We
    // need control over the order of these operations when doing user
    // restores so that we finish applying any old log applies before we
    // take a user snapshot on the leader, otherwise we might restore the
    // snapshot and apply old logs to it that were in the pipe.
    fsmMutateCh chan interface{}

    // fsmSnapshotCh is used to trigger a new snapshot being taken
    fsmSnapshotCh chan *reqSnapshotFuture

    // lastContact is the last time we had contact from the
    // leader node. This can be used to gauge staleness.
    lastContact     time.Time
```

```
    lastContactLock sync.RWMutex

    // Leader is the current cluster leader
    leader     ServerAddress
    leaderLock sync.RWMutex

    // leaderCh is used to notify of leadership changes
    leaderCh chan bool

    // leaderState used only while state is leader
    leaderState leaderState

    // candidateFromLeadershipTransfer is used to indicate that this server
became
    // candidate because the leader tries to transfer leadership. This flag is
    // used in RequestVoteRequest to express that a leadership transfer is
going
    // on.
    candidateFromLeadershipTransfer bool

    // Stores our local server ID, used to avoid sending RPCs to ourself
    localID ServerID

    // Stores our local addr
    localAddr ServerAddress

    // Used for our logging
    logger hclog.Logger

    // LogStore provides durable storage for logs
    logs LogStore

    // Used to request the leader to make configuration changes.
    configurationChangeCh chan *configurationChangeFuture

    // Tracks the latest configuration and latest committed configuration from
    // the log/snapshot.
    configurations configurations

    // RPC chan comes from the transport layer
    rpcCh <-chan RPC

    // Shutdown channel to exit, protected to prevent concurrent exits
    shutdown     bool
    shutdownCh   chan struct{}
    shutdownLock sync.Mutex

    // snapshots is used to store and retrieve snapshots
    snapshots SnapshotStore

    // userSnapshotCh is used for user-triggered snapshots
    userSnapshotCh chan *userSnapshotFuture
```

```
    // userRestoreCh is used for user-triggered restores of external
    // snapshots
    userRestoreCh chan *userRestoreFuture

    // stable is a StableStore implementation for durable state
    // It provides stable storage for many fields in raftState
    stable StableStore

    // The transport layer we use
    trans Transport

    // verifyCh is used to async send verify futures to the main thread
    // to verify we are still the leader
    verifyCh chan *verifyFuture

    // configurationsCh is used to get the configuration data safely from
    // outside of the main thread.
    configurationsCh chan *configurationsFuture

    // bootstrapCh is used to attempt an initial bootstrap from outside of
    // the main thread.
    bootstrapCh chan *bootstrapFuture

    // List of observers and the mutex that protects them. The observers list
    // is indexed by an artificial ID which is used for deregistration.
    observersLock sync.RWMutex
    observers     map[uint64]*Observer

    // leadershipTransferCh is used to start a leadership transfer from outside
of
    // the main thread.
    leadershipTransferCh chan *leadershipTransferFuture
}

// BootstrapCluster initializes a server's storage with the given cluster
// configuration. This should only be called at the beginning of time for the
// cluster with an identical configuration listing all Voter servers. There is
// no need to bootstrap Nonvoter and Staging servers.
//
// A cluster can only be bootstrapped once from a single participating Voter
// server. Any further attempts to bootstrap will return an error that can be
// safely ignored.
//
// One sane approach is to bootstrap a single server with a configuration
// listing just itself as a Voter, then invoke AddVoter() on it to add other
// servers to the cluster.
func BootstrapCluster(conf *Config, logs LogStore, stable StableStore,
    snaps SnapshotStore, trans Transport, configuration Configuration) error {
    // Validate the Raft server config.
    if err := ValidateConfig(conf); err != nil {
        return err
    }
```

```
    // Sanity check the Raft peer configuration.
    if err := checkConfiguration(configuration); err != nil {
        return err
    }

    // Make sure the cluster is in a clean state.
    hasState, err := HasExistingState(logs, stable, snaps)
    if err != nil {
        return fmt.Errorf("failed to check for existing state: %v", err)
    }
    if hasState {
        return ErrCantBootstrap
    }

    // Set current term to 1.
    if err := stable.SetUint64(keyCurrentTerm, 1); err != nil {
        return fmt.Errorf("failed to save current term: %v", err)
    }

    // Append configuration entry to log.
    entry := &Log{
        Index: 1,
        Term:  1,
    }
    if conf.ProtocolVersion < 3 {
        entry.Type = LogRemovePeerDeprecated
        entry.Data = encodePeers(configuration, trans)
    } else {
        entry.Type = LogConfiguration
        entry.Data = EncodeConfiguration(configuration)
    }
    if err := logs.StoreLog(entry); err != nil {
        return fmt.Errorf("failed to append configuration entry to log: %v",
err)
    }

    return nil
}

// RecoverCluster is used to manually force a new configuration in order to
// recover from a loss of quorum where the current configuration cannot be
// restored, such as when several servers die at the same time. This works by
// reading all the current state for this server, creating a snapshot with the
// supplied configuration, and then truncating the Raft log. This is the only
// safe way to force a given configuration without actually altering the log to
// insert any new entries, which could cause conflicts with other servers with
// different state.
//
// WARNING! This operation implicitly commits all entries in the Raft log, so
// in general this is an extremely unsafe operation. If you've lost your other
// servers and are performing a manual recovery, then you've also lost the
// commit information, so this is likely the best you can do, but you should be
// aware that calling this can cause Raft log entries that were in the process
```

```
// of being replicated but not yet be committed to be committed.
//
// Note the FSM passed here is used for the snapshot operations and will be
// left in a state that should not be used by the application. Be sure to
// discard this FSM and any associated state and provide a fresh one when
// calling NewRaft later.
//
// A typical way to recover the cluster is to shut down all servers and then
// run RecoverCluster on every server using an identical configuration. When
// the cluster is then restarted, and election should occur and then Raft will
// resume normal operation. If it's desired to make a particular server the
// leader, this can be used to inject a new configuration with that server as
// the sole voter, and then join up other new clean-state peer servers using
// the usual APIs in order to bring the cluster back into a known state.
func RecoverCluster(conf *Config, fsm FSM, logs LogStore, stable StableStore,
    snaps SnapshotStore, trans Transport, configuration Configuration) error {
    // Validate the Raft server config.
    if err := ValidateConfig(conf); err != nil {
        return err
    }

    // Sanity check the Raft peer configuration.
    if err := checkConfiguration(configuration); err != nil {
        return err
    }

    // Refuse to recover if there's no existing state. This would be safe to
    // do, but it is likely an indication of an operator error where they
    // expect data to be there and it's not. By refusing, we force them
    // to show intent to start a cluster fresh by explicitly doing a
    // bootstrap, rather than quietly fire up a fresh cluster here.
    if hasState, err := HasExistingState(logs, stable, snaps); err != nil {
        return fmt.Errorf("failed to check for existing state: %v", err)
    } else if !hasState {
        return fmt.Errorf("refused to recover cluster with no initial state,
this is probably an operator error")
    }

    // Attempt to restore any snapshots we find, newest to oldest.
    var (
        snapshotIndex  uint64
        snapshotTerm   uint64
        snapshots, err = snaps.List()
    )
    if err != nil {
        return fmt.Errorf("failed to list snapshots: %v", err)
    }
    for _, snapshot := range snapshots {
        var source io.ReadCloser
        _, source, err = snaps.Open(snapshot.ID)
        if err != nil {
            // Skip this one and try the next. We will detect if we
            // couldn't open any snapshots.
```

```
            continue
        }

        err = fsm.Restore(source)
        // Close the source after the restore has completed
        source.Close()
        if err != nil {
            // Same here, skip and try the next one.
            continue
        }

        snapshotIndex = snapshot.Index
        snapshotTerm = snapshot.Term
        break
    }
    if len(snapshots) > 0 && (snapshotIndex == 0 || snapshotTerm == 0) {
        return fmt.Errorf("failed to restore any of the available snapshots")
    }

    // The snapshot information is the best known end point for the data
    // until we play back the Raft log entries.
    lastIndex := snapshotIndex
    lastTerm := snapshotTerm

    // Apply any Raft log entries past the snapshot.
    lastLogIndex, err := logs.LastIndex()
    if err != nil {
        return fmt.Errorf("failed to find last log: %v", err)
    }
    for index := snapshotIndex + 1; index <= lastLogIndex; index++ {
        var entry Log
        if err = logs.GetLog(index, &entry); err != nil {
            return fmt.Errorf("failed to get log at index %d: %v", index, err)
        }
        if entry.Type == LogCommand {
            _ = fsm.Apply(&entry)
        }
        lastIndex = entry.Index
        lastTerm = entry.Term
    }

    // Create a new snapshot, placing the configuration in as if it was
    // committed at index 1.
    snapshot, err := fsm.Snapshot()
    if err != nil {
        return fmt.Errorf("failed to snapshot FSM: %v", err)
    }
    version := getSnapshotVersion(conf.ProtocolVersion)
    sink, err := snaps.Create(version, lastIndex, lastTerm, configuration, 1,
trans)
    if err != nil {
        return fmt.Errorf("failed to create snapshot: %v", err)
    }
```

```
    if err = snapshot.Persist(sink); err != nil {
        return fmt.Errorf("failed to persist snapshot: %v", err)
    }
    if err = sink.Close(); err != nil {
        return fmt.Errorf("failed to finalize snapshot: %v", err)
    }

    // Compact the log so that we don't get bad interference from any
    // configuration change log entries that might be there.
    firstLogIndex, err := logs.FirstIndex()
    if err != nil {
        return fmt.Errorf("failed to get first log index: %v", err)
    }
    if err := logs.DeleteRange(firstLogIndex, lastLogIndex); err != nil {
        return fmt.Errorf("log compaction failed: %v", err)
    }

    return nil
}

// GetConfiguration returns the configuration of the Raft cluster without
// starting a Raft instance or connecting to the cluster
// This function has identical behavior to Raft.GetConfiguration
func GetConfiguration(conf *Config, fsm FSM, logs LogStore, stable StableStore,
    snaps SnapshotStore, trans Transport) (Configuration, error) {
    conf.skipStartup = true
    r, err := NewRaft(conf, fsm, logs, stable, snaps, trans)
    if err != nil {
        return Configuration{}, err
    }
    future := r.GetConfiguration()
    if err = future.Error(); err != nil {
        return Configuration{}, err
    }
    return future.Configuration(), nil
}

// HasExistingState returns true if the server has any existing state (logs,
// knowledge of a current term, or any snapshots).
func HasExistingState(logs LogStore, stable StableStore, snaps SnapshotStore)
(bool, error) {
    // Make sure we don't have a current term.
    currentTerm, err := stable.GetUint64(keyCurrentTerm)
    if err == nil {
        if currentTerm > 0 {
            return true, nil
        }
    } else {
        if err != ErrKeyNotFound {
            return false, fmt.Errorf("failed to read current term: %v", err)
        }
    }
```

```
    // Make sure we have an empty log.
    lastIndex, err := logs.LastIndex()
    if err != nil {
        return false, fmt.Errorf("failed to get last log index: %v", err)
    }
    if lastIndex > 0 {
        return true, nil
    }

    // Make sure we have no snapshots
    snapshots, err := snaps.List()
    if err != nil {
        return false, fmt.Errorf("failed to list snapshots: %v", err)
    }
    if len(snapshots) > 0 {
        return true, nil
    }

    return false, nil
}

// NewRaft is used to construct a new Raft node. It takes a configuration, as
well
// as implementations of various interfaces that are required. If we have any
// old state, such as snapshots, logs, peers, etc, all those will be restored
// when creating the Raft node.
func NewRaft(conf *Config, fsm FSM, logs LogStore, stable StableStore, snaps
SnapshotStore, trans Transport) (*Raft, error) {
    // Validate the configuration.
    if err := ValidateConfig(conf); err != nil {
        return nil, err
    }

    // Ensure we have a LogOutput.
    var logger hclog.Logger
    if conf.Logger != nil {
        logger = conf.Logger
    } else {
        if conf.LogOutput == nil {
            conf.LogOutput = os.Stderr
        }

        logger = hclog.New(&hclog.LoggerOptions{
            Name:   "raft",
            Level:  hclog.LevelFromString(conf.LogLevel),
            Output: conf.LogOutput,
        })
    }

    // Try to restore the current term.
    currentTerm, err := stable.GetUint64(keyCurrentTerm)
    if err != nil && err != ErrKeyNotFound {
        return nil, fmt.Errorf("failed to load current term: %v", err)
```

```
    }

    // Read the index of the last log entry.
    lastIndex, err := logs.LastIndex()
    if err != nil {
        return nil, fmt.Errorf("failed to find last log: %v", err)
    }

    // Get the last log entry.
    var lastLog Log
    if lastIndex > 0 {
        if err = logs.GetLog(lastIndex, &lastLog); err != nil {
            return nil, fmt.Errorf("failed to get last log at index %d: %v",
lastIndex, err)
        }
    }

    // Make sure we have a valid server address and ID.
    protocolVersion := conf.ProtocolVersion
    localAddr := ServerAddress(trans.LocalAddr())
    localID := conf.LocalID

    // TODO (slackpad) - When we deprecate protocol version 2, remove this
    // along with the AddPeer() and RemovePeer() APIs.
    if protocolVersion < 3 && string(localID) != string(localAddr) {
        return nil, fmt.Errorf("when running with ProtocolVersion < 3, LocalID
must be set to the network address")
    }

    // Create Raft struct.
    r := &Raft{
        protocolVersion:       protocolVersion,
        applyCh:               make(chan *logFuture),
        conf:                  *conf,
        fsm:                   fsm,
        fsmMutateCh:           make(chan interface{}, 128),
        fsmSnapshotCh:         make(chan *reqSnapshotFuture),
        leaderCh:              make(chan bool),
        localID:               localID,
        localAddr:             localAddr,
        logger:                logger,
        logs:                  logs,
        configurationChangeCh: make(chan *configurationChangeFuture),
        configurations:        configurations{},
        rpcCh:                 trans.Consumer(),
        snapshots:             snaps,
        userSnapshotCh:        make(chan *userSnapshotFuture),
        userRestoreCh:         make(chan *userRestoreFuture),
        shutdownCh:            make(chan struct{}),
        stable:                stable,
        trans:                 trans,
        verifyCh:              make(chan *verifyFuture, 64),
        configurationsCh:      make(chan *configurationsFuture, 8),
```

```
        bootstrapCh:           make(chan *bootstrapFuture),
        observers:             make(map[uint64]*Observer),
        leadershipTransferCh:  make(chan *leadershipTransferFuture, 1),
    }

    // Initialize as a follower.
    r.setState(Follower)

    // Start as leader if specified. This should only be used
    // for testing purposes.
    if conf.StartAsLeader {
        r.setState(Leader)
        r.setLeader(r.localAddr)
    }

    // Restore the current term and the last log.
    r.setCurrentTerm(currentTerm)
    r.setLastLog(lastLog.Index, lastLog.Term)

    // Attempt to restore a snapshot if there are any.
    if err := r.restoreSnapshot(); err != nil {
        return nil, err
    }

    // Scan through the log for any configuration change entries.
    snapshotIndex, _ := r.getLastSnapshot()
    for index := snapshotIndex + 1; index <= lastLog.Index; index++ {
        var entry Log
        if err := r.logs.GetLog(index, &entry); err != nil {
            r.logger.Error("failed to get log", "index", index, "error", err)
            panic(err)
        }
        r.processConfigurationLogEntry(&entry)
    }
    r.logger.Info("initial configuration",
        "index", r.configurations.latestIndex,
        "servers", hclog.Fmt("%+v", r.configurations.latest.Servers))

    // Setup a heartbeat fast-path to avoid head-of-line
    // blocking where possible. It MUST be safe for this
    // to be called concurrently with a blocking RPC.
    trans.SetHeartbeatHandler(r.processHeartbeat)

    if conf.skipStartup {
        return r, nil
    }
    // Start the background work.
    r.goFunc(r.run)
    r.goFunc(r.runFSM)
    r.goFunc(r.runSnapshots)
    return r, nil
}
```

```
// restoreSnapshot attempts to restore the latest snapshots, and fails if none
// of them can be restored. This is called at initialization time, and is
// completely unsafe to call at any other time.
func (r *Raft) restoreSnapshot() error {
    snapshots, err := r.snapshots.List()
    if err != nil {
        r.logger.Error("failed to list snapshots", "error", err)
        return err
    }

    // Try to load in order of newest to oldest
    for _, snapshot := range snapshots {
        if !r.conf.NoSnapshotRestoreOnStart {
            _, source, err := r.snapshots.Open(snapshot.ID)
            if err != nil {
                r.logger.Error("failed to open snapshot", "id", snapshot.ID,
"error", err)
                continue
            }

            err = r.fsm.Restore(source)
            // Close the source after the restore has completed
            source.Close()
            if err != nil {
                r.logger.Error("failed to restore snapshot", "id", snapshot.ID,
"error", err)
                continue
            }

            r.logger.Info("restored from snapshot", "id", snapshot.ID)
        }
        // Update the lastApplied so we don't replay old logs
        r.setLastApplied(snapshot.Index)

        // Update the last stable snapshot info
        r.setLastSnapshot(snapshot.Index, snapshot.Term)

        // Update the configuration
        if snapshot.Version > 0 {
            r.configurations.committed = snapshot.Configuration
            r.configurations.committedIndex = snapshot.ConfigurationIndex
            r.configurations.latest = snapshot.Configuration
            r.configurations.latestIndex = snapshot.ConfigurationIndex
        } else {
            configuration := decodePeers(snapshot.Peers, r.trans)
            r.configurations.committed = configuration
            r.configurations.committedIndex = snapshot.Index
            r.configurations.latest = configuration
            r.configurations.latestIndex = snapshot.Index
        }

        // Success!
        return nil
```

```
    }

    // If we had snapshots and failed to load them, its an error
    if len(snapshots) > 0 {
        return fmt.Errorf("failed to load any existing snapshots")
    }
    return nil
}

// BootstrapCluster is equivalent to non-member BootstrapCluster but can be
// called on an un-bootstrapped Raft instance after it has been created. This
// should only be called at the beginning of time for the cluster with an
// identical configuration listing all Voter servers. There is no need to
// bootstrap Nonvoter and Staging servers.
//
// A cluster can only be bootstrapped once from a single participating Voter
// server. Any further attempts to bootstrap will return an error that can be
// safely ignored.
//
// One sane approach is to bootstrap a single server with a configuration
// listing just itself as a Voter, then invoke AddVoter() on it to add other
// servers to the cluster.
func (r *Raft) BootstrapCluster(configuration Configuration) Future {
    bootstrapReq := &bootstrapFuture{}
    bootstrapReq.init()
    bootstrapReq.configuration = configuration
    select {
    case <-r.shutdownCh:
        return errorFuture{ErrRaftShutdown}
    case r.bootstrapCh <- bootstrapReq:
        return bootstrapReq
    }
}

// Leader is used to return the current leader of the cluster.
// It may return empty string if there is no current leader
// or the leader is unknown.
func (r *Raft) Leader() ServerAddress {
    r.leaderLock.RLock()
    leader := r.leader
    r.leaderLock.RUnlock()
    return leader
}

// Apply is used to apply a command to the FSM in a highly consistent
// manner. This returns a future that can be used to wait on the application.
// An optional timeout can be provided to limit the amount of time we wait
// for the command to be started. This must be run on the leader or it
// will fail.
func (r *Raft) Apply(cmd []byte, timeout time.Duration) ApplyFuture {
    return r.ApplyLog(Log{Data: cmd}, timeout)
}
```

```
// ApplyLog performs Apply but takes in a Log directly. The only values
// currently taken from the submitted Log are Data and Extensions.
func (r *Raft) ApplyLog(log Log, timeout time.Duration) ApplyFuture {
    metrics.IncrCounter([]string{"raft", "apply"}, 1)

    var timer <-chan time.Time
    if timeout > 0 {
        timer = time.After(timeout)
    }

    // Create a log future, no index or term yet
    logFuture := &logFuture{
        log: Log{
            Type:       LogCommand,
            Data:       log.Data,
            Extensions: log.Extensions,
        },
    }
    logFuture.init()

    select {
    case <-timer:
        return errorFuture{ErrEnqueueTimeout}
    case <-r.shutdownCh:
        return errorFuture{ErrRaftShutdown}
    case r.applyCh <- logFuture:
        return logFuture
    }
}

// Barrier is used to issue a command that blocks until all preceding
// operations have been applied to the FSM. It can be used to ensure the
// FSM reflects all queued writes. An optional timeout can be provided to
// limit the amount of time we wait for the command to be started. This
// must be run on the leader or it will fail.
func (r *Raft) Barrier(timeout time.Duration) Future {
    metrics.IncrCounter([]string{"raft", "barrier"}, 1)
    var timer <-chan time.Time
    if timeout > 0 {
        timer = time.After(timeout)
    }

    // Create a log future, no index or term yet
    logFuture := &logFuture{
        log: Log{
            Type: LogBarrier,
        },
    }
    logFuture.init()

    select {
    case <-timer:
        return errorFuture{ErrEnqueueTimeout}
```

```
    case <-r.shutdownCh:
        return errorFuture{ErrRaftShutdown}
    case r.applyCh <- logFuture:
        return logFuture
    }
}

// VerifyLeader is used to ensure the current node is still
// the leader. This can be done to prevent stale reads when a
// new leader has potentially been elected.
func (r *Raft) VerifyLeader() Future {
    metrics.IncrCounter([]string{"raft", "verify_leader"}, 1)
    verifyFuture := &verifyFuture{}
    verifyFuture.init()
    select {
    case <-r.shutdownCh:
        return errorFuture{ErrRaftShutdown}
    case r.verifyCh <- verifyFuture:
        return verifyFuture
    }
}

// GetConfiguration returns the latest configuration and its associated index
// currently in use. This may not yet be committed. This must not be called on
// the main thread (which can access the information directly).
func (r *Raft) GetConfiguration() ConfigurationFuture {
    configReq := &configurationsFuture{}
    configReq.init()
    select {
    case <-r.shutdownCh:
        configReq.respond(ErrRaftShutdown)
        return configReq
    case r.configurationsCh <- configReq:
        return configReq
    }
}

// AddPeer (deprecated) is used to add a new peer into the cluster. This must
be
// run on the leader or it will fail. Use AddVoter/AddNonvoter instead.
func (r *Raft) AddPeer(peer ServerAddress) Future {
    if r.protocolVersion > 2 {
        return errorFuture{ErrUnsupportedProtocol}
    }

    return r.requestConfigChange(configurationChangeRequest{
        command:       AddStaging,
        serverID:      ServerID(peer),
        serverAddress: peer,
        prevIndex:     0,
    }, 0)
}
```

```
// RemovePeer (deprecated) is used to remove a peer from the cluster. If the
// current leader is being removed, it will cause a new election
// to occur. This must be run on the leader or it will fail.
// Use RemoveServer instead.
func (r *Raft) RemovePeer(peer ServerAddress) Future {
    if r.protocolVersion > 2 {
        return errorFuture{ErrUnsupportedProtocol}
    }

    return r.requestConfigChange(configurationChangeRequest{
        command:   RemoveServer,
        serverID:  ServerID(peer),
        prevIndex: 0,
    }, 0)
}

// AddVoter will add the given server to the cluster as a staging server. If
the
// server is already in the cluster as a voter, this updates the server's
address.
// This must be run on the leader or it will fail. The leader will promote the
// staging server to a voter once that server is ready. If nonzero, prevIndex
is
// the index of the only configuration upon which this change may be applied;
if
// another configuration entry has been added in the meantime, this request
will
// fail. If nonzero, timeout is how long this server should wait before the
// configuration change log entry is appended.
func (r *Raft) AddVoter(id ServerID, address ServerAddress, prevIndex uint64,
timeout time.Duration) IndexFuture {
    if r.protocolVersion < 2 {
        return errorFuture{ErrUnsupportedProtocol}
    }

    return r.requestConfigChange(configurationChangeRequest{
        command:       AddStaging,
        serverID:      id,
        serverAddress: address,
        prevIndex:     prevIndex,
    }, timeout)
}

// AddNonvoter will add the given server to the cluster but won't assign it a
// vote. The server will receive log entries, but it won't participate in
// elections or log entry commitment. If the server is already in the cluster,
// this updates the server's address. This must be run on the leader or it will
// fail. For prevIndex and timeout, see AddVoter.
func (r *Raft) AddNonvoter(id ServerID, address ServerAddress, prevIndex
uint64, timeout time.Duration) IndexFuture {
    if r.protocolVersion < 3 {
        return errorFuture{ErrUnsupportedProtocol}
    }
```

```
    return r.requestConfigChange(configurationChangeRequest{
        command:       AddNonvoter,
        serverID:      id,
        serverAddress: address,
        prevIndex:     prevIndex,
    }, timeout)
}

// RemoveServer will remove the given server from the cluster. If the current
// leader is being removed, it will cause a new election to occur. This must be
// run on the leader or it will fail. For prevIndex and timeout, see AddVoter.
func (r *Raft) RemoveServer(id ServerID, prevIndex uint64, timeout
time.Duration) IndexFuture {
    if r.protocolVersion < 2 {
        return errorFuture{ErrUnsupportedProtocol}
    }

    return r.requestConfigChange(configurationChangeRequest{
        command:   RemoveServer,
        serverID:  id,
        prevIndex: prevIndex,
    }, timeout)
}

// DemoteVoter will take away a server's vote, if it has one. If present, the
// server will continue to receive log entries, but it won't participate in
// elections or log entry commitment. If the server is not in the cluster, this
// does nothing. This must be run on the leader or it will fail. For prevIndex
// and timeout, see AddVoter.
func (r *Raft) DemoteVoter(id ServerID, prevIndex uint64, timeout
time.Duration) IndexFuture {
    if r.protocolVersion < 3 {
        return errorFuture{ErrUnsupportedProtocol}
    }

    return r.requestConfigChange(configurationChangeRequest{
        command:   DemoteVoter,
        serverID:  id,
        prevIndex: prevIndex,
    }, timeout)
}

// Shutdown is used to stop the Raft background routines.
// This is not a graceful operation. Provides a future that
// can be used to block until all background routines have exited.
func (r *Raft) Shutdown() Future {
    r.shutdownLock.Lock()
    defer r.shutdownLock.Unlock()

    if !r.shutdown {
        close(r.shutdownCh)
        r.shutdown = true
```

```
        r.setState(Shutdown)
        return &shutdownFuture{r}
    }

    // avoid closing transport twice
    return &shutdownFuture{nil}
}

// Snapshot is used to manually force Raft to take a snapshot. Returns a future
// that can be used to block until complete, and that contains a function that
// can be used to open the snapshot.
func (r *Raft) Snapshot() SnapshotFuture {
    future := &userSnapshotFuture{}
    future.init()
    select {
    case r.userSnapshotCh <- future:
        return future
    case <-r.shutdownCh:
        future.respond(ErrRaftShutdown)
        return future
    }
}

// Restore is used to manually force Raft to consume an external snapshot, such
// as if restoring from a backup. We will use the current Raft configuration,
// not the one from the snapshot, so that we can restore into a new cluster. We
// will also use the higher of the index of the snapshot, or the current index,
// and then add 1 to that, so we force a new state with a hole in the Raft log,
// so that the snapshot will be sent to followers and used for any new joiners.
// This can only be run on the leader, and blocks until the restore is complete
// or an error occurs.
//
// WARNING! This operation has the leader take on the state of the snapshot and
// then sets itself up so that it replicates that to its followers though the
// install snapshot process. This involves a potentially dangerous period where
// the leader commits ahead of its followers, so should only be used for
disaster
// recovery into a fresh cluster, and should not be used in normal operations.
func (r *Raft) Restore(meta *SnapshotMeta, reader io.Reader, timeout
time.Duration) error {
    metrics.IncrCounter([]string{"raft", "restore"}, 1)
    var timer <-chan time.Time
    if timeout > 0 {
        timer = time.After(timeout)
    }

    // Perform the restore.
    restore := &userRestoreFuture{
        meta:   meta,
        reader: reader,
    }
    restore.init()
    select {
```

```
    case <-timer:
        return ErrEnqueueTimeout
    case <-r.shutdownCh:
        return ErrRaftShutdown
    case r.userRestoreCh <- restore:
        // If the restore is ingested then wait for it to complete.
        if err := restore.Error(); err != nil {
            return err
        }
    }

    // Apply a no-op log entry. Waiting for this allows us to wait until the
    // followers have gotten the restore and replicated at least this new
    // entry, which shows that we've also faulted and installed the
    // snapshot with the contents of the restore.
    noop := &logFuture{
        log: Log{
            Type: LogNoop,
        },
    }
    noop.init()
    select {
    case <-timer:
        return ErrEnqueueTimeout
    case <-r.shutdownCh:
        return ErrRaftShutdown
    case r.applyCh <- noop:
        return noop.Error()
    }
}

// State is used to return the current raft state.
func (r *Raft) State() RaftState {
    return r.getState()
}

// LeaderCh is used to get a channel which delivers signals on
// acquiring or losing leadership. It sends true if we become
// the leader, and false if we lose it. The channel is not buffered,
// and does not block on writes.
func (r *Raft) LeaderCh() <-chan bool {
    return r.leaderCh
}

// String returns a string representation of this Raft node.
func (r *Raft) String() string {
    return fmt.Sprintf("Node at %s [%v]", r.localAddr, r.getState())
}

// LastContact returns the time of last contact by a leader.
// This only makes sense if we are currently a follower.
func (r *Raft) LastContact() time.Time {
    r.lastContactLock.RLock()
```

```
    last := r.lastContact
    r.lastContactLock.RUnlock()
    return last
}

// Stats is used to return a map of various internal stats. This
// should only be used for informative purposes or debugging.
//
// Keys are: "state", "term", "last_log_index", "last_log_term",
// "commit_index", "applied_index", "fsm_pending",
// "last_snapshot_index", "last_snapshot_term",
// "latest_configuration", "last_contact", and "num_peers".
//
// The value of "state" is a numeric constant representing one of
// the possible leadership states the node is in at any given time.
// the possible states are: "Follower", "Candidate", "Leader", "Shutdown".
//
// The value of "latest_configuration" is a string which contains
// the id of each server, its suffrage status, and its address.
//
// The value of "last_contact" is either "never" if there
// has been no contact with a leader, "0" if the node is in the
// leader state, or the time since last contact with a leader
// formatted as a string.
//
// The value of "num_peers" is the number of other voting servers in the
// cluster, not including this node. If this node isn't part of the
// configuration then this will be "0".
//
// All other values are uint64s, formatted as strings.
func (r *Raft) Stats() map[string]string {
    toString := func(v uint64) string {
        return strconv.FormatUint(v, 10)
    }
    lastLogIndex, lastLogTerm := r.getLastLog()
    lastSnapIndex, lastSnapTerm := r.getLastSnapshot()
    s := map[string]string{
        "state":               r.getState().String(),
        "term":                toString(r.getCurrentTerm()),
        "last_log_index":      toString(lastLogIndex),
        "last_log_term":       toString(lastLogTerm),
        "commit_index":        toString(r.getCommitIndex()),
        "applied_index":       toString(r.getLastApplied()),
        "fsm_pending":         toString(uint64(len(r.fsmMutateCh))),
        "last_snapshot_index": toString(lastSnapIndex),
        "last_snapshot_term":  toString(lastSnapTerm),
        "protocol_version":    toString(uint64(r.protocolVersion)),
        "protocol_version_min": toString(uint64(ProtocolVersionMin)),
        "protocol_version_max": toString(uint64(ProtocolVersionMax)),
        "snapshot_version_min": toString(uint64(SnapshotVersionMin)),
        "snapshot_version_max": toString(uint64(SnapshotVersionMax)),
    }
```

```
    future := r.GetConfiguration()
    if err := future.Error(); err != nil {
        r.logger.Warn("could not get configuration for stats", "error", err)
    } else {
        configuration := future.Configuration()
        s["latest_configuration_index"] = toString(future.Index())
        s["latest_configuration"] = fmt.Sprintf("%+v", configuration.Servers)

        // This is a legacy metric that we've seen people use in the wild.
        hasUs := false
        numPeers := 0
        for _, server := range configuration.Servers {
            if server.Suffrage == Voter {
                if server.ID == r.localID {
                    hasUs = true
                } else {
                    numPeers++
                }
            }
        }
        if !hasUs {
            numPeers = 0
        }
        s["num_peers"] = toString(uint64(numPeers))
    }

    last := r.LastContact()
    if r.getState() == Leader {
        s["last_contact"] = "0"
    } else if last.IsZero() {
        s["last_contact"] = "never"
    } else {
        s["last_contact"] = fmt.Sprintf("%v", time.Now().Sub(last))
    }
    return s
}

// LastIndex returns the last index in stable storage,
// either from the last log or from the last snapshot.
func (r *Raft) LastIndex() uint64 {
    return r.getLastIndex()
}

// AppliedIndex returns the last index applied to the FSM. This is generally
// lagging behind the last index, especially for indexes that are persisted but
// have not yet been considered committed by the leader. NOTE - this reflects
// the last index that was sent to the application's FSM over the apply channel
// but DOES NOT mean that the application's FSM has yet consumed it and applied
// it to its internal state. Thus, the application's state may lag behind this
// index.
func (r *Raft) AppliedIndex() uint64 {
    return r.getLastApplied()
}
```

```go
// LeadershipTransfer will transfer leadership to a server in the cluster.
// This can only be called from the leader, or it will fail. The leader will
// stop accepting client requests, make sure the target server is up to date
// and starts the transfer with a TimeoutNow message. This message has the same
// effect as if the election timeout on the on the target server fires. Since
// it is unlikely that another server is starting an election, it is very
// likely that the target server is able to win the election.  Note that raft
// protocol version 3 is not sufficient to use LeadershipTransfer. A recent
// version of that library has to be used that includes this feature.  Using
// transfer leadership is safe however in a cluster where not every node has
// the latest version. If a follower cannot be promoted, it will fail
// gracefully.
func (r *Raft) LeadershipTransfer() Future {
    if r.protocolVersion < 3 {
        return errorFuture{ErrUnsupportedProtocol}
    }

    return r.initiateLeadershipTransfer(nil, nil)
}

// LeadershipTransferToServer does the same as LeadershipTransfer but takes a
// server in the arguments in case a leadership should be transitioned to a
// specific server in the cluster.  Note that raft protocol version 3 is not
// sufficient to use LeadershipTransfer. A recent version of that library has
// to be used that includes this feature. Using transfer leadership is safe
// however in a cluster where not every node has the latest version. If a
// follower cannot be promoted, it will fail gracefully.
func (r *Raft) LeadershipTransferToServer(id ServerID, address ServerAddress)
Future {
    if r.protocolVersion < 3 {
        return errorFuture{ErrUnsupportedProtocol}
    }

    return r.initiateLeadershipTransfer(&id, &address)
}
```

../raft/commands.go

```
package raft

// RPCHeader is a common sub-structure used to pass along protocol version and
// other information about the cluster. For older Raft implementations before
// versioning was added this will default to a zero-valued structure when read
// by newer Raft versions.
type RPCHeader struct {
    // ProtocolVersion is the version of the protocol the sender is
    // speaking.
    ProtocolVersion ProtocolVersion
}

// WithRPCHeader is an interface that exposes the RPC header.
type WithRPCHeader interface {
    GetRPCHeader() RPCHeader
}

// AppendEntriesRequest is the command used to append entries to the
// replicated log.
type AppendEntriesRequest struct {
    RPCHeader

    // Provide the current term and leader
    Term   uint64
    Leader []byte

    // Provide the previous entries for integrity checking
    PrevLogEntry uint64
    PrevLogTerm  uint64

    // New entries to commit
    Entries []*Log

    // Commit index on the leader
    LeaderCommitIndex uint64
}

// GetRPCHeader - See WithRPCHeader.
func (r *AppendEntriesRequest) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}

// AppendEntriesResponse is the response returned from an
// AppendEntriesRequest.
type AppendEntriesResponse struct {
    RPCHeader

    // Newer term if leader is out of date
    Term uint64

    // Last Log is a hint to help accelerate rebuilding slow nodes
    LastLog uint64
```

```
    // We may not succeed if we have a conflicting entry
    Success bool

    // There are scenarios where this request didn't succeed
    // but there's no need to wait/back-off the next attempt.
    NoRetryBackoff bool
}

// GetRPCHeader - See WithRPCHeader.
func (r *AppendEntriesResponse) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}

// RequestVoteRequest is the command used by a candidate to ask a Raft peer
// for a vote in an election.
type RequestVoteRequest struct {
    RPCHeader

    // Provide the term and our id
    Term      uint64
    Candidate []byte

    // Used to ensure safety
    LastLogIndex uint64
    LastLogTerm  uint64

    // Used to indicate to peers if this vote was triggered by a leadership
    // transfer. It is required for leadership transfer to work, because
servers
    // wouldn't vote otherwise if they are aware of an existing leader.
    LeadershipTransfer bool
}

// GetRPCHeader - See WithRPCHeader.
func (r *RequestVoteRequest) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}

// RequestVoteResponse is the response returned from a RequestVoteRequest.
type RequestVoteResponse struct {
    RPCHeader

    // Newer term if leader is out of date.
    Term uint64

    // Peers is deprecated, but required by servers that only understand
    // protocol version 0. This is not populated in protocol version 2
    // and later.
    Peers []byte

    // Is the vote granted.
    Granted bool
```

```
}

// GetRPCHeader - See WithRPCHeader.
func (r *RequestVoteResponse) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}

// InstallSnapshotRequest is the command sent to a Raft peer to bootstrap its
// log (and state machine) from a snapshot on another peer.
type InstallSnapshotRequest struct {
    RPCHeader
    SnapshotVersion SnapshotVersion

    Term   uint64
    Leader []byte

    // These are the last index/term included in the snapshot
    LastLogIndex uint64
    LastLogTerm  uint64

    // Peer Set in the snapshot. This is deprecated in favor of Configuration
    // but remains here in case we receive an InstallSnapshot from a leader
    // that's running old code.
    Peers []byte

    // Cluster membership.
    Configuration []byte
    // Log index where 'Configuration' entry was originally written.
    ConfigurationIndex uint64

    // Size of the snapshot
    Size int64
}

// GetRPCHeader - See WithRPCHeader.
func (r *InstallSnapshotRequest) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}

// InstallSnapshotResponse is the response returned from an
// InstallSnapshotRequest.
type InstallSnapshotResponse struct {
    RPCHeader

    Term    uint64
    Success bool
}

// GetRPCHeader - See WithRPCHeader.
func (r *InstallSnapshotResponse) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}
```

```go
// TimeoutNowRequest is the command used by a leader to signal another server
to
// start an election.
type TimeoutNowRequest struct {
    RPCHeader
}

// GetRPCHeader - See WithRPCHeader.
func (r *TimeoutNowRequest) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}

// TimeoutNowResponse is the response to TimeoutNowRequest.
type TimeoutNowResponse struct {
    RPCHeader
}

// GetRPCHeader - See WithRPCHeader.
func (r *TimeoutNowResponse) GetRPCHeader() RPCHeader {
    return r.RPCHeader
}
```

../raft/commitment.go

```
package raft

import (
    "sort"
    "sync"
)

// Commitment is used to advance the leader's commit index. The leader and
// replication goroutines report in newly written entries with Match(), and
// this notifies on commitCh when the commit index has advanced.
type commitment struct {
    // protects matchIndexes and commitIndex
    sync.Mutex
    // notified when commitIndex increases
    commitCh chan struct{}
    // voter ID to log index: the server stores up through this log entry
    matchIndexes map[ServerID]uint64
    // a quorum stores up through this log entry. monotonically increases.
    commitIndex uint64
    // the first index of this leader's term: this needs to be replicated to a
    // majority of the cluster before this leader may mark anything committed
    // (per Raft's commitment rule)
    startIndex uint64
}

// newCommitment returns an commitment struct that notifies the provided
// channel when log entries have been committed. A new commitment struct is
// created each time this server becomes leader for a particular term.
// 'configuration' is the servers in the cluster.
// 'startIndex' is the first index created in this term (see
// its description above).
func newCommitment(commitCh chan struct{}, configuration Configuration,
startIndex uint64) *commitment {
    matchIndexes := make(map[ServerID]uint64)
    for _, server := range configuration.Servers {
        if server.Suffrage == Voter {
            matchIndexes[server.ID] = 0
        }
    }
    return &commitment{
        commitCh:     commitCh,
        matchIndexes: matchIndexes,
        commitIndex:  0,
        startIndex:   startIndex,
    }
}

// Called when a new cluster membership configuration is created: it will be
// used to determine commitment from now on. 'configuration' is the servers in
// the cluster.
func (c *commitment) setConfiguration(configuration Configuration) {
    c.Lock()
```

```
    defer c.Unlock()
    oldMatchIndexes := c.matchIndexes
    c.matchIndexes = make(map[ServerID]uint64)
    for _, server := range configuration.Servers {
        if server.Suffrage == Voter {
            c.matchIndexes[server.ID] = oldMatchIndexes[server.ID] // defaults
to 0
        }
    }
    c.recalculate()
}

// Called by leader after commitCh is notified
func (c *commitment) getCommitIndex() uint64 {
    c.Lock()
    defer c.Unlock()
    return c.commitIndex
}

// Match is called once a server completes writing entries to disk: either the
// leader has written the new entry or a follower has replied to an
// AppendEntries RPC. The given server's disk agrees with this server's log up
// through the given index.
func (c *commitment) match(server ServerID, matchIndex uint64) {
    c.Lock()
    defer c.Unlock()
    if prev, hasVote := c.matchIndexes[server]; hasVote && matchIndex > prev {
        c.matchIndexes[server] = matchIndex
        c.recalculate()
    }
}

// Internal helper to calculate new commitIndex from matchIndexes.
// Must be called with lock held.
func (c *commitment) recalculate() {
    if len(c.matchIndexes) == 0 {
        return
    }

    matched := make([]uint64, 0, len(c.matchIndexes))
    for _, idx := range c.matchIndexes {
        matched = append(matched, idx)
    }
    sort.Sort(uint64Slice(matched))
    quorumMatchIndex := matched[(len(matched)-1)/2]

    if quorumMatchIndex > c.commitIndex && quorumMatchIndex >= c.startIndex {
        c.commitIndex = quorumMatchIndex
        asyncNotifyCh(c.commitCh)
    }
}
```

../raft/commitment_test.go

```
package raft

import (
    "testing"
)

func makeConfiguration(voters []string) Configuration {
    var configuration Configuration
    for _, voter := range voters {
        configuration.Servers = append(configuration.Servers, Server{
            Suffrage: Voter,
            Address:  ServerAddress(voter + "addr"),
            ID:       ServerID(voter),
        })
    }
    return configuration
}

// Returns a slice of server names of size n.
func voters(n int) Configuration {
    if n > 7 {
        panic("only up to 7 servers implemented")
    }
    return makeConfiguration([]string{"s1", "s2", "s3", "s4", "s5", "s6", "s7"}
[:n])
}

// Tests setVoters() keeps matchIndexes where possible.
func TestCommitment_setVoters(t *testing.T) {
    commitCh := make(chan struct{}, 1)
    c := newCommitment(commitCh, makeConfiguration([]string{"a", "b", "c"}), 0)
    c.match("a", 10)
    c.match("b", 20)
    c.match("c", 30)
    // commitIndex: 20
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }
    c.setConfiguration(makeConfiguration([]string{"c", "d", "e"}))
    // c: 30, d: 0, e: 0
    c.match("e", 40)
    if c.getCommitIndex() != 30 {
        t.Fatalf("expected 30 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }
}

// Tests match() being called with smaller index than before.
func TestCommitment_match_max(t *testing.T) {
```

```
    commitCh := make(chan struct{}, 1)
    c := newCommitment(commitCh, voters(5), 4)

    c.match("s1", 8)
    c.match("s2", 8)
    c.match("s2", 1)
    c.match("s3", 8)

    if c.getCommitIndex() != 8 {
        t.Fatalf("calling match with an earlier index should be ignored")
    }
}

// Tests match() being called with non-voters.
func TestCommitment_match_nonVoting(t *testing.T) {
    commitCh := make(chan struct{}, 1)
    c := newCommitment(commitCh, voters(5), 4)

    c.match("s1", 8)
    c.match("s2", 8)
    c.match("s3", 8)

    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }

    c.match("s90", 10)
    c.match("s91", 10)
    c.match("s92", 10)

    if c.getCommitIndex() != 8 {
        t.Fatalf("non-voting servers shouldn't be able to commit")
    }
    if drainNotifyCh(commitCh) {
        t.Fatalf("unexpected commit notify")
    }
}

// Tests recalculate() algorithm.
func TestCommitment_recalculate(t *testing.T) {
    commitCh := make(chan struct{}, 1)
    c := newCommitment(commitCh, voters(5), 0)

    c.match("s1", 30)
    c.match("s2", 20)

    if c.getCommitIndex() != 0 {
        t.Fatalf("shouldn't commit after two of five servers")
    }
    if drainNotifyCh(commitCh) {
        t.Fatalf("unexpected commit notify")
    }
```

```
    c.match("s3", 10)
    if c.getCommitIndex() != 10 {
        t.Fatalf("expected 10 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }
    c.match("s4", 15)
    if c.getCommitIndex() != 15 {
        t.Fatalf("expected 15 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }

    c.setConfiguration(voters(3))
    // s1: 30, s2: 20, s3: 10
    if c.getCommitIndex() != 20 {
        t.Fatalf("expected 20 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }

    c.setConfiguration(voters(4))
    // s1: 30, s2: 20, s3: 10, s4: 0
    c.match("s2", 25)
    if c.getCommitIndex() != 20 {
        t.Fatalf("expected 20 entries committed, found %d",
            c.getCommitIndex())
    }
    if drainNotifyCh(commitCh) {
        t.Fatalf("unexpected commit notify")
    }
    c.match("s4", 23)
    if c.getCommitIndex() != 23 {
        t.Fatalf("expected 23 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }
}

// Tests recalculate() respecting startIndex.
func TestCommitment_recalculate_startIndex(t *testing.T) {
    commitCh := make(chan struct{}, 1)
    c := newCommitment(commitCh, voters(5), 4)

    c.match("s1", 3)
```

```
    c.match("s2", 3)
    c.match("s3", 3)

    if c.getCommitIndex() != 0 {
        t.Fatalf("can't commit until startIndex is replicated to a quorum")
    }
    if drainNotifyCh(commitCh) {
        t.Fatalf("unexpected commit notify")
    }

    c.match("s1", 4)
    c.match("s2", 4)
    c.match("s3", 4)

    if c.getCommitIndex() != 4 {
        t.Fatalf("should be able to commit startIndex once replicated to a
quorum")
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }
}

// With no voting members in the cluster, the most sane behavior is probably
// to not mark anything committed.
func TestCommitment_noVoterSanity(t *testing.T) {
    commitCh := make(chan struct{}, 1)
    c := newCommitment(commitCh, makeConfiguration([]string{}), 4)
    c.match("s1", 10)
    c.setConfiguration(makeConfiguration([]string{}))
    c.match("s1", 10)
    if c.getCommitIndex() != 0 {
        t.Fatalf("no voting servers: shouldn't be able to commit")
    }
    if drainNotifyCh(commitCh) {
        t.Fatalf("unexpected commit notify")
    }

    // add a voter so we can commit something and then remove it
    c.setConfiguration(voters(1))
    c.match("s1", 10)
    if c.getCommitIndex() != 10 {
        t.Fatalf("expected 10 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }

    c.setConfiguration(makeConfiguration([]string{}))
    c.match("s1", 20)
    if c.getCommitIndex() != 10 {
        t.Fatalf("expected 10 entries committed, found %d",
```

```
            c.getCommitIndex())
    }
    if drainNotifyCh(commitCh) {
        t.Fatalf("unexpected commit notify")
    }

}

// Single voter commits immediately.
func TestCommitment_singleVoter(t *testing.T) {
    commitCh := make(chan struct{}, 1)
    c := newCommitment(commitCh, voters(1), 4)
    c.match("s1", 10)
    if c.getCommitIndex() != 10 {
        t.Fatalf("expected 10 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }
    c.setConfiguration(voters(1))
    if drainNotifyCh(commitCh) {
        t.Fatalf("unexpected commit notify")
    }
    c.match("s1", 12)
    if c.getCommitIndex() != 12 {
        t.Fatalf("expected 12 entries committed, found %d",
            c.getCommitIndex())
    }
    if !drainNotifyCh(commitCh) {
        t.Fatalf("expected commit notify")
    }
}
```

../raft/config.go

```
package raft

import (
    "fmt"
    "io"
    "time"

    "github.com/hashicorp/go-hclog"
)

// ProtocolVersion is the version of the protocol (which includes RPC messages
// as well as Raft-specific log entries) that this server can _understand_. Use
// the ProtocolVersion member of the Config object to control the version of
// the protocol to use when _speaking_ to other servers. Note that depending on
// the protocol version being spoken, some otherwise understood RPC messages
// may be refused. See dispositionRPC for details of this logic.
//
// There are notes about the upgrade path in the description of the versions
// below. If you are starting a fresh cluster then there's no reason not to
// jump right to the latest protocol version. If you need to interoperate with
// older, version 0 Raft servers you'll need to drive the cluster through the
// different versions in order.
//
// The version details are complicated, but here's a summary of what's required
// to get from a version 0 cluster to version 3:
//
// 1. In version N of your app that starts using the new Raft library with
//    versioning, set ProtocolVersion to 1.
// 2. Make version N+1 of your app require version N as a prerequisite (all
//    servers must be upgraded). For version N+1 of your app set
ProtocolVersion
//    to 2.
// 3. Similarly, make version N+2 of your app require version N+1 as a
//    prerequisite. For version N+2 of your app, set ProtocolVersion to 3.
//
// During this upgrade, older cluster members will still have Server IDs equal
// to their network addresses. To upgrade an older member and give it an ID, it
// needs to leave the cluster and re-enter:
//
// 1. Remove the server from the cluster with RemoveServer, using its network
//    address as its ServerID.
// 2. Update the server's config to use a UUID or something else that is
//    not tied to the machine as the ServerID (restarting the server).
// 3. Add the server back to the cluster with AddVoter, using its new ID.
//
// You can do this during the rolling upgrade from N+1 to N+2 of your app, or
// as a rolling change at any time after the upgrade.
//
// Version History
//
// 0: Original Raft library before versioning was added. Servers running this
//    version of the Raft library use AddPeerDeprecated/RemovePeerDeprecated
```

```
//    for all configuration changes, and have no support for LogConfiguration.
// 1: First versioned protocol, used to interoperate with old servers, and
begin
//    the migration path to newer versions of the protocol. Under this version
//    all configuration changes are propagated using the now-deprecated
//    RemovePeerDeprecated Raft log entry. This means that server IDs are
always
//    set to be the same as the server addresses (since the old log entry type
//    cannot transmit an ID), and only AddPeer/RemovePeer APIs are supported.
//    Servers running this version of the protocol can understand the new
//    LogConfiguration Raft log entry but will never generate one so they can
//    remain compatible with version 0 Raft servers in the cluster.
// 2: Transitional protocol used when migrating an existing cluster to the new
//    server ID system. Server IDs are still set to be the same as server
//    addresses, but all configuration changes are propagated using the new
//    LogConfiguration Raft log entry type, which can carry full ID
information.
//    This version supports the old AddPeer/RemovePeer APIs as well as the new
//    ID-based AddVoter/RemoveServer APIs which should be used when adding
//    version 3 servers to the cluster later. This version sheds all
//    interoperability with version 0 servers, but can interoperate with newer
//    Raft servers running with protocol version 1 since they can understand
the
//    new LogConfiguration Raft log entry, and this version can still
understand
//    their RemovePeerDeprecated Raft log entries. We need this protocol
version
//    as an intermediate step between 1 and 3 so that servers will propagate
the
//    ID information that will come from newly-added (or -rolled) servers using
//    protocol version 3, but since they are still using their address-based
IDs
//    from the previous step they will still be able to track commitments and
//    their own voting status properly. If we skipped this step, servers would
//    be started with their new IDs, but they wouldn't see themselves in the
old
//    address-based configuration, so none of the servers would think they had
a
//    vote.
// 3: Protocol adding full support for server IDs and new ID-based server APIs
//    (AddVoter, AddNonvoter, etc.), old AddPeer/RemovePeer APIs are no longer
//    supported. Version 2 servers should be swapped out by removing them from
//    the cluster one-by-one and re-adding them with updated configuration for
//    this protocol version, along with their server ID. The remove/add cycle
//    is required to populate their server ID. Note that removing must be done
//    by ID, which will be the old server's address.
type ProtocolVersion int

const (
    // ProtocolVersionMin is the minimum protocol version
    ProtocolVersionMin ProtocolVersion = 0
    // ProtocolVersionMax is the maximum protocol version
    ProtocolVersionMax = 3
```

```
)

// SnapshotVersion is the version of snapshots that this server can understand.
// Currently, it is always assumed that the server generates the latest
version,
// though this may be changed in the future to include a configurable version.
//
// Version History
//
// 0: Original Raft library before versioning was added. The peers portion of
//    these snapshots is encoded in the legacy format which requires
decodePeers
//    to parse. This version of snapshots should only be produced by the
//    unversioned Raft library.
// 1: New format which adds support for a full configuration structure and its
//    associated log index, with support for server IDs and non-voting server
//    modes. To ease upgrades, this also includes the legacy peers structure
but
//    that will never be used by servers that understand version 1 snapshots.
//    Since the original Raft library didn't enforce any versioning, we must
//    include the legacy peers structure for this version, but we can deprecate
//    it in the next snapshot version.
type SnapshotVersion int

const (
    // SnapshotVersionMin is the minimum snapshot version
    SnapshotVersionMin SnapshotVersion = 0
    // SnapshotVersionMax is the maximum snapshot version
    SnapshotVersionMax = 1
)

// Config provides any necessary configuration for the Raft server.
type Config struct {
    // ProtocolVersion allows a Raft server to inter-operate with older
    // Raft servers running an older version of the code. This is used to
    // version the wire protocol as well as Raft-specific log entries that
    // the server uses when _speaking_ to other servers. There is currently
    // no auto-negotiation of versions so all servers must be manually
    // configured with compatible versions. See ProtocolVersionMin and
    // ProtocolVersionMax for the versions of the protocol that this server
    // can _understand_.
    ProtocolVersion ProtocolVersion

    // HeartbeatTimeout specifies the time in follower state without
    // a leader before we attempt an election.
    HeartbeatTimeout time.Duration

    // ElectionTimeout specifies the time in candidate state without
    // a leader before we attempt an election.
    ElectionTimeout time.Duration

    // CommitTimeout controls the time without an Apply() operation
    // before we heartbeat to ensure a timely commit. Due to random
```

```
    // staggering, may be delayed as much as 2x this value.
    CommitTimeout time.Duration

    // MaxAppendEntries controls the maximum number of append entries
    // to send at once. We want to strike a balance between efficiency
    // and avoiding waste if the follower is going to reject because of
    // an inconsistent log.
    MaxAppendEntries int

    // If we are a member of a cluster, and RemovePeer is invoked for the
    // local node, then we forget all peers and transition into the follower
state.
    // If ShutdownOnRemove is is set, we additional shutdown Raft. Otherwise,
    // we can become a leader of a cluster containing only this node.
    ShutdownOnRemove bool

    // TrailingLogs controls how many logs we leave after a snapshot. This is
    // used so that we can quickly replay logs on a follower instead of being
    // forced to send an entire snapshot.
    TrailingLogs uint64

    // SnapshotInterval controls how often we check if we should perform a
snapshot.
    // We randomly stagger between this value and 2x this value to avoid the
entire
    // cluster from performing a snapshot at once.
    SnapshotInterval time.Duration

    // SnapshotThreshold controls how many outstanding logs there must be
before
    // we perform a snapshot. This is to prevent excessive snapshots when we
can
    // just replay a small set of logs.
    SnapshotThreshold uint64

    // LeaderLeaseTimeout is used to control how long the "lease" lasts
    // for being the leader without being able to contact a quorum
    // of nodes. If we reach this interval without contact, we will
    // step down as leader.
    LeaderLeaseTimeout time.Duration

    // StartAsLeader forces Raft to start in the leader state. This should
    // never be used except for testing purposes, as it can cause a split-
brain.
    StartAsLeader bool

    // The unique ID for this server across all time. When running with
    // ProtocolVersion < 3, you must set this to be the same as the network
    // address of your transport.
    LocalID ServerID

    // NotifyCh is used to provide a channel that will be notified of
leadership
```

```
    // changes. Raft will block writing to this channel, so it should either be
    // buffered or aggressively consumed.
    NotifyCh chan<- bool

    // LogOutput is used as a sink for logs, unless Logger is specified.
    // Defaults to os.Stderr.
    LogOutput io.Writer

    // LogLevel represents a log level. If a no matching string is specified,
    // hclog.NoLevel is assumed.
    LogLevel string

    // Logger is a user-provided hc-log logger. If nil, a logger writing to
    // LogOutput with LogLevel is used.
    Logger hclog.Logger

    // NoSnapshotRestoreOnStart controls if raft will restore a snapshot to the
    // FSM on start. This is useful if your FSM recovers from other mechanisms
    // than raft snapshotting. Snapshot metadata will still be used to
initalize
    // raft's configuration and index values. This is used in NewRaft and
    // RestoreCluster.
    NoSnapshotRestoreOnStart bool

    // skipStartup allows NewRaft() to bypass all background work goroutines
    skipStartup bool
}

// DefaultConfig returns a Config with usable defaults.
func DefaultConfig() *Config {
    return &Config{
        ProtocolVersion:    ProtocolVersionMax,
        HeartbeatTimeout:   1000 * time.Millisecond,
        ElectionTimeout:    1000 * time.Millisecond,
        CommitTimeout:      50 * time.Millisecond,
        MaxAppendEntries:   64,
        ShutdownOnRemove:   true,
        TrailingLogs:       10240,
        SnapshotInterval:   120 * time.Second,
        SnapshotThreshold:  8192,
        LeaderLeaseTimeout: 500 * time.Millisecond,
        LogLevel:           "DEBUG",
    }
}

// ValidateConfig is used to validate a sane configuration
func ValidateConfig(config *Config) error {
    // We don't actually support running as 0 in the library any more, but
    // we do understand it.
    protocolMin := ProtocolVersionMin
    if protocolMin == 0 {
        protocolMin = 1
    }
```

```
    if config.ProtocolVersion < protocolMin ||
        config.ProtocolVersion > ProtocolVersionMax {
        return fmt.Errorf("Protocol version %d must be >= %d and <= %d",
            config.ProtocolVersion, protocolMin, ProtocolVersionMax)
    }
    if len(config.LocalID) == 0 {
        return fmt.Errorf("LocalID cannot be empty")
    }
    if config.HeartbeatTimeout < 5*time.Millisecond {
        return fmt.Errorf("Heartbeat timeout is too low")
    }
    if config.ElectionTimeout < 5*time.Millisecond {
        return fmt.Errorf("Election timeout is too low")
    }
    if config.CommitTimeout < time.Millisecond {
        return fmt.Errorf("Commit timeout is too low")
    }
    if config.MaxAppendEntries <= 0 {
        return fmt.Errorf("MaxAppendEntries must be positive")
    }
    if config.MaxAppendEntries > 1024 {
        return fmt.Errorf("MaxAppendEntries is too large")
    }
    if config.SnapshotInterval < 5*time.Millisecond {
        return fmt.Errorf("Snapshot interval is too low")
    }
    if config.LeaderLeaseTimeout < 5*time.Millisecond {
        return fmt.Errorf("Leader lease timeout is too low")
    }
    if config.LeaderLeaseTimeout > config.HeartbeatTimeout {
        return fmt.Errorf("Leader lease timeout cannot be larger than heartbeat
timeout")
    }
    if config.ElectionTimeout < config.HeartbeatTimeout {
        return fmt.Errorf("Election timeout must be equal or greater than
Heartbeat Timeout")
    }
    return nil
}
```

../raft/configuration.go

```
package raft

import "fmt"

// ServerSuffrage determines whether a Server in a Configuration gets a vote.
type ServerSuffrage int

// Note: Don't renumber these, since the numbers are written into the log.
const (
    // Voter is a server whose vote is counted in elections and whose match
index
    // is used in advancing the leader's commit index.
    Voter ServerSuffrage = iota
    // Nonvoter is a server that receives log entries but is not considered for
    // elections or commitment purposes.
    Nonvoter
    // Staging is a server that acts like a nonvoter with one exception: once a
    // staging server receives enough log entries to be sufficiently caught up
to
    // the leader's log, the leader will invoke a  membership change to change
    // the Staging server to a Voter.
    Staging
)

func (s ServerSuffrage) String() string {
    switch s {
    case Voter:
        return "Voter"
    case Nonvoter:
        return "Nonvoter"
    case Staging:
        return "Staging"
    }
    return "ServerSuffrage"
}

// ConfigurationStore provides an interface that can optionally be implemented
by FSMs
// to store configuration updates made in the replicated log. In general this
is only
// necessary for FSMs that mutate durable state directly instead of applying
changes
// in memory and snapshotting periodically. By storing configuration changes,
the
// persistent FSM state can behave as a complete snapshot, and be able to
recover
// without an external snapshot just for persisting the raft configuration.
type ConfigurationStore interface {
    // ConfigurationStore is a superset of the FSM functionality
    FSM

    // StoreConfiguration is invoked once a log entry containing a
```

```
configuration
    // change is committed. It takes the index at which the configuration was
    // written and the configuration value.
    StoreConfiguration(index uint64, configuration Configuration)
}

type nopConfigurationStore struct{}

func (s nopConfigurationStore) StoreConfiguration(_ uint64, _ Configuration) {}

// ServerID is a unique string identifying a server for all time.
type ServerID string

// ServerAddress is a network address for a server that a transport can
contact.
type ServerAddress string

// Server tracks the information about a single server in a configuration.
type Server struct {
    // Suffrage determines whether the server gets a vote.
    Suffrage ServerSuffrage
    // ID is a unique string identifying this server for all time.
    ID ServerID
    // Address is its network address that a transport can contact.
    Address ServerAddress
}

// Configuration tracks which servers are in the cluster, and whether they have
// votes. This should include the local server, if it's a member of the
cluster.
// The servers are listed no particular order, but each should only appear
once.
// These entries are appended to the log during membership changes.
type Configuration struct {
    Servers []Server
}

// Clone makes a deep copy of a Configuration.
func (c *Configuration) Clone() (copy Configuration) {
    copy.Servers = append(copy.Servers, c.Servers...)
    return
}

// ConfigurationChangeCommand is the different ways to change the cluster
// configuration.
type ConfigurationChangeCommand uint8

const (
    // AddStaging makes a server Staging unless its Voter.
    AddStaging ConfigurationChangeCommand = iota
    // AddNonvoter makes a server Nonvoter unless its Staging or Voter.
    AddNonvoter
    // DemoteVoter makes a server Nonvoter unless its absent.
```

```
    DemoteVoter
    // RemoveServer removes a server entirely from the cluster membership.
    RemoveServer
    // Promote is created automatically by a leader; it turns a Staging server
    // into a Voter.
    Promote
)

func (c ConfigurationChangeCommand) String() string {
    switch c {
    case AddStaging:
        return "AddStaging"
    case AddNonvoter:
        return "AddNonvoter"
    case DemoteVoter:
        return "DemoteVoter"
    case RemoveServer:
        return "RemoveServer"
    case Promote:
        return "Promote"
    }
    return "ConfigurationChangeCommand"
}

// configurationChangeRequest describes a change that a leader would like to
// make to its current configuration. It's used only within a single server
// (never serialized into the log), as part of `configurationChangeFuture`.
type configurationChangeRequest struct {
    command       ConfigurationChangeCommand
    serverID      ServerID
    serverAddress ServerAddress // only present for AddStaging, AddNonvoter
    // prevIndex, if nonzero, is the index of the only configuration upon which
    // this change may be applied; if another configuration entry has been
    // added in the meantime, this request will fail.
    prevIndex uint64
}

// configurations is state tracked on every server about its Configurations.
// Note that, per Diego's dissertation, there can be at most one uncommitted
// configuration at a time (the next configuration may not be created until the
// prior one has been committed).
//
// One downside to storing just two configurations is that if you try to take a
// snapshot when your state machine hasn't yet applied the committedIndex, we
// have no record of the configuration that would logically fit into that
// snapshot. We disallow snapshots in that case now. An alternative approach,
// which LogCabin uses, is to track every configuration change in the
// log.
type configurations struct {
    // committed is the latest configuration in the log/snapshot that has been
    // committed (the one with the largest index).
    committed Configuration
    // committedIndex is the log index where 'committed' was written.
```

```
    committedIndex uint64
    // latest is the latest configuration in the log/snapshot (may be committed
    // or uncommitted)
    latest Configuration
    // latestIndex is the log index where 'latest' was written.
    latestIndex uint64
}

// Clone makes a deep copy of a configurations object.
func (c *configurations) Clone() (copy configurations) {
    copy.committed = c.committed.Clone()
    copy.committedIndex = c.committedIndex
    copy.latest = c.latest.Clone()
    copy.latestIndex = c.latestIndex
    return
}

// hasVote returns true if the server identified by 'id' is a Voter in the
// provided Configuration.
func hasVote(configuration Configuration, id ServerID) bool {
    for _, server := range configuration.Servers {
        if server.ID == id {
            return server.Suffrage == Voter
        }
    }
    return false
}

// checkConfiguration tests a cluster membership configuration for common
// errors.
func checkConfiguration(configuration Configuration) error {
    idSet := make(map[ServerID]bool)
    addressSet := make(map[ServerAddress]bool)
    var voters int
    for _, server := range configuration.Servers {
        if server.ID == "" {
            return fmt.Errorf("Empty ID in configuration: %v", configuration)
        }
        if server.Address == "" {
            return fmt.Errorf("Empty address in configuration: %v", server)
        }
        if idSet[server.ID] {
            return fmt.Errorf("Found duplicate ID in configuration: %v",
server.ID)
        }
        idSet[server.ID] = true
        if addressSet[server.Address] {
            return fmt.Errorf("Found duplicate address in configuration: %v",
server.Address)
        }
        addressSet[server.Address] = true
        if server.Suffrage == Voter {
            voters++
```

```
        }
    }
    if voters == 0 {
        return fmt.Errorf("Need at least one voter in configuration: %v",
configuration)
    }
    return nil
}

// nextConfiguration generates a new Configuration from the current one and a
// configuration change request. It's split from appendConfigurationEntry so
// that it can be unit tested easily.
func nextConfiguration(current Configuration, currentIndex uint64, change
configurationChangeRequest) (Configuration, error) {
    if change.prevIndex > 0 && change.prevIndex != currentIndex {
        return Configuration{}, fmt.Errorf("Configuration changed since %v
(latest is %v)", change.prevIndex, currentIndex)
    }

    configuration := current.Clone()
    switch change.command {
    case AddStaging:
        // TODO: barf on new address?
        newServer := Server{
            // TODO: This should add the server as Staging, to be automatically
            // promoted to Voter later. However, the promotion to Voter is not
yet
            // implemented, and doing so is not trivial with the way the leader
loop
            // coordinates with the replication goroutines today. So, for now,
the
            // server will have a vote right away, and the Promote case below
is
            // unused.
            Suffrage: Voter,
            ID:       change.serverID,
            Address:  change.serverAddress,
        }
        found := false
        for i, server := range configuration.Servers {
            if server.ID == change.serverID {
                if server.Suffrage == Voter {
                    configuration.Servers[i].Address = change.serverAddress
                } else {
                    configuration.Servers[i] = newServer
                }
                found = true
                break
            }
        }
        if !found {
            configuration.Servers = append(configuration.Servers, newServer)
        }
```

```
    case AddNonvoter:
        newServer := Server{
            Suffrage: Nonvoter,
            ID:       change.serverID,
            Address:  change.serverAddress,
        }
        found := false
        for i, server := range configuration.Servers {
            if server.ID == change.serverID {
                if server.Suffrage != Nonvoter {
                    configuration.Servers[i].Address = change.serverAddress
                } else {
                    configuration.Servers[i] = newServer
                }
                found = true
                break
            }
        }
        if !found {
            configuration.Servers = append(configuration.Servers, newServer)
        }
    case DemoteVoter:
        for i, server := range configuration.Servers {
            if server.ID == change.serverID {
                configuration.Servers[i].Suffrage = Nonvoter
                break
            }
        }
    case RemoveServer:
        for i, server := range configuration.Servers {
            if server.ID == change.serverID {
                configuration.Servers = append(configuration.Servers[:i],
configuration.Servers[i+1:]...)
                break
            }
        }
    case Promote:
        for i, server := range configuration.Servers {
            if server.ID == change.serverID && server.Suffrage == Staging {
                configuration.Servers[i].Suffrage = Voter
                break
            }
        }
    }

    // Make sure we didn't do something bad like remove the last voter
    if err := checkConfiguration(configuration); err != nil {
        return Configuration{}, err
    }

    return configuration, nil
}
```

```
// encodePeers is used to serialize a Configuration into the old peers format.
// This is here for backwards compatibility when operating with a mix of old
// servers and should be removed once we deprecate support for protocol version
1.
func encodePeers(configuration Configuration, trans Transport) []byte {
    // Gather up all the voters, other suffrage types are not supported by
    // this data format.
    var encPeers [][]byte
    for _, server := range configuration.Servers {
        if server.Suffrage == Voter {
            encPeers = append(encPeers, trans.EncodePeer(server.ID,
server.Address))
        }
    }

    // Encode the entire array.
    buf, err := encodeMsgPack(encPeers)
    if err != nil {
        panic(fmt.Errorf("failed to encode peers: %v", err))
    }

    return buf.Bytes()
}

// decodePeers is used to deserialize an old list of peers into a
Configuration.
// This is here for backwards compatibility with old log entries and snapshots;
// it should be removed eventually.
func decodePeers(buf []byte, trans Transport) Configuration {
    // Decode the buffer first.
    var encPeers [][]byte
    if err := decodeMsgPack(buf, &encPeers); err != nil {
        panic(fmt.Errorf("failed to decode peers: %v", err))
    }

    // Deserialize each peer.
    var servers []Server
    for _, enc := range encPeers {
        p := trans.DecodePeer(enc)
        servers = append(servers, Server{
            Suffrage: Voter,
            ID:       ServerID(p),
            Address:  ServerAddress(p),
        })
    }

    return Configuration{
        Servers: servers,
    }
}

// EncodeConfiguration serializes a Configuration using MsgPack, or panics on
// errors.
```

```go
func EncodeConfiguration(configuration Configuration) []byte {
    buf, err := encodeMsgPack(configuration)
    if err != nil {
        panic(fmt.Errorf("failed to encode configuration: %v", err))
    }
    return buf.Bytes()
}

// DecodeConfiguration deserializes a Configuration using MsgPack, or panics on
// errors.
func DecodeConfiguration(buf []byte) Configuration {
    var configuration Configuration
    if err := decodeMsgPack(buf, &configuration); err != nil {
        panic(fmt.Errorf("failed to decode configuration: %v", err))
    }
    return configuration
}
```

../raft/configuration_test.go

```
package raft

import (
    "fmt"
    "reflect"
    "strings"
    "testing"
)

var sampleConfiguration = Configuration{
    Servers: []Server{
        {
            Suffrage: Nonvoter,
            ID:       ServerID("id0"),
            Address:  ServerAddress("addr0"),
        },
        {
            Suffrage: Voter,
            ID:       ServerID("id1"),
            Address:  ServerAddress("addr1"),
        },
        {
            Suffrage: Staging,
            ID:       ServerID("id2"),
            Address:  ServerAddress("addr2"),
        },
    },
}

func TestConfiguration_Configuration_Clone(t *testing.T) {
    cloned := sampleConfiguration.Clone()
    if !reflect.DeepEqual(sampleConfiguration, cloned) {
        t.Fatalf("mismatch %v %v", sampleConfiguration, cloned)
    }
    cloned.Servers[1].ID = "scribble"
    if sampleConfiguration.Servers[1].ID == "scribble" {
        t.Fatalf("cloned configuration shouldn't alias Servers")
    }
}

func TestConfiguration_configurations_Clone(t *testing.T) {
    configuration := configurations{
        committed:      sampleConfiguration,
        committedIndex: 1,
        latest:         sampleConfiguration,
        latestIndex:    2,
    }
    cloned := configuration.Clone()
    if !reflect.DeepEqual(configuration, cloned) {
        t.Fatalf("mismatch %v %v", configuration, cloned)
    }
    cloned.committed.Servers[1].ID = "scribble"
```

```
    cloned.latest.Servers[1].ID = "scribble"
    if configuration.committed.Servers[1].ID == "scribble" ||
        configuration.latest.Servers[1].ID == "scribble" {
        t.Fatalf("cloned configuration shouldn't alias Servers")
    }
}

func TestConfiguration_hasVote(t *testing.T) {
    if hasVote(sampleConfiguration, "id0") {
        t.Fatalf("id0 should not have vote")
    }
    if !hasVote(sampleConfiguration, "id1") {
        t.Fatalf("id1 should have vote")
    }
    if hasVote(sampleConfiguration, "id2") {
        t.Fatalf("id2 should not have vote")
    }
    if hasVote(sampleConfiguration, "someotherid") {
        t.Fatalf("someotherid should not have vote")
    }
}

func TestConfiguration_checkConfiguration(t *testing.T) {
    var configuration Configuration
    if checkConfiguration(configuration) == nil {
        t.Fatalf("empty configuration should be error")
    }

    configuration.Servers = append(configuration.Servers, Server{
        Suffrage: Nonvoter,
        ID:       ServerID("id0"),
        Address:  ServerAddress("addr0"),
    })
    if checkConfiguration(configuration) == nil {
        t.Fatalf("lack of voter should be error")
    }

    configuration.Servers = append(configuration.Servers, Server{
        Suffrage: Voter,
        ID:       ServerID("id1"),
        Address:  ServerAddress("addr1"),
    })
    if err := checkConfiguration(configuration); err != nil {
        t.Fatalf("should be OK: %v", err)
    }

    configuration.Servers[1].ID = "id0"
    err := checkConfiguration(configuration)
    if err == nil {
        t.Fatalf("duplicate ID should be error")
    }
    if !strings.Contains(err.Error(), "duplicate ID") {
        t.Fatalf("unexpected error: %v", err)
```

```
    }
    configuration.Servers[1].ID = "id1"

    configuration.Servers[1].Address = "addr0"
    err = checkConfiguration(configuration)
    if err == nil {
        t.Fatalf("duplicate address should be error")
    }
    if !strings.Contains(err.Error(), "duplicate address") {
        t.Fatalf("unexpected error: %v", err)
    }
}

var singleServer = Configuration{
    Servers: []Server{
        {
            Suffrage: Voter,
            ID:       ServerID("id1"),
            Address:  ServerAddress("addr1x"),
        },
    },
}

var oneOfEach = Configuration{
    Servers: []Server{
        {
            Suffrage: Voter,
            ID:       ServerID("id1"),
            Address:  ServerAddress("addr1x"),
        },
        {
            Suffrage: Staging,
            ID:       ServerID("id2"),
            Address:  ServerAddress("addr2x"),
        },
        {
            Suffrage: Nonvoter,
            ID:       ServerID("id3"),
            Address:  ServerAddress("addr3x"),
        },
    },
}

var voterPair = Configuration{
    Servers: []Server{
        {
            Suffrage: Voter,
            ID:       ServerID("id1"),
            Address:  ServerAddress("addr1x"),
        },
        {
            Suffrage: Voter,
            ID:       ServerID("id2"),
```

```
            Address:  ServerAddress("addr2x"),
        },
    },
}

var nextConfigurationTests = []struct {
    current  Configuration
    command  ConfigurationChangeCommand
    serverID int
    next     string
}{
    // AddStaging: was missing.
    {Configuration{}, AddStaging, 1, "{[{Voter id1 addr1}]}"},
    {singleServer, AddStaging, 2, "{[{Voter id1 addr1x} {Voter id2 addr2}]}"},
    // AddStaging: was Voter.
    {singleServer, AddStaging, 1, "{[{Voter id1 addr1}]}"},
    // AddStaging: was Staging.
    {oneOfEach, AddStaging, 2, "{[{Voter id1 addr1x} {Voter id2 addr2}
{Nonvoter id3 addr3x}]}"},
    // AddStaging: was Nonvoter.
    {oneOfEach, AddStaging, 3, "{[{Voter id1 addr1x} {Staging id2 addr2x}
{Voter id3 addr3}]}"},

    // AddNonvoter: was missing.
    {singleServer, AddNonvoter, 2, "{[{Voter id1 addr1x} {Nonvoter id2
addr2}]}"},
    // AddNonvoter: was Voter.
    {singleServer, AddNonvoter, 1, "{[{Voter id1 addr1}]}"},
    // AddNonvoter: was Staging.
    {oneOfEach, AddNonvoter, 2, "{[{Voter id1 addr1x} {Staging id2 addr2}
{Nonvoter id3 addr3x}]}"},
    // AddNonvoter: was Nonvoter.
    {oneOfEach, AddNonvoter, 3, "{[{Voter id1 addr1x} {Staging id2 addr2x}
{Nonvoter id3 addr3}]}"},

    // DemoteVoter: was missing.
    {singleServer, DemoteVoter, 2, "{[{Voter id1 addr1x}]}"},
    // DemoteVoter: was Voter.
    {voterPair, DemoteVoter, 2, "{[{Voter id1 addr1x} {Nonvoter id2
addr2x}]}"},
    // DemoteVoter: was Staging.
    {oneOfEach, DemoteVoter, 2, "{[{Voter id1 addr1x} {Nonvoter id2 addr2x}
{Nonvoter id3 addr3x}]}"},
    // DemoteVoter: was Nonvoter.
    {oneOfEach, DemoteVoter, 3, "{[{Voter id1 addr1x} {Staging id2 addr2x}
{Nonvoter id3 addr3x}]}"},

    // RemoveServer: was missing.
    {singleServer, RemoveServer, 2, "{[{Voter id1 addr1x}]}"},
    // RemoveServer: was Voter.
    {voterPair, RemoveServer, 2, "{[{Voter id1 addr1x}]}"},
    // RemoveServer: was Staging.
    {oneOfEach, RemoveServer, 2, "{[{Voter id1 addr1x} {Nonvoter id3
```

```
addr3x}]}"},
    // RemoveServer: was Nonvoter.
    {oneOfEach, RemoveServer, 3, "{[{Voter id1 addr1x} {Staging id2
addr2x}]}"},

    // Promote: was missing.
    {singleServer, Promote, 2, "{[{Voter id1 addr1x}]}"},
    // Promote: was Voter.
    {singleServer, Promote, 1, "{[{Voter id1 addr1x}]}"},
    // Promote: was Staging.
    {oneOfEach, Promote, 2, "{[{Voter id1 addr1x} {Voter id2 addr2x} {Nonvoter
id3 addr3x}]}"},
    // Promote: was Nonvoter.
    {oneOfEach, Promote, 3, "{[{Voter id1 addr1x} {Staging id2 addr2x}
{Nonvoter id3 addr3x}]}"},
}

func TestConfiguration_nextConfiguration_table(t *testing.T) {
    for i, tt := range nextConfigurationTests {
        req := configurationChangeRequest{
            command:       tt.command,
            serverID:      ServerID(fmt.Sprintf("id%d", tt.serverID)),
            serverAddress: ServerAddress(fmt.Sprintf("addr%d", tt.serverID)),
        }
        next, err := nextConfiguration(tt.current, 1, req)
        if err != nil {
            t.Errorf("nextConfiguration %d should have succeeded, got %v", i,
err)
            continue
        }
        if fmt.Sprintf("%v", next) != tt.next {
            t.Errorf("nextConfiguration %d returned %v, expected %s", i, next,
tt.next)
            continue
        }
    }
}

func TestConfiguration_nextConfiguration_prevIndex(t *testing.T) {
    // Stale prevIndex.
    req := configurationChangeRequest{
        command:       AddStaging,
        serverID:      ServerID("id1"),
        serverAddress: ServerAddress("addr1"),
        prevIndex:     1,
    }
    _, err := nextConfiguration(singleServer, 2, req)
    if err == nil || !strings.Contains(err.Error(), "changed") {
        t.Fatalf("nextConfiguration should have failed due to intervening
configuration change")
    }

    // Current prevIndex.
```

```
    req = configurationChangeRequest{
        command:       AddStaging,
        serverID:      ServerID("id2"),
        serverAddress: ServerAddress("addr2"),
        prevIndex:     2,
    }
    _, err = nextConfiguration(singleServer, 2, req)
    if err != nil {
        t.Fatalf("nextConfiguration should have succeeded, got %v", err)
    }

    // Zero prevIndex.
    req = configurationChangeRequest{
        command:       AddStaging,
        serverID:      ServerID("id3"),
        serverAddress: ServerAddress("addr3"),
        prevIndex:     0,
    }
    _, err = nextConfiguration(singleServer, 2, req)
    if err != nil {
        t.Fatalf("nextConfiguration should have succeeded, got %v", err)
    }
}

func TestConfiguration_nextConfiguration_checkConfiguration(t *testing.T) {
    req := configurationChangeRequest{
        command:       AddNonvoter,
        serverID:      ServerID("id1"),
        serverAddress: ServerAddress("addr1"),
    }
    _, err := nextConfiguration(Configuration{}, 1, req)
    if err == nil || !strings.Contains(err.Error(), "at least one voter") {
        t.Fatalf("nextConfiguration should have failed for not having a voter")
    }
}

func TestConfiguration_encodeDecodePeers(t *testing.T) {
    // Set up configuration.
    var configuration Configuration
    for i := 0; i < 3; i++ {
        address := NewInmemAddr()
        configuration.Servers = append(configuration.Servers, Server{
            Suffrage: Voter,
            ID:       ServerID(address),
            Address:  ServerAddress(address),
        })
    }

    // Encode into the old format.
    _, trans := NewInmemTransport("")
    buf := encodePeers(configuration, trans)

    // Decode from old format, as if reading an old log entry.
```

```
    decoded := decodePeers(buf, trans)
    if !reflect.DeepEqual(configuration, decoded) {
        t.Fatalf("mismatch %v %v", configuration, decoded)
    }
}

func TestConfiguration_encodeDecodeConfiguration(t *testing.T) {
    decoded := DecodeConfiguration(EncodeConfiguration(sampleConfiguration))
    if !reflect.DeepEqual(sampleConfiguration, decoded) {
        t.Fatalf("mismatch %v %v", sampleConfiguration, decoded)
    }
}
```

../raft/discard_snapshot.go

```
package raft

import (
    "fmt"
    "io"
)

// DiscardSnapshotStore is used to successfully snapshot while
// always discarding the snapshot. This is useful for when the
// log should be truncated but no snapshot should be retained.
// This should never be used for production use, and is only
// suitable for testing.
type DiscardSnapshotStore struct{}

// DiscardSnapshotSink is used to fulfill the SnapshotSink interface
// while always discarding the . This is useful for when the log
// should be truncated but no snapshot should be retained. This
// should never be used for production use, and is only suitable
// for testing.
type DiscardSnapshotSink struct{}

// NewDiscardSnapshotStore is used to create a new DiscardSnapshotStore.
func NewDiscardSnapshotStore() *DiscardSnapshotStore {
    return &DiscardSnapshotStore{}
}

// Create returns a valid type implementing the SnapshotSink which
// always discards the snapshot.
func (d *DiscardSnapshotStore) Create(version SnapshotVersion, index, term
uint64,
    configuration Configuration, configurationIndex uint64, trans Transport)
(SnapshotSink, error) {
    return &DiscardSnapshotSink{}, nil
}

// List returns successfully with a nil for []*SnapshotMeta.
func (d *DiscardSnapshotStore) List() ([]*SnapshotMeta, error) {
    return nil, nil
}

// Open returns an error since the DiscardSnapshotStore does not
// support opening snapshots.
func (d *DiscardSnapshotStore) Open(id string) (*SnapshotMeta, io.ReadCloser,
error) {
    return nil, nil, fmt.Errorf("open is not supported")
}

// Write returns successfully with the lenght of the input byte slice
// to satisfy the WriteCloser interface
func (d *DiscardSnapshotSink) Write(b []byte) (int, error) {
    return len(b), nil
}
```

```
// Close returns a nil error
func (d *DiscardSnapshotSink) Close() error {
    return nil
}

// ID returns "discard" for DiscardSnapshotSink
func (d *DiscardSnapshotSink) ID() string {
    return "discard"
}

// Cancel returns successfully with a nil error
func (d *DiscardSnapshotSink) Cancel() error {
    return nil
}
```

../raft/discard_snapshot_test.go

```
package raft

import "testing"

func TestDiscardSnapshotStoreImpl(t *testing.T) {
    var impl interface{} = &DiscardSnapshotStore{}
    if _, ok := impl.(SnapshotStore); !ok {
        t.Fatalf("DiscardSnapshotStore not a SnapshotStore")
    }
}

func TestDiscardSnapshotSinkImpl(t *testing.T) {
    var impl interface{} = &DiscardSnapshotSink{}
    if _, ok := impl.(SnapshotSink); !ok {
        t.Fatalf("DiscardSnapshotSink not a SnapshotSink")
    }
}
```

../raft/file_snapshot.go

```
package raft

import (
    "bufio"
    "bytes"
    "encoding/json"
    "fmt"
    "github.com/hashicorp/go-hclog"
    "hash"
    "hash/crc64"
    "io"
    "io/ioutil"
    "os"
    "path/filepath"
    "runtime"
    "sort"
    "strings"
    "time"
)

const (
    testPath      = "permTest"
    snapPath      = "snapshots"
    metaFilePath  = "meta.json"
    stateFilePath = "state.bin"
    tmpSuffix     = ".tmp"
)

// FileSnapshotStore implements the SnapshotStore interface and allows
// snapshots to be made on the local disk.
type FileSnapshotStore struct {
    path   string
    retain int
    logger hclog.Logger
}

type snapMetaSlice []*fileSnapshotMeta

// FileSnapshotSink implements SnapshotSink with a file.
type FileSnapshotSink struct {
    store     *FileSnapshotStore
    logger    hclog.Logger
    dir       string
    parentDir string
    meta      fileSnapshotMeta

    stateFile *os.File
    stateHash hash.Hash64
    buffered  *bufio.Writer

    closed bool
}
```

```
// fileSnapshotMeta is stored on disk. We also put a CRC
// on disk so that we can verify the snapshot.
type fileSnapshotMeta struct {
    SnapshotMeta
    CRC []byte
}

// bufferedFile is returned when we open a snapshot. This way
// reads are buffered and the file still gets closed.
type bufferedFile struct {
    bh *bufio.Reader
    fh *os.File
}

func (b *bufferedFile) Read(p []byte) (n int, err error) {
    return b.bh.Read(p)
}

func (b *bufferedFile) Close() error {
    return b.fh.Close()
}

// NewFileSnapshotStoreWithLogger creates a new FileSnapshotStore based
// on a base directory. The `retain` parameter controls how many
// snapshots are retained. Must be at least 1.
func NewFileSnapshotStoreWithLogger(base string, retain int, logger
hclog.Logger) (*FileSnapshotStore, error) {
    if retain < 1 {
        return nil, fmt.Errorf("must retain at least one snapshot")
    }
    if logger == nil {
        logger = hclog.New(&hclog.LoggerOptions{
            Name:   "snapshot",
            Output: hclog.DefaultOutput,
            Level:  hclog.DefaultLevel,
        })
    }

    // Ensure our path exists
    path := filepath.Join(base, snapPath)
    if err := os.MkdirAll(path, 0755); err != nil && !os.IsExist(err) {
        return nil, fmt.Errorf("snapshot path not accessible: %v", err)
    }

    // Setup the store
    store := &FileSnapshotStore{
        path:   path,
        retain: retain,
        logger: logger,
    }

    // Do a permissions test
```

```
    if err := store.testPermissions(); err != nil {
        return nil, fmt.Errorf("permissions test failed: %v", err)
    }
    return store, nil
}

// NewFileSnapshotStore creates a new FileSnapshotStore based
// on a base directory. The `retain` parameter controls how many
// snapshots are retained. Must be at least 1.
func NewFileSnapshotStore(base string, retain int, logOutput io.Writer)
(*FileSnapshotStore, error) {
    if logOutput == nil {
        logOutput = os.Stderr
    }
    return NewFileSnapshotStoreWithLogger(base, retain,
hclog.New(&hclog.LoggerOptions{
        Name:   "snapshot",
        Output: logOutput,
        Level:  hclog.DefaultLevel,
    }))
}

// testPermissions tries to touch a file in our path to see if it works.
func (f *FileSnapshotStore) testPermissions() error {
    path := filepath.Join(f.path, testPath)
    fh, err := os.Create(path)
    if err != nil {
        return err
    }

    if err = fh.Close(); err != nil {
        return err
    }

    if err = os.Remove(path); err != nil {
        return err
    }
    return nil
}

// snapshotName generates a name for the snapshot.
func snapshotName(term, index uint64) string {
    now := time.Now()
    msec := now.UnixNano() / int64(time.Millisecond)
    return fmt.Sprintf("%d-%d-%d", term, index, msec)
}

// Create is used to start a new snapshot
func (f *FileSnapshotStore) Create(version SnapshotVersion, index, term uint64,
    configuration Configuration, configurationIndex uint64, trans Transport)
(SnapshotSink, error) {
    // We only support version 1 snapshots at this time.
    if version != 1 {
```

```
        return nil, fmt.Errorf("unsupported snapshot version %d", version)
    }

    // Create a new path
    name := snapshotName(term, index)
    path := filepath.Join(f.path, name+tmpSuffix)
    f.logger.Info("creating new snapshot", "path", path)

    // Make the directory
    if err := os.MkdirAll(path, 0755); err != nil {
        f.logger.Error("failed to make snapshot directly", "error", err)
        return nil, err
    }

    // Create the sink
    sink := &FileSnapshotSink{
        store:     f,
        logger:    f.logger,
        dir:       path,
        parentDir: f.path,
        meta: fileSnapshotMeta{
            SnapshotMeta: SnapshotMeta{
                Version:            version,
                ID:                 name,
                Index:              index,
                Term:               term,
                Peers:              encodePeers(configuration, trans),
                Configuration:      configuration,
                ConfigurationIndex: configurationIndex,
            },
            CRC: nil,
        },
    }

    // Write out the meta data
    if err := sink.writeMeta(); err != nil {
        f.logger.Error("failed to write metadata", "error", err)
        return nil, err
    }

    // Open the state file
    statePath := filepath.Join(path, stateFilePath)
    fh, err := os.Create(statePath)
    if err != nil {
        f.logger.Error("failed to create state file", "error", err)
        return nil, err
    }
    sink.stateFile = fh

    // Create a CRC64 hash
    sink.stateHash = crc64.New(crc64.MakeTable(crc64.ECMA))

    // Wrap both the hash and file in a MultiWriter with buffering
```

```
    multi := io.MultiWriter(sink.stateFile, sink.stateHash)
    sink.buffered = bufio.NewWriter(multi)

    // Done
    return sink, nil
}

// List returns available snapshots in the store.
func (f *FileSnapshotStore) List() ([]*SnapshotMeta, error) {
    // Get the eligible snapshots
    snapshots, err := f.getSnapshots()
    if err != nil {
        f.logger.Error("failed to get snapshots", "error", err)
        return nil, err
    }

    var snapMeta []*SnapshotMeta
    for _, meta := range snapshots {
        snapMeta = append(snapMeta, &meta.SnapshotMeta)
        if len(snapMeta) == f.retain {
            break
        }
    }
    return snapMeta, nil
}

// getSnapshots returns all the known snapshots.
func (f *FileSnapshotStore) getSnapshots() ([]*fileSnapshotMeta, error) {
    // Get the eligible snapshots
    snapshots, err := ioutil.ReadDir(f.path)
    if err != nil {
        f.logger.Error("failed to scan snapshot directory", "error", err)
        return nil, err
    }

    // Populate the metadata
    var snapMeta []*fileSnapshotMeta
    for _, snap := range snapshots {
        // Ignore any files
        if !snap.IsDir() {
            continue
        }

        // Ignore any temporary snapshots
        dirName := snap.Name()
        if strings.HasSuffix(dirName, tmpSuffix) {
            f.logger.Warn("found temporary snapshot", "name", dirName)
            continue
        }

        // Try to read the meta data
        meta, err := f.readMeta(dirName)
        if err != nil {
```

```
            f.logger.Warn("failed to read metadata", "name", dirName, "error",
err)
            continue
        }

        // Make sure we can understand this version.
        if meta.Version < SnapshotVersionMin || meta.Version >
SnapshotVersionMax {
            f.logger.Warn("snapshot version not supported", "name", dirName,
"version", meta.Version)
            continue
        }

        // Append, but only return up to the retain count
        snapMeta = append(snapMeta, meta)
    }

    // Sort the snapshot, reverse so we get new -> old
    sort.Sort(sort.Reverse(snapMetaSlice(snapMeta)))

    return snapMeta, nil
}

// readMeta is used to read the meta data for a given named backup
func (f *FileSnapshotStore) readMeta(name string) (*fileSnapshotMeta, error) {
    // Open the meta file
    metaPath := filepath.Join(f.path, name, metaFilePath)
    fh, err := os.Open(metaPath)
    if err != nil {
        return nil, err
    }
    defer fh.Close()

    // Buffer the file IO
    buffered := bufio.NewReader(fh)

    // Read in the JSON
    meta := &fileSnapshotMeta{}
    dec := json.NewDecoder(buffered)
    if err := dec.Decode(meta); err != nil {
        return nil, err
    }
    return meta, nil
}

// Open takes a snapshot ID and returns a ReadCloser for that snapshot.
func (f *FileSnapshotStore) Open(id string) (*SnapshotMeta, io.ReadCloser,
error) {
    // Get the metadata
    meta, err := f.readMeta(id)
    if err != nil {
        f.logger.Error("failed to get meta data to open snapshot", "error",
err)
```

```
        return nil, nil, err
    }

    // Open the state file
    statePath := filepath.Join(f.path, id, stateFilePath)
    fh, err := os.Open(statePath)
    if err != nil {
        f.logger.Error("failed to open state file", "error", err)
        return nil, nil, err
    }

    // Create a CRC64 hash
    stateHash := crc64.New(crc64.MakeTable(crc64.ECMA))

    // Compute the hash
    _, err = io.Copy(stateHash, fh)
    if err != nil {
        f.logger.Error("failed to read state file", "error", err)
        fh.Close()
        return nil, nil, err
    }

    // Verify the hash
    computed := stateHash.Sum(nil)
    if bytes.Compare(meta.CRC, computed) != 0 {
        f.logger.Error("CRC checksum failed", "stored", meta.CRC, "computed",
computed)
        fh.Close()
        return nil, nil, fmt.Errorf("CRC mismatch")
    }

    // Seek to the start
    if _, err := fh.Seek(0, 0); err != nil {
        f.logger.Error("state file seek failed", "error", err)
        fh.Close()
        return nil, nil, err
    }

    // Return a buffered file
    buffered := &bufferedFile{
        bh: bufio.NewReader(fh),
        fh: fh,
    }

    return &meta.SnapshotMeta, buffered, nil
}

// ReapSnapshots reaps any snapshots beyond the retain count.
func (f *FileSnapshotStore) ReapSnapshots() error {
    snapshots, err := f.getSnapshots()
    if err != nil {
        f.logger.Error("failed to get snapshots", "error", err)
        return err
```

```
    }

    for i := f.retain; i < len(snapshots); i++ {
        path := filepath.Join(f.path, snapshots[i].ID)
        f.logger.Info("reaping snapshot", "path", path)
        if err := os.RemoveAll(path); err != nil {
            f.logger.Error("failed to reap snapshot", "path", path, "error",
err)
            return err
        }
    }
    return nil
}

// ID returns the ID of the snapshot, can be used with Open()
// after the snapshot is finalized.
func (s *FileSnapshotSink) ID() string {
    return s.meta.ID
}

// Write is used to append to the state file. We write to the
// buffered IO object to reduce the amount of context switches.
func (s *FileSnapshotSink) Write(b []byte) (int, error) {
    return s.buffered.Write(b)
}

// Close is used to indicate a successful end.
func (s *FileSnapshotSink) Close() error {
    // Make sure close is idempotent
    if s.closed {
        return nil
    }
    s.closed = true

    // Close the open handles
    if err := s.finalize(); err != nil {
        s.logger.Error("failed to finalize snapshot", "error", err)
        if delErr := os.RemoveAll(s.dir); delErr != nil {
            s.logger.Error("failed to delete temporary snapshot directory",
"path", s.dir, "error", delErr)
            return delErr
        }
        return err
    }

    // Write out the meta data
    if err := s.writeMeta(); err != nil {
        s.logger.Error("failed to write metadata", "error", err)
        return err
    }

    // Move the directory into place
    newPath := strings.TrimSuffix(s.dir, tmpSuffix)
```

```
    if err := os.Rename(s.dir, newPath); err != nil {
        s.logger.Error("failed to move snapshot into place", "error", err)
        return err
    }

    if runtime.GOOS != "windows" { // skipping fsync for directory entry edits
on Windows, only needed for *nix style file systems
        parentFH, err := os.Open(s.parentDir)
        defer parentFH.Close()
        if err != nil {
            s.logger.Error("failed to open snapshot parent directory", "path",
s.parentDir, "error", err)
            return err
        }

        if err = parentFH.Sync(); err != nil {
            s.logger.Error("failed syncing parent directory", "path",
s.parentDir, "error", err)
            return err
        }
    }

    // Reap any old snapshots
    if err := s.store.ReapSnapshots(); err != nil {
        return err
    }

    return nil
}

// Cancel is used to indicate an unsuccessful end.
func (s *FileSnapshotSink) Cancel() error {
    // Make sure close is idempotent
    if s.closed {
        return nil
    }
    s.closed = true

    // Close the open handles
    if err := s.finalize(); err != nil {
        s.logger.Error("failed to finalize snapshot", "error", err)
        return err
    }

    // Attempt to remove all artifacts
    return os.RemoveAll(s.dir)
}

// finalize is used to close all of our resources.
func (s *FileSnapshotSink) finalize() error {
    // Flush any remaining data
    if err := s.buffered.Flush(); err != nil {
        return err
```

```
    }

    // Sync to force fsync to disk
    if err := s.stateFile.Sync(); err != nil {
        return err
    }

    // Get the file size
    stat, statErr := s.stateFile.Stat()

    // Close the file
    if err := s.stateFile.Close(); err != nil {
        return err
    }

    // Set the file size, check after we close
    if statErr != nil {
        return statErr
    }
    s.meta.Size = stat.Size()

    // Set the CRC
    s.meta.CRC = s.stateHash.Sum(nil)
    return nil
}

// writeMeta is used to write out the metadata we have.
func (s *FileSnapshotSink) writeMeta() error {
    var err error
    // Open the meta file
    metaPath := filepath.Join(s.dir, metaFilePath)
    var fh *os.File
    fh, err = os.Create(metaPath)
    if err != nil {
        return err
    }
    defer fh.Close()

    // Buffer the file IO
    buffered := bufio.NewWriter(fh)

    // Write out as JSON
    enc := json.NewEncoder(buffered)
    if err = enc.Encode(&s.meta); err != nil {
        return err
    }

    if err = buffered.Flush(); err != nil {
        return err
    }

    if err = fh.Sync(); err != nil {
        return err
```

```
    }

    return nil
}

// Implement the sort interface for []*fileSnapshotMeta.
func (s snapMetaSlice) Len() int {
    return len(s)
}

func (s snapMetaSlice) Less(i, j int) bool {
    if s[i].Term != s[j].Term {
        return s[i].Term < s[j].Term
    }
    if s[i].Index != s[j].Index {
        return s[i].Index < s[j].Index
    }
    return s[i].ID < s[j].ID
}

func (s snapMetaSlice) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
```

../raft/file_snapshot_test.go

```
package raft

import (
    "bytes"
    "io"
    "io/ioutil"
    "os"
    "reflect"
    "runtime"
    "testing"
)

func TestFileSnapshotStoreImpl(t *testing.T) {
    var impl interface{} = &FileSnapshotStore{}
    if _, ok := impl.(SnapshotStore); !ok {
        t.Fatalf("FileSnapshotStore not a SnapshotStore")
    }
}

func TestFileSnapshotSinkImpl(t *testing.T) {
    var impl interface{} = &FileSnapshotSink{}
    if _, ok := impl.(SnapshotSink); !ok {
        t.Fatalf("FileSnapshotSink not a SnapshotSink")
    }
}

func TestFileSS_CreateSnapshotMissingParentDir(t *testing.T) {
    parent, err := ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }
    defer os.RemoveAll(parent)

    dir, err := ioutil.TempDir(parent, "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }

    snap, err := NewFileSnapshotStoreWithLogger(dir, 3, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    os.RemoveAll(parent)
    _, trans := NewInmemTransport(NewInmemAddr())
    _, err = snap.Create(SnapshotVersionMax, 10, 3, Configuration{}, 0, trans)
    if err != nil {
        t.Fatalf("should not fail when using non existing parent")
    }

}
func TestFileSS_CreateSnapshot(t *testing.T) {
```

```
    // Create a test dir
    dir, err := ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }
    defer os.RemoveAll(dir)

    snap, err := NewFileSnapshotStoreWithLogger(dir, 3, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Check no snapshots
    snaps, err := snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 0 {
        t.Fatalf("did not expect any snapshots: %v", snaps)
    }

    // Create a new sink
    var configuration Configuration
    configuration.Servers = append(configuration.Servers, Server{
        Suffrage: Voter,
        ID:       ServerID("my id"),
        Address:  ServerAddress("over here"),
    })
    _, trans := NewInmemTransport(NewInmemAddr())
    sink, err := snap.Create(SnapshotVersionMax, 10, 3, configuration, 2,
trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // The sink is not done, should not be in a list!
    snaps, err = snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 0 {
        t.Fatalf("did not expect any snapshots: %v", snaps)
    }

    // Write to the sink
    _, err = sink.Write([]byte("first\n"))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    _, err = sink.Write([]byte("second\n"))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
```

```
    // Done!
    err = sink.Close()
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Should have a snapshot!
    snaps, err = snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 1 {
        t.Fatalf("expect a snapshots: %v", snaps)
    }

    // Check the latest
    latest := snaps[0]
    if latest.Index != 10 {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if latest.Term != 3 {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if !reflect.DeepEqual(latest.Configuration, configuration) {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if latest.ConfigurationIndex != 2 {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if latest.Size != 13 {
        t.Fatalf("bad snapshot: %v", *latest)
    }

    // Read the snapshot
    _, r, err := snap.Open(latest.ID)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Read out everything
    var buf bytes.Buffer
    if _, err := io.Copy(&buf, r); err != nil {
        t.Fatalf("err: %v", err)
    }
    if err := r.Close(); err != nil {
        t.Fatalf("err: %v", err)
    }

    // Ensure a match
    if bytes.Compare(buf.Bytes(), []byte("first\nsecond\n")) != 0 {
        t.Fatalf("content mismatch")
    }
```

```
}

func TestFileSS_CancelSnapshot(t *testing.T) {
    // Create a test dir
    dir, err := ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }
    defer os.RemoveAll(dir)

    snap, err := NewFileSnapshotStoreWithLogger(dir, 3, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Create a new sink
    _, trans := NewInmemTransport(NewInmemAddr())
    sink, err := snap.Create(SnapshotVersionMax, 10, 3, Configuration{}, 0,
trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Cancel the snapshot! Should delete
    err = sink.Cancel()
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // The sink is canceled, should not be in a list!
    snaps, err := snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 0 {
        t.Fatalf("did not expect any snapshots: %v", snaps)
    }
}

func TestFileSS_Retention(t *testing.T) {
    var err error
    // Create a test dir
    var dir string
    dir, err = ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }
    defer os.RemoveAll(dir)

    var snap *FileSnapshotStore
    snap, err = NewFileSnapshotStoreWithLogger(dir, 2, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
```

```
    }

    // Create a few snapshots
    _, trans := NewInmemTransport(NewInmemAddr())
    for i := 10; i < 15; i++ {
        var sink SnapshotSink
        sink, err = snap.Create(SnapshotVersionMax, uint64(i), 3,
Configuration{}, 0, trans)
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        err = sink.Close()
        if err != nil {
            t.Fatalf("err: %v", err)
        }
    }

    // Should only have 2 listed!
    var snaps []*SnapshotMeta
    snaps, err = snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 2 {
        t.Fatalf("expect 2 snapshots: %v", snaps)
    }

    // Check they are the latest
    if snaps[0].Index != 14 {
        t.Fatalf("bad snap: %#v", *snaps[0])
    }
    if snaps[1].Index != 13 {
        t.Fatalf("bad snap: %#v", *snaps[1])
    }
}

func TestFileSS_BadPerm(t *testing.T) {
    var err error
    if runtime.GOOS == "windows" {
        t.Skip("skipping file permission test on windows")
    }

    // Create a temp dir
    var dir1 string
    dir1, err = ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %s", err)
    }
    defer os.RemoveAll(dir1)

    // Create a sub dir and remove all permissions
    var dir2 string
    dir2, err = ioutil.TempDir(dir1, "badperm")
```

```
    if err != nil {
        t.Fatalf("err: %s", err)
    }
    if err = os.Chmod(dir2, 000); err != nil {
        t.Fatalf("err: %s", err)
    }
    defer os.Chmod(dir2, 777) // Set perms back for delete

    // Should fail
    if _, err = NewFileSnapshotStore(dir2, 3, nil); err == nil {
        t.Fatalf("should fail to use dir with bad perms")
    }
}

func TestFileSS_MissingParentDir(t *testing.T) {
    parent, err := ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }
    defer os.RemoveAll(parent)

    dir, err := ioutil.TempDir(parent, "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }

    os.RemoveAll(parent)
    _, err = NewFileSnapshotStore(dir, 3, nil)
    if err != nil {
        t.Fatalf("should not fail when using non existing parent")
    }
}

func TestFileSS_Ordering(t *testing.T) {
    // Create a test dir
    dir, err := ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }
    defer os.RemoveAll(dir)

    snap, err := NewFileSnapshotStoreWithLogger(dir, 3, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Create a new sink
    _, trans := NewInmemTransport(NewInmemAddr())
    sink, err := snap.Create(SnapshotVersionMax, 130350, 5, Configuration{}, 0,
trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
```

```
    err = sink.Close()
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    sink, err = snap.Create(SnapshotVersionMax, 204917, 36, Configuration{}, 0,
trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    err = sink.Close()
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Should only have 2 listed!
    snaps, err := snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 2 {
        t.Fatalf("expect 2 snapshots: %v", snaps)
    }

    // Check they are ordered
    if snaps[0].Term != 36 {
        t.Fatalf("bad snap: %#v", *snaps[0])
    }
    if snaps[1].Term != 5 {
        t.Fatalf("bad snap: %#v", *snaps[1])
    }
}
```

../raft/fsm.go

```
package raft

import (
    "fmt"
    "io"
    "time"

    "github.com/armon/go-metrics"
)

// FSM provides an interface that can be implemented by
// clients to make use of the replicated log.
type FSM interface {
    // Apply log is invoked once a log entry is committed.
    // It returns a value which will be made available in the
    // ApplyFuture returned by Raft.Apply method if that
    // method was called on the same Raft node as the FSM.
    Apply(*Log) interface{}

    // Snapshot is used to support log compaction. This call should
    // return an FSMSnapshot which can be used to save a point-in-time
    // snapshot of the FSM. Apply and Snapshot are not called in multiple
    // threads, but Apply will be called concurrently with Persist. This means
    // the FSM should be implemented in a fashion that allows for concurrent
    // updates while a snapshot is happening.
    Snapshot() (FSMSnapshot, error)

    // Restore is used to restore an FSM from a snapshot. It is not called
    // concurrently with any other command. The FSM must discard all previous
    // state.
    Restore(io.ReadCloser) error
}

// BatchingFSM extends the FSM interface to add an ApplyBatch function. This
can
// optionally be implemented by clients to enable multiple logs to be applied
to
// the FSM in batches. Up to MaxAppendEntries could be sent in a batch.
type BatchingFSM interface {
    // ApplyBatch is invoked once a batch of log entries has been committed and
    // are ready to be applied to the FSM. ApplyBatch will take in an array of
    // log entries. These log entries will be in the order they were committed,
    // will not have gaps, and could be of a few log types. Clients should
check
    // the log type prior to attempting to decode the data attached. Presently
    // the LogCommand and LogConfiguration types will be sent.
    //
    // The returned slice must be the same length as the input and each
response
    // should correlate to the log at the same index of the input. The returned
    // values will be made available in the ApplyFuture returned by Raft.Apply
    // method if that method was called on the same Raft node as the FSM.
```

```
    ApplyBatch([]*Log) []interface{}

    FSM
}

// FSMSnapshot is returned by an FSM in response to a Snapshot
// It must be safe to invoke FSMSnapshot methods with concurrent
// calls to Apply.
type FSMSnapshot interface {
    // Persist should dump all necessary state to the WriteCloser 'sink',
    // and call sink.Close() when finished or call sink.Cancel() on error.
    Persist(sink SnapshotSink) error

    // Release is invoked when we are finished with the snapshot.
    Release()
}

// runFSM is a long running goroutine responsible for applying logs
// to the FSM. This is done async of other logs since we don't want
// the FSM to block our internal operations.
func (r *Raft) runFSM() {
    var lastIndex, lastTerm uint64

    batchingFSM, batchingEnabled := r.fsm.(BatchingFSM)
    configStore, configStoreEnabled := r.fsm.(ConfigurationStore)

    commitSingle := func(req *commitTuple) {
        // Apply the log if a command or config change
        var resp interface{}
        // Make sure we send a response
        defer func() {
            // Invoke the future if given
            if req.future != nil {
                req.future.response = resp
                req.future.respond(nil)
            }
        }()

        switch req.log.Type {
        case LogCommand:
            start := time.Now()
            resp = r.fsm.Apply(req.log)
            metrics.MeasureSince([]string{"raft", "fsm", "apply"}, start)

        case LogConfiguration:
            if !configStoreEnabled {
                // Return early to avoid incrementing the index and term for
                // an unimplemented operation.
                return
            }

            start := time.Now()
            configStore.StoreConfiguration(req.log.Index,
```

```
DecodeConfiguration(req.log.Data))
            metrics.MeasureSince([]string{"raft", "fsm", "store_config"},
start)
        }

        // Update the indexes
        lastIndex = req.log.Index
        lastTerm = req.log.Term
    }

    commitBatch := func(reqs []*commitTuple) {
        if !batchingEnabled {
            for _, ct := range reqs {
                commitSingle(ct)
            }
            return
        }

        // Only send LogCommand and LogConfiguration log types. LogBarrier
types
        // will not be sent to the FSM.
        shouldSend := func(l *Log) bool {
            switch l.Type {
            case LogCommand, LogConfiguration:
                return true
            }
            return false
        }

        var lastBatchIndex, lastBatchTerm uint64
        sendLogs := make([]*Log, 0, len(reqs))
        for _, req := range reqs {
            if shouldSend(req.log) {
                sendLogs = append(sendLogs, req.log)
            }
            lastBatchIndex = req.log.Index
            lastBatchTerm = req.log.Term
        }

        var responses []interface{}
        if len(sendLogs) > 0 {
            start := time.Now()
            responses = batchingFSM.ApplyBatch(sendLogs)
            metrics.MeasureSince([]string{"raft", "fsm", "applyBatch"}, start)
            metrics.AddSample([]string{"raft", "fsm", "applyBatchNum"},
float32(len(reqs)))

            // Ensure we get the expected responses
            if len(sendLogs) != len(responses) {
                panic("invalid number of responses")
            }
        }
```

```
        // Update the indexes
        lastIndex = lastBatchIndex
        lastTerm = lastBatchTerm

        var i int
        for _, req := range reqs {
            var resp interface{}
            // If the log was sent to the FSM, retrieve the response.
            if shouldSend(req.log) {
                resp = responses[i]
                i++
            }

            if req.future != nil {
                req.future.response = resp
                req.future.respond(nil)
            }
        }
    }

    restore := func(req *restoreFuture) {
        // Open the snapshot
        meta, source, err := r.snapshots.Open(req.ID)
        if err != nil {
            req.respond(fmt.Errorf("failed to open snapshot %v: %v", req.ID,
err))
            return
        }

        // Attempt to restore
        start := time.Now()
        if err := r.fsm.Restore(source); err != nil {
            req.respond(fmt.Errorf("failed to restore snapshot %v: %v", req.ID,
err))
            source.Close()
            return
        }
        source.Close()
        metrics.MeasureSince([]string{"raft", "fsm", "restore"}, start)

        // Update the last index and term
        lastIndex = meta.Index
        lastTerm = meta.Term
        req.respond(nil)
    }

    snapshot := func(req *reqSnapshotFuture) {
        // Is there something to snapshot?
        if lastIndex == 0 {
            req.respond(ErrNothingNewToSnapshot)
            return
        }
```

```
        // Start a snapshot
        start := time.Now()
        snap, err := r.fsm.Snapshot()
        metrics.MeasureSince([]string{"raft", "fsm", "snapshot"}, start)

        // Respond to the request
        req.index = lastIndex
        req.term = lastTerm
        req.snapshot = snap
        req.respond(err)
    }

    for {
        select {
        case ptr := <-r.fsmMutateCh:
            switch req := ptr.(type) {
            case []*commitTuple:
                commitBatch(req)

            case *restoreFuture:
                restore(req)

            default:
                panic(fmt.Errorf("bad type passed to fsmMutateCh: %#v", ptr))
            }

        case req := <-r.fsmSnapshotCh:
            snapshot(req)

        case <-r.shutdownCh:
            return
        }
    }
}
```

../raft/future.go

```go
package raft

import (
    "fmt"
    "io"
    "sync"
    "time"
)

// Future is used to represent an action that may occur in the future.
type Future interface {
    // Error blocks until the future arrives and then
    // returns the error status of the future.
    // This may be called any number of times - all
    // calls will return the same value.
    // Note that it is not OK to call this method
    // twice concurrently on the same Future instance.
    Error() error
}

// IndexFuture is used for future actions that can result in a raft log entry
// being created.
type IndexFuture interface {
    Future

    // Index holds the index of the newly applied log entry.
    // This must not be called until after the Error method has returned.
    Index() uint64
}

// ApplyFuture is used for Apply and can return the FSM response.
type ApplyFuture interface {
    IndexFuture

    // Response returns the FSM response as returned
    // by the FSM.Apply method. This must not be called
    // until after the Error method has returned.
    Response() interface{}
}

// ConfigurationFuture is used for GetConfiguration and can return the
// latest configuration in use by Raft.
type ConfigurationFuture interface {
    IndexFuture

    // Configuration contains the latest configuration. This must
    // not be called until after the Error method has returned.
    Configuration() Configuration
}

// SnapshotFuture is used for waiting on a user-triggered snapshot to complete.
type SnapshotFuture interface {
```

```
    Future

    // Open is a function you can call to access the underlying snapshot and
    // its metadata. This must not be called until after the Error method
    // has returned.
    Open() (*SnapshotMeta, io.ReadCloser, error)
}

// LeadershipTransferFuture is used for waiting on a user-triggered leadership
// transfer to complete.
type LeadershipTransferFuture interface {
    Future
}

// errorFuture is used to return a static error.
type errorFuture struct {
    err error
}

func (e errorFuture) Error() error {
    return e.err
}

func (e errorFuture) Response() interface{} {
    return nil
}

func (e errorFuture) Index() uint64 {
    return 0
}

// deferError can be embedded to allow a future
// to provide an error in the future.
type deferError struct {
    err       error
    errCh     chan error
    responded bool
}

func (d *deferError) init() {
    d.errCh = make(chan error, 1)
}

func (d *deferError) Error() error {
    if d.err != nil {
        // Note that when we've received a nil error, this
        // won't trigger, but the channel is closed after
        // send so we'll still return nil below.
        return d.err
    }
    if d.errCh == nil {
        panic("waiting for response on nil channel")
    }
```

```
    d.err = <-d.errCh
    return d.err
}

func (d *deferError) respond(err error) {
    if d.errCh == nil {
        return
    }
    if d.responded {
        return
    }
    d.errCh <- err
    close(d.errCh)
    d.responded = true
}

// There are several types of requests that cause a configuration entry to
// be appended to the log. These are encoded here for leaderLoop() to process.
// This is internal to a single server.
type configurationChangeFuture struct {
    logFuture
    req configurationChangeRequest
}

// bootstrapFuture is used to attempt a live bootstrap of the cluster. See the
// Raft object's BootstrapCluster member function for more details.
type bootstrapFuture struct {
    deferError

    // configuration is the proposed bootstrap configuration to apply.
    configuration Configuration
}

// logFuture is used to apply a log entry and waits until
// the log is considered committed.
type logFuture struct {
    deferError
    log      Log
    response interface{}
    dispatch time.Time
}

func (l *logFuture) Response() interface{} {
    return l.response
}

func (l *logFuture) Index() uint64 {
    return l.log.Index
}

type shutdownFuture struct {
    raft *Raft
}
```

```
func (s *shutdownFuture) Error() error {
    if s.raft == nil {
        return nil
    }
    s.raft.waitShutdown()
    if closeable, ok := s.raft.trans.(WithClose); ok {
        closeable.Close()
    }
    return nil
}

// userSnapshotFuture is used for waiting on a user-triggered snapshot to
// complete.
type userSnapshotFuture struct {
    deferError

    // opener is a function used to open the snapshot. This is filled in
    // once the future returns with no error.
    opener func() (*SnapshotMeta, io.ReadCloser, error)
}

// Open is a function you can call to access the underlying snapshot and its
// metadata.
func (u *userSnapshotFuture) Open() (*SnapshotMeta, io.ReadCloser, error) {
    if u.opener == nil {
        return nil, nil, fmt.Errorf("no snapshot available")
    }
    // Invalidate the opener so it can't get called multiple times,
    // which isn't generally safe.
    defer func() {
        u.opener = nil
    }()
    return u.opener()
}

// userRestoreFuture is used for waiting on a user-triggered restore of an
// external snapshot to complete.
type userRestoreFuture struct {
    deferError

    // meta is the metadata that belongs with the snapshot.
    meta *SnapshotMeta

    // reader is the interface to read the snapshot contents from.
    reader io.Reader
}

// reqSnapshotFuture is used for requesting a snapshot start.
// It is only used internally.
type reqSnapshotFuture struct {
    deferError
```

```
    // snapshot details provided by the FSM runner before responding
    index    uint64
    term     uint64
    snapshot FSMSnapshot
}

// restoreFuture is used for requesting an FSM to perform a
// snapshot restore. Used internally only.
type restoreFuture struct {
    deferError
    ID string
}

// verifyFuture is used to verify the current node is still
// the leader. This is to prevent a stale read.
type verifyFuture struct {
    deferError
    notifyCh   chan *verifyFuture
    quorumSize int
    votes      int
    voteLock   sync.Mutex
}

// leadershipTransferFuture is used to track the progress of a leadership
// transfer internally.
type leadershipTransferFuture struct {
    deferError

    ID      *ServerID
    Address *ServerAddress
}

// configurationsFuture is used to retrieve the current configurations. This is
// used to allow safe access to this information outside of the main thread.
type configurationsFuture struct {
    deferError
    configurations configurations
}

// Configuration returns the latest configuration in use by Raft.
func (c *configurationsFuture) Configuration() Configuration {
    return c.configurations.latest
}

// Index returns the index of the latest configuration in use by Raft.
func (c *configurationsFuture) Index() uint64 {
    return c.configurations.latestIndex
}

// vote is used to respond to a verifyFuture.
// This may block when responding on the notifyCh.
func (v *verifyFuture) vote(leader bool) {
    v.voteLock.Lock()
```

```
    defer v.voteLock.Unlock()

    // Guard against having notified already
    if v.notifyCh == nil {
        return
    }

    if leader {
        v.votes++
        if v.votes >= v.quorumSize {
            v.notifyCh <- v
            v.notifyCh = nil
        }
    } else {
        v.notifyCh <- v
        v.notifyCh = nil
    }
}

// appendFuture is used for waiting on a pipelined append
// entries RPC.
type appendFuture struct {
    deferError
    start time.Time
    args  *AppendEntriesRequest
    resp  *AppendEntriesResponse
}

func (a *appendFuture) Start() time.Time {
    return a.start
}

func (a *appendFuture) Request() *AppendEntriesRequest {
    return a.args
}

func (a *appendFuture) Response() *AppendEntriesResponse {
    return a.resp
}
```

../raft/future_test.go

```
package raft

import (
    "errors"
    "testing"
)

func TestDeferFutureSuccess(t *testing.T) {
    var f deferError
    f.init()
    f.respond(nil)
    if err := f.Error(); err != nil {
        t.Fatalf("unexpected error result; got %#v want nil", err)
    }
    if err := f.Error(); err != nil {
        t.Fatalf("unexpected error result; got %#v want nil", err)
    }
}

func TestDeferFutureError(t *testing.T) {
    want := errors.New("x")
    var f deferError
    f.init()
    f.respond(want)
    if got := f.Error(); got != want {
        t.Fatalf("unexpected error result; got %#v want %#v", got, want)
    }
    if got := f.Error(); got != want {
        t.Fatalf("unexpected error result; got %#v want %#v", got, want)
    }
}

func TestDeferFutureConcurrent(t *testing.T) {
    // Food for the race detector.
    want := errors.New("x")
    var f deferError
    f.init()
    go f.respond(want)
    if got := f.Error(); got != want {
        t.Errorf("unexpected error result; got %#v want %#v", got, want)
    }
}
```

../raft/inmem_snapshot.go

```
package raft

import (
    "bytes"
    "fmt"
    "io"
    "io/ioutil"
    "sync"
)

// InmemSnapshotStore implements the SnapshotStore interface and
// retains only the most recent snapshot
type InmemSnapshotStore struct {
    latest      *InmemSnapshotSink
    hasSnapshot bool
    sync.RWMutex
}

// InmemSnapshotSink implements SnapshotSink in memory
type InmemSnapshotSink struct {
    meta     SnapshotMeta
    contents *bytes.Buffer
}

// NewInmemSnapshotStore creates a blank new InmemSnapshotStore
func NewInmemSnapshotStore() *InmemSnapshotStore {
    return &InmemSnapshotStore{
        latest: &InmemSnapshotSink{
            contents: &bytes.Buffer{},
        },
    }
}

// Create replaces the stored snapshot with a new one using the given args
func (m *InmemSnapshotStore) Create(version SnapshotVersion, index, term
uint64,
    configuration Configuration, configurationIndex uint64, trans Transport)
(SnapshotSink, error) {
    // We only support version 1 snapshots at this time.
    if version != 1 {
        return nil, fmt.Errorf("unsupported snapshot version %d", version)
    }

    name := snapshotName(term, index)

    m.Lock()
    defer m.Unlock()

    sink := &InmemSnapshotSink{
        meta: SnapshotMeta{
            Version:            version,
            ID:                 name,
```

```
            Index:              index,
            Term:               term,
            Peers:              encodePeers(configuration, trans),
            Configuration:      configuration,
            ConfigurationIndex: configurationIndex,
        },
        contents: &bytes.Buffer{},
    }
    m.hasSnapshot = true
    m.latest = sink

    return sink, nil
}

// List returns the latest snapshot taken
func (m *InmemSnapshotStore) List() ([]*SnapshotMeta, error) {
    m.RLock()
    defer m.RUnlock()

    if !m.hasSnapshot {
        return []*SnapshotMeta{}, nil
    }
    return []*SnapshotMeta{&m.latest.meta}, nil
}

// Open wraps an io.ReadCloser around the snapshot contents
func (m *InmemSnapshotStore) Open(id string) (*SnapshotMeta, io.ReadCloser,
error) {
    m.RLock()
    defer m.RUnlock()

    if m.latest.meta.ID != id {
        return nil, nil, fmt.Errorf("[ERR] snapshot: failed to open snapshot
id: %s", id)
    }

    // Make a copy of the contents, since a bytes.Buffer can only be read
    // once.
    contents := bytes.NewBuffer(m.latest.contents.Bytes())
    return &m.latest.meta, ioutil.NopCloser(contents), nil
}

// Write appends the given bytes to the snapshot contents
func (s *InmemSnapshotSink) Write(p []byte) (n int, err error) {
    written, err := io.Copy(s.contents, bytes.NewReader(p))
    s.meta.Size += written
    return int(written), err
}

// Close updates the Size and is otherwise a no-op
func (s *InmemSnapshotSink) Close() error {
    return nil
}
```

```go
// ID returns the ID of the SnapshotMeta
func (s *InmemSnapshotSink) ID() string {
    return s.meta.ID
}

// Cancel returns successfully with a nil error
func (s *InmemSnapshotSink) Cancel() error {
    return nil
}
```

../raft/inmem_snapshot_test.go

```
package raft

import (
    "bytes"
    "io"
    "reflect"
    "testing"
)

func TestInmemSnapshotStoreImpl(t *testing.T) {
    var impl interface{} = &InmemSnapshotStore{}
    if _, ok := impl.(SnapshotStore); !ok {
        t.Fatalf("InmemSnapshotStore not a SnapshotStore")
    }
}

func TestInmemSnapshotSinkImpl(t *testing.T) {
    var impl interface{} = &InmemSnapshotSink{}
    if _, ok := impl.(SnapshotSink); !ok {
        t.Fatalf("InmemSnapshotSink not a SnapshotSink")
    }
}

func TestInmemSS_CreateSnapshot(t *testing.T) {
    snap := NewInmemSnapshotStore()

    // Check no snapshots
    snaps, err := snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 0 {
        t.Fatalf("did not expect any snapshots: %v", snaps)
    }

    // Create a new sink
    var configuration Configuration
    configuration.Servers = append(configuration.Servers, Server{
        Suffrage: Voter,
        ID:       ServerID("my id"),
        Address:  ServerAddress("over here"),
    })
    _, trans := NewInmemTransport(NewInmemAddr())
    sink, err := snap.Create(SnapshotVersionMax, 10, 3, configuration, 2,
trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // The sink is not done, should not be in a list!
    snaps, err = snap.List()
    if err != nil {
```

```
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 1 {
        t.Fatalf("should always be 1 snapshot: %v", snaps)
    }

    // Write to the sink
    _, err = sink.Write([]byte("first\n"))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    _, err = sink.Write([]byte("second\n"))
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Done!
    err = sink.Close()
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Should have a snapshot!
    snaps, err = snap.List()
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if len(snaps) != 1 {
        t.Fatalf("expect a snapshots: %v", snaps)
    }

    // Check the latest
    latest := snaps[0]
    if latest.Index != 10 {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if latest.Term != 3 {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if !reflect.DeepEqual(latest.Configuration, configuration) {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if latest.ConfigurationIndex != 2 {
        t.Fatalf("bad snapshot: %v", *latest)
    }
    if latest.Size != 13 {
        t.Fatalf("bad snapshot: %v", *latest)
    }

    // Read the snapshot
    _, r, err := snap.Open(latest.ID)
    if err != nil {
        t.Fatalf("err: %v", err)
```

```
    }

    // Read out everything
    var buf bytes.Buffer
    if _, err := io.Copy(&buf, r); err != nil {
        t.Fatalf("err: %v", err)
    }
    if err := r.Close(); err != nil {
        t.Fatalf("err: %v", err)
    }

    // Ensure a match
    if bytes.Compare(buf.Bytes(), []byte("first\nsecond\n")) != 0 {
        t.Fatalf("content mismatch")
    }
}

func TestInmemSS_OpenSnapshotTwice(t *testing.T) {
    snap := NewInmemSnapshotStore()

    // Create a new sink
    var configuration Configuration
    configuration.Servers = append(configuration.Servers, Server{
        Suffrage: Voter,
        ID:       ServerID("my id"),
        Address:  ServerAddress("over here"),
    })
    _, trans := NewInmemTransport(NewInmemAddr())
    sink, err := snap.Create(SnapshotVersionMax, 10, 3, configuration, 2,
trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Write to the sink
    _, err = sink.Write([]byte("data\n"))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    err = sink.Close()
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Read the snapshot a first time
    _, r, err := snap.Open(sink.ID())
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Read out everything
    var buf1 bytes.Buffer
    if _, err = io.Copy(&buf1, r); err != nil {
```

```
        t.Fatalf("err: %v", err)
    }
    if err = r.Close(); err != nil {
        t.Fatalf("err: %v", err)
    }

    // Ensure a match
    if bytes.Compare(buf1.Bytes(), []byte("data\n")) != 0 {
        t.Fatalf("content mismatch")
    }

    // Read the snapshot a second time.
    _, r, err = snap.Open(sink.ID())
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Read out everything again
    var buf2 bytes.Buffer
    if _, err := io.Copy(&buf2, r); err != nil {
        t.Fatalf("err: %v", err)
    }
    if err := r.Close(); err != nil {
        t.Fatalf("err: %v", err)
    }

    // Ensure it's still the same content
    if bytes.Compare(buf2.Bytes(), []byte("data\n")) != 0 {
        t.Fatalf("content mismatch")
    }
}
```

../raft/inmem_store.go

```
package raft

import (
    "errors"
    "sync"
)

var (
    // ErrKeyNotFound is returned when a key does not exist in collection.
    ErrKeyNotFound = errors.New("not found")
)

// InmemStore implements the LogStore and StableStore interface.
// It should NOT EVER be used for production. It is used only for
// unit tests. Use the MDBStore implementation instead.
type InmemStore struct {
    l         sync.RWMutex
    lowIndex  uint64
    highIndex uint64
    logs      map[uint64]*Log
    kv        map[string][]byte
    kvInt     map[string]uint64
}

// NewInmemStore returns a new in-memory backend. Do not ever
// use for production. Only for testing.
func NewInmemStore() *InmemStore {
    i := &InmemStore{
        logs:  make(map[uint64]*Log),
        kv:    make(map[string][]byte),
        kvInt: make(map[string]uint64),
    }
    return i
}

// FirstIndex implements the LogStore interface.
func (i *InmemStore) FirstIndex() (uint64, error) {
    i.l.RLock()
    defer i.l.RUnlock()
    return i.lowIndex, nil
}

// LastIndex implements the LogStore interface.
func (i *InmemStore) LastIndex() (uint64, error) {
    i.l.RLock()
    defer i.l.RUnlock()
    return i.highIndex, nil
}

// GetLog implements the LogStore interface.
func (i *InmemStore) GetLog(index uint64, log *Log) error {
    i.l.RLock()
```

```
    defer i.l.RUnlock()
    l, ok := i.logs[index]
    if !ok {
        return ErrLogNotFound
    }
    *log = *l
    return nil
}

// StoreLog implements the LogStore interface.
func (i *InmemStore) StoreLog(log *Log) error {
    return i.StoreLogs([]*Log{log})
}

// StoreLogs implements the LogStore interface.
func (i *InmemStore) StoreLogs(logs []*Log) error {
    i.l.Lock()
    defer i.l.Unlock()
    for _, l := range logs {
        i.logs[l.Index] = l
        if i.lowIndex == 0 {
            i.lowIndex = l.Index
        }
        if l.Index > i.highIndex {
            i.highIndex = l.Index
        }
    }
    return nil
}

// DeleteRange implements the LogStore interface.
func (i *InmemStore) DeleteRange(min, max uint64) error {
    i.l.Lock()
    defer i.l.Unlock()
    for j := min; j <= max; j++ {
        delete(i.logs, j)
    }
    if min <= i.lowIndex {
        i.lowIndex = max + 1
    }
    if max >= i.highIndex {
        i.highIndex = min - 1
    }
    if i.lowIndex > i.highIndex {
        i.lowIndex = 0
        i.highIndex = 0
    }
    return nil
}

// Set implements the StableStore interface.
func (i *InmemStore) Set(key []byte, val []byte) error {
    i.l.Lock()
```

```
    defer i.l.Unlock()
    i.kv[string(key)] = val
    return nil
}

// Get implements the StableStore interface.
func (i *InmemStore) Get(key []byte) ([]byte, error) {
    i.l.RLock()
    defer i.l.RUnlock()
    val := i.kv[string(key)]
    if val == nil {
        return nil, ErrKeyNotFound
    }
    return val, nil
}

// SetUint64 implements the StableStore interface.
func (i *InmemStore) SetUint64(key []byte, val uint64) error {
    i.l.Lock()
    defer i.l.Unlock()
    i.kvInt[string(key)] = val
    return nil
}

// GetUint64 implements the StableStore interface.
func (i *InmemStore) GetUint64(key []byte) (uint64, error) {
    i.l.RLock()
    defer i.l.RUnlock()
    return i.kvInt[string(key)], nil
}
```

../raft/inmem_transport.go

```
package raft

import (
    "fmt"
    "io"
    "sync"
    "time"
)

// NewInmemAddr returns a new in-memory addr with
// a randomly generate UUID as the ID.
func NewInmemAddr() ServerAddress {
    return ServerAddress(generateUUID())
}

// inmemPipeline is used to pipeline requests for the in-mem transport.
type inmemPipeline struct {
    trans    *InmemTransport
    peer     *InmemTransport
    peerAddr ServerAddress

    doneCh       chan AppendFuture
    inprogressCh chan *inmemPipelineInflight

    shutdown     bool
    shutdownCh   chan struct{}
    shutdownLock sync.RWMutex
}

type inmemPipelineInflight struct {
    future *appendFuture
    respCh <-chan RPCResponse
}

// InmemTransport Implements the Transport interface, to allow Raft to be
// tested in-memory without going over a network.
type InmemTransport struct {
    sync.RWMutex
    consumerCh chan RPC
    localAddr  ServerAddress
    peers      map[ServerAddress]*InmemTransport
    pipelines  []*inmemPipeline
    timeout    time.Duration
}

// NewInmemTransportWithTimeout is used to initialize a new transport and
// generates a random local address if none is specified. The given timeout
// will be used to decide how long to wait for a connected peer to process the
// RPCs that we're sending it. See also Connect() and Consumer().
func NewInmemTransportWithTimeout(addr ServerAddress, timeout time.Duration)
(ServerAddress, *InmemTransport) {
    if string(addr) == "" {
```

```
        addr = NewInmemAddr()
    }
    trans := &InmemTransport{
        consumerCh: make(chan RPC, 16),
        localAddr:  addr,
        peers:      make(map[ServerAddress]*InmemTransport),
        timeout:    timeout,
    }
    return addr, trans
}

// NewInmemTransport is used to initialize a new transport
// and generates a random local address if none is specified
func NewInmemTransport(addr ServerAddress) (ServerAddress, *InmemTransport) {
    return NewInmemTransportWithTimeout(addr, 50*time.Millisecond)
}

// SetHeartbeatHandler is used to set optional fast-path for
// heartbeats, not supported for this transport.
func (i *InmemTransport) SetHeartbeatHandler(cb func(RPC)) {
}

// Consumer implements the Transport interface.
func (i *InmemTransport) Consumer() <-chan RPC {
    return i.consumerCh
}

// LocalAddr implements the Transport interface.
func (i *InmemTransport) LocalAddr() ServerAddress {
    return i.localAddr
}

// AppendEntriesPipeline returns an interface that can be used to pipeline
// AppendEntries requests.
func (i *InmemTransport) AppendEntriesPipeline(id ServerID, target
ServerAddress) (AppendPipeline, error) {
    i.Lock()
    defer i.Unlock()

    peer, ok := i.peers[target]
    if !ok {
        return nil, fmt.Errorf("failed to connect to peer: %v", target)
    }
    pipeline := newInmemPipeline(i, peer, target)
    i.pipelines = append(i.pipelines, pipeline)
    return pipeline, nil
}

// AppendEntries implements the Transport interface.
func (i *InmemTransport) AppendEntries(id ServerID, target ServerAddress, args
*AppendEntriesRequest, resp *AppendEntriesResponse) error {
    rpcResp, err := i.makeRPC(target, args, nil, i.timeout)
    if err != nil {
```

```
        return err
    }

    // Copy the result back
    out := rpcResp.Response.(*AppendEntriesResponse)
    *resp = *out
    return nil
}

// RequestVote implements the Transport interface.
func (i *InmemTransport) RequestVote(id ServerID, target ServerAddress, args
*RequestVoteRequest, resp *RequestVoteResponse) error {
    rpcResp, err := i.makeRPC(target, args, nil, i.timeout)
    if err != nil {
        return err
    }

    // Copy the result back
    out := rpcResp.Response.(*RequestVoteResponse)
    *resp = *out
    return nil
}

// InstallSnapshot implements the Transport interface.
func (i *InmemTransport) InstallSnapshot(id ServerID, target ServerAddress,
args *InstallSnapshotRequest, resp *InstallSnapshotResponse, data io.Reader)
error {
    rpcResp, err := i.makeRPC(target, args, data, 10*i.timeout)
    if err != nil {
        return err
    }

    // Copy the result back
    out := rpcResp.Response.(*InstallSnapshotResponse)
    *resp = *out
    return nil
}

// TimeoutNow implements the Transport interface.
func (i *InmemTransport) TimeoutNow(id ServerID, target ServerAddress, args
*TimeoutNowRequest, resp *TimeoutNowResponse) error {
    rpcResp, err := i.makeRPC(target, args, nil, 10*i.timeout)
    if err != nil {
        return err
    }

    // Copy the result back
    out := rpcResp.Response.(*TimeoutNowResponse)
    *resp = *out
    return nil
}

func (i *InmemTransport) makeRPC(target ServerAddress, args interface{}, r
```

```
io.Reader, timeout time.Duration) (rpcResp RPCResponse, err error) {
    i.RLock()
    peer, ok := i.peers[target]
    i.RUnlock()

    if !ok {
        err = fmt.Errorf("failed to connect to peer: %v", target)
        return
    }

    // Send the RPC over
    respCh := make(chan RPCResponse)
    req := RPC{
        Command:  args,
        Reader:   r,
        RespChan: respCh,
    }
    select {
    case peer.consumerCh <- req:
    case <-time.After(timeout):
        err = fmt.Errorf("send timed out")
        return
    }

    // Wait for a response
    select {
    case rpcResp = <-respCh:
        if rpcResp.Error != nil {
            err = rpcResp.Error
        }
    case <-time.After(timeout):
        err = fmt.Errorf("command timed out")
    }
    return
}

// EncodePeer implements the Transport interface.
func (i *InmemTransport) EncodePeer(id ServerID, p ServerAddress) []byte {
    return []byte(p)
}

// DecodePeer implements the Transport interface.
func (i *InmemTransport) DecodePeer(buf []byte) ServerAddress {
    return ServerAddress(buf)
}

// Connect is used to connect this transport to another transport for
// a given peer name. This allows for local routing.
func (i *InmemTransport) Connect(peer ServerAddress, t Transport) {
    trans := t.(*InmemTransport)
    i.Lock()
    defer i.Unlock()
    i.peers[peer] = trans
```

```
}

// Disconnect is used to remove the ability to route to a given peer.
func (i *InmemTransport) Disconnect(peer ServerAddress) {
    i.Lock()
    defer i.Unlock()
    delete(i.peers, peer)

    // Disconnect any pipelines
    n := len(i.pipelines)
    for idx := 0; idx < n; idx++ {
        if i.pipelines[idx].peerAddr == peer {
            i.pipelines[idx].Close()
            i.pipelines[idx], i.pipelines[n-1] = i.pipelines[n-1], nil
            idx--
            n--
        }
    }
    i.pipelines = i.pipelines[:n]
}

// DisconnectAll is used to remove all routes to peers.
func (i *InmemTransport) DisconnectAll() {
    i.Lock()
    defer i.Unlock()
    i.peers = make(map[ServerAddress]*InmemTransport)

    // Handle pipelines
    for _, pipeline := range i.pipelines {
        pipeline.Close()
    }
    i.pipelines = nil
}

// Close is used to permanently disable the transport
func (i *InmemTransport) Close() error {
    i.DisconnectAll()
    return nil
}

func newInmemPipeline(trans *InmemTransport, peer *InmemTransport, addr
ServerAddress) *inmemPipeline {
    i := &inmemPipeline{
        trans:        trans,
        peer:         peer,
        peerAddr:     addr,
        doneCh:       make(chan AppendFuture, 16),
        inprogressCh: make(chan *inmemPipelineInflight, 16),
        shutdownCh:   make(chan struct{}),
    }
    go i.decodeResponses()
    return i
}
```

```
func (i *inmemPipeline) decodeResponses() {
    timeout := i.trans.timeout
    for {
        select {
        case inp := <-i.inprogressCh:
            var timeoutCh <-chan time.Time
            if timeout > 0 {
                timeoutCh = time.After(timeout)
            }

            select {
            case rpcResp := <-inp.respCh:
                // Copy the result back
                *inp.future.resp = *rpcResp.Response.(*AppendEntriesResponse)
                inp.future.respond(rpcResp.Error)

                select {
                case i.doneCh <- inp.future:
                case <-i.shutdownCh:
                    return
                }

            case <-timeoutCh:
                inp.future.respond(fmt.Errorf("command timed out"))
                select {
                case i.doneCh <- inp.future:
                case <-i.shutdownCh:
                    return
                }

            case <-i.shutdownCh:
                return
            }
        case <-i.shutdownCh:
            return
        }
    }
}

func (i *inmemPipeline) AppendEntries(args *AppendEntriesRequest, resp
*AppendEntriesResponse) (AppendFuture, error) {
    // Create a new future
    future := &appendFuture{
        start: time.Now(),
        args:  args,
        resp:  resp,
    }
    future.init()

    // Handle a timeout
    var timeout <-chan time.Time
    if i.trans.timeout > 0 {
```

```
        timeout = time.After(i.trans.timeout)
    }

    // Send the RPC over
    respCh := make(chan RPCResponse, 1)
    rpc := RPC{
        Command:  args,
        RespChan: respCh,
    }

    // Check if we have been already shutdown, otherwise the random choose
    // made by select statement below might pick consumerCh even if
    // shutdownCh was closed.
    i.shutdownLock.RLock()
    shutdown := i.shutdown
    i.shutdownLock.RUnlock()
    if shutdown {
        return nil, ErrPipelineShutdown
    }

    select {
    case i.peer.consumerCh <- rpc:
    case <-timeout:
        return nil, fmt.Errorf("command enqueue timeout")
    case <-i.shutdownCh:
        return nil, ErrPipelineShutdown
    }

    // Send to be decoded
    select {
    case i.inprogressCh <- &inmemPipelineInflight{future, respCh}:
        return future, nil
    case <-i.shutdownCh:
        return nil, ErrPipelineShutdown
    }
}

func (i *inmemPipeline) Consumer() <-chan AppendFuture {
    return i.doneCh
}

func (i *inmemPipeline) Close() error {
    i.shutdownLock.Lock()
    defer i.shutdownLock.Unlock()
    if i.shutdown {
        return nil
    }

    i.shutdown = true
    close(i.shutdownCh)
    return nil
}
```

../raft/inmem_transport_test.go

```
package raft

import (
    "github.com/stretchr/testify/require"
    "testing"
    "time"
)

func TestInmemTransportImpl(t *testing.T) {
    var inm interface{} = &InmemTransport{}
    if _, ok := inm.(Transport); !ok {
        t.Fatalf("InmemTransport is not a Transport")
    }
    if _, ok := inm.(LoopbackTransport); !ok {
        t.Fatalf("InmemTransport is not a Loopback Transport")
    }
    if _, ok := inm.(WithPeers); !ok {
        t.Fatalf("InmemTransport is not a WithPeers Transport")
    }
}

func TestInmemTransportWriteTimeout(t *testing.T) {
    // InmemTransport should timeout if the other end has gone away
    // when it tries to send a request.
    // Use unbuffered channels so that we can see the write failing
    // without having to contrive to fill up the buffer first.
    timeout := 10 * time.Millisecond
    t1 := &InmemTransport{
        consumerCh: make(chan RPC),
        localAddr:  NewInmemAddr(),
        peers:      make(map[ServerAddress]*InmemTransport),
        timeout:    timeout,
    }
    t2 := &InmemTransport{
        consumerCh: make(chan RPC),
        localAddr:  NewInmemAddr(),
        peers:      make(map[ServerAddress]*InmemTransport),
        timeout:    timeout,
    }
    a2 := t2.LocalAddr()
    t1.Connect(a2, t2)

    stop := make(chan struct{})
    stopped := make(chan struct{})
    go func() {
        defer close(stopped)
        var i uint64
        for {
            select {
            case <-stop:
                return
            case rpc := <-t2.Consumer():
```

```
                i++
                rpc.Respond(&AppendEntriesResponse{
                    Success: true,
                    LastLog: i,
                }, nil)
            }
        }
    }()

    var resp AppendEntriesResponse
    // Sanity check that sending is working before stopping the
    // responder.
    err := t1.AppendEntries("server1", a2, &AppendEntriesRequest{}, &resp)
    NoErr(err, t)
    require.True(t, resp.LastLog == 1)

    close(stop)
    select {
    case <-stopped:
    case <-time.After(time.Second):
        t.Fatalf("timed out waiting for responder to stop")
    }

    err = t1.AppendEntries("server1", a2, &AppendEntriesRequest{}, &resp)
    if err == nil {
        t.Fatalf("expected AppendEntries to time out")
    }
    if err.Error() != "send timed out" {
        t.Fatalf("unexpected error: %v", err)
    }
}
```

../raft/integ_test.go

```
package raft

import (
    "bytes"
    "fmt"
    "io/ioutil"
    "os"
    "testing"
    "time"

    "github.com/hashicorp/go-hclog"
)

// CheckInteg will skip a test if integration testing is not enabled.
func CheckInteg(t *testing.T) {
    if !IsInteg() {
        t.SkipNow()
    }
}

// IsInteg returns a boolean telling you if we're in integ testing mode.
func IsInteg() bool {
    return os.Getenv("INTEG_TESTS") != ""
}

type RaftEnv struct {
    dir      string
    conf     *Config
    fsm      *MockFSM
    store    *InmemStore
    snapshot *FileSnapshotStore
    trans    *NetworkTransport
    raft     *Raft
    logger   hclog.Logger
}

// Release shuts down and cleans up any stored data, its not restartable after
this
func (r *RaftEnv) Release() {
    r.Shutdown()
    os.RemoveAll(r.dir)
}

// Shutdown shuts down raft & transport, but keeps track of its data, its
restartable
// after a Shutdown() by calling Start()
func (r *RaftEnv) Shutdown() {
    r.logger.Warn(fmt.Sprintf("Shutdown node at %v", r.raft.localAddr))
    f := r.raft.Shutdown()
    if err := f.Error(); err != nil {
        panic(err)
    }
```

```
    r.trans.Close()
}

// Restart will start a raft node that was previously Shutdown()
func (r *RaftEnv) Restart(t *testing.T) {
    trans, err := NewTCPTransport(string(r.raft.localAddr), nil, 2,
time.Second, nil)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    r.trans = trans
    r.logger.Info("starting node", "addr", trans.LocalAddr())
    raft, err := NewRaft(r.conf, r.fsm, r.store, r.store, r.snapshot, r.trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    r.raft = raft
}

func MakeRaft(t *testing.T, conf *Config, bootstrap bool) *RaftEnv {
    // Set the config
    if conf == nil {
        conf = inmemConfig(t)
    }

    dir, err := ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }

    stable := NewInmemStore()

    snap, err := NewFileSnapshotStore(dir, 3, nil)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    env := &RaftEnv{
        conf:     conf,
        dir:      dir,
        store:    stable,
        snapshot: snap,
        fsm:      &MockFSM{},
    }
    trans, err := NewTCPTransport("127.0.0.1:0", nil, 2, time.Second, nil)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    env.logger = hclog.New(&hclog.LoggerOptions{
        Name: string(trans.LocalAddr()) + " :",
    })
    env.trans = trans
```

```
    if bootstrap {
        var configuration Configuration
        configuration.Servers = append(configuration.Servers, Server{
            Suffrage: Voter,
            ID:       conf.LocalID,
            Address:  trans.LocalAddr(),
        })
        err = BootstrapCluster(conf, stable, stable, snap, trans,
configuration)
        if err != nil {
            t.Fatalf("err: %v", err)
        }
    }
    env.logger.Info("starting node", "addr", trans.LocalAddr())
    conf.Logger = env.logger
    raft, err := NewRaft(conf, env.fsm, stable, stable, snap, trans)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    env.raft = raft
    return env
}

func WaitFor(env *RaftEnv, state RaftState) error {
    limit := time.Now().Add(200 * time.Millisecond)
    for env.raft.State() != state {
        if time.Now().Before(limit) {
            time.Sleep(10 * time.Millisecond)
        } else {
            return fmt.Errorf("failed to transition to state %v", state)
        }
    }
    return nil
}

func WaitForAny(state RaftState, envs []*RaftEnv) (*RaftEnv, error) {
    limit := time.Now().Add(200 * time.Millisecond)
CHECK:
    for _, env := range envs {
        if env.raft.State() == state {
            return env, nil
        }
    }
    if time.Now().Before(limit) {
        goto WAIT
    }
    return nil, fmt.Errorf("failed to find node in %v state", state)
WAIT:
    time.Sleep(10 * time.Millisecond)
    goto CHECK
}
```

```
func WaitFuture(f Future, t *testing.T) error {
    timer := time.AfterFunc(200*time.Millisecond, func() {
        panic(fmt.Errorf("timeout waiting for future %v", f))
    })
    defer timer.Stop()
    return f.Error()
}

func NoErr(err error, t *testing.T) {
    if err != nil {
        t.Fatalf("err: %v", err)
    }
}

func CheckConsistent(envs []*RaftEnv, t *testing.T) {
    limit := time.Now().Add(400 * time.Millisecond)
    first := envs[0]
    first.fsm.Lock()
    defer first.fsm.Unlock()
    var err error
CHECK:
    l1 := len(first.fsm.logs)
    for i := 1; i < len(envs); i++ {
        env := envs[i]
        env.fsm.Lock()
        l2 := len(env.fsm.logs)
        if l1 != l2 {
            err = fmt.Errorf("log length mismatch %d %d", l1, l2)
            env.fsm.Unlock()
            goto ERR
        }
        for idx, log := range first.fsm.logs {
            other := env.fsm.logs[idx]
            if bytes.Compare(log, other) != 0 {
                err = fmt.Errorf("log entry %d mismatch between %s/%s : '%s' /
'%s'", idx, first.raft.localAddr, env.raft.localAddr, log, other)
                env.fsm.Unlock()
                goto ERR
            }
        }
        env.fsm.Unlock()
    }
    return
ERR:
    if time.Now().After(limit) {
        t.Fatalf("%v", err)
    }
    first.fsm.Unlock()
    time.Sleep(20 * time.Millisecond)
    first.fsm.Lock()
    goto CHECK
}
```

```
// return a log entry that's at least sz long that has the prefix 'test i '
func logBytes(i, sz int) []byte {
    var logBuffer bytes.Buffer
    fmt.Fprintf(&logBuffer, "test %d ", i)
    for logBuffer.Len() < sz {
        logBuffer.WriteByte('x')
    }
    return logBuffer.Bytes()
}

// Tests Raft by creating a cluster, growing it to 5 nodes while
// causing various stressful conditions
func TestRaft_Integ(t *testing.T) {
    CheckInteg(t)
    conf := DefaultConfig()
    conf.LocalID = ServerID("first")
    conf.HeartbeatTimeout = 50 * time.Millisecond
    conf.ElectionTimeout = 50 * time.Millisecond
    conf.LeaderLeaseTimeout = 50 * time.Millisecond
    conf.CommitTimeout = 5 * time.Millisecond
    conf.SnapshotThreshold = 100
    conf.TrailingLogs = 10

    // Create a single node
    env1 := MakeRaft(t, conf, true)
    NoErr(WaitFor(env1, Leader), t)

    totalApplied := 0
    applyAndWait := func(leader *RaftEnv, n, sz int) {
        // Do some commits
        var futures []ApplyFuture
        for i := 0; i < n; i++ {
            futures = append(futures, leader.raft.Apply(logBytes(i, sz), 0))
        }
        for _, f := range futures {
            NoErr(WaitFuture(f, t), t)
            leader.logger.Debug("applied", "index", f.Index(), "size", sz)
        }
        totalApplied += n
    }
    // Do some commits
    applyAndWait(env1, 100, 10)

    // Do a snapshot
    NoErr(WaitFuture(env1.raft.Snapshot(), t), t)

    // Join a few nodes!
    var envs []*RaftEnv
    for i := 0; i < 4; i++ {
        conf.LocalID = ServerID(fmt.Sprintf("next-batch-%d", i))
        env := MakeRaft(t, conf, false)
        addr := env.trans.LocalAddr()
        NoErr(WaitFuture(env1.raft.AddVoter(conf.LocalID, addr, 0, 0), t), t)
```

```
        envs = append(envs, env)
    }

    // Wait for a leader
    leader, err := WaitForAny(Leader, append([]*RaftEnv{env1}, envs...))
    NoErr(err, t)

    // Do some more commits
    applyAndWait(leader, 100, 10)

    // Snapshot the leader
    NoErr(WaitFuture(leader.raft.Snapshot(), t), t)

    CheckConsistent(append([]*RaftEnv{env1}, envs...), t)

    // shutdown a follower
    disconnected := envs[len(envs)-1]
    disconnected.Shutdown()

    // Do some more commits [make sure the resulting snapshot will be a
reasonable size]
    applyAndWait(leader, 100, 10000)

    // snapshot the leader [leaders log should be compacted past the
disconnected follower log now]
    NoErr(WaitFuture(leader.raft.Snapshot(), t), t)

    // Unfortunately we need to wait for the leader to start backing off RPCs
to the down follower
    // such that when the follower comes back up it'll run an election before
it gets an rpc from
    // the leader
    time.Sleep(time.Second * 5)

    // start the now out of date follower back up
    disconnected.Restart(t)

    // wait for it to get caught up
    timeout := time.Now().Add(time.Second * 10)
    for disconnected.raft.getLastApplied() < leader.raft.getLastApplied() {
        time.Sleep(time.Millisecond)
        if time.Now().After(timeout) {
            t.Fatalf("Gave up waiting for follower to get caught up to leader")
        }
    }

    CheckConsistent(append([]*RaftEnv{env1}, envs...), t)

    // Shoot two nodes in the head!
    rm1, rm2 := envs[0], envs[1]
    rm1.Release()
    rm2.Release()
    envs = envs[2:]
```

```
    time.Sleep(10 * time.Millisecond)

    // Wait for a leader
    leader, err = WaitForAny(Leader, append([]*RaftEnv{env1}, envs...))
    NoErr(err, t)

    // Do some more commits
    applyAndWait(leader, 100, 10)

    // Join a few new nodes!
    for i := 0; i < 2; i++ {
        conf.LocalID = ServerID(fmt.Sprintf("final-batch-%d", i))
        env := MakeRaft(t, conf, false)
        addr := env.trans.LocalAddr()
        NoErr(WaitFuture(env1.raft.AddVoter(conf.LocalID, addr, 0, 0), t), t)
        envs = append(envs, env)
    }

    // Wait for a leader
    leader, err = WaitForAny(Leader, append([]*RaftEnv{env1}, envs...))
    NoErr(err, t)

    // Remove the old nodes
    NoErr(WaitFuture(leader.raft.RemoveServer(rm1.raft.localID, 0, 0), t), t)
    NoErr(WaitFuture(leader.raft.RemoveServer(rm2.raft.localID, 0, 0), t), t)

    // Shoot the leader
    env1.Release()
    time.Sleep(3 * conf.HeartbeatTimeout)

    // Wait for a leader
    leader, err = WaitForAny(Leader, envs)
    NoErr(err, t)

    allEnvs := append([]*RaftEnv{env1}, envs...)
    CheckConsistent(allEnvs, t)

    if len(env1.fsm.logs) != totalApplied {
        t.Fatalf("should apply %d logs! %d", totalApplied, len(env1.fsm.logs))
    }

    for _, e := range envs {
        e.Release()
    }
}
```

../raft/log.go

```
package raft

// LogType describes various types of log entries.
type LogType uint8

const (
    // LogCommand is applied to a user FSM.
    LogCommand LogType = iota

    // LogNoop is used to assert leadership.
    LogNoop

    // LogAddPeerDeprecated is used to add a new peer. This should only be used
with
    // older protocol versions designed to be compatible with unversioned
    // Raft servers. See comments in config.go for details.
    LogAddPeerDeprecated

    // LogRemovePeerDeprecated is used to remove an existing peer. This should
only be
    // used with older protocol versions designed to be compatible with
    // unversioned Raft servers. See comments in config.go for details.
    LogRemovePeerDeprecated

    // LogBarrier is used to ensure all preceding operations have been
    // applied to the FSM. It is similar to LogNoop, but instead of returning
    // once committed, it only returns once the FSM manager acks it. Otherwise
    // it is possible there are operations committed but not yet applied to
    // the FSM.
    LogBarrier

    // LogConfiguration establishes a membership change configuration. It is
    // created when a server is added, removed, promoted, etc. Only used
    // when protocol version 1 or greater is in use.
    LogConfiguration
)

// Log entries are replicated to all members of the Raft cluster
// and form the heart of the replicated state machine.
type Log struct {
    // Index holds the index of the log entry.
    Index uint64

    // Term holds the election term of the log entry.
    Term uint64

    // Type holds the type of the log entry.
    Type LogType

    // Data holds the log entry's type-specific data.
    Data []byte
```

```
    // Extensions holds an opaque byte slice of information for middleware. It
    // is up to the client of the library to properly modify this as it adds
    // layers and remove those layers when appropriate. This value is a part of
    // the log, so very large values could cause timing issues.
    //
    // N.B. It is _up to the client_ to handle upgrade paths. For instance if
    // using this with go-raftchunking, the client should ensure that all Raft
    // peers are using a version that can handle that extension before ever
    // actually triggering chunking behavior. It is sometimes sufficient to
    // ensure that non-leaders are upgraded first, then the current leader is
    // upgraded, but a leader changeover during this process could lead to
    // trouble, so gating extension behavior via some flag in the client
    // program is also a good idea.
    Extensions []byte
}

// LogStore is used to provide an interface for storing
// and retrieving logs in a durable fashion.
type LogStore interface {
    // FirstIndex returns the first index written. 0 for no entries.
    FirstIndex() (uint64, error)

    // LastIndex returns the last index written. 0 for no entries.
    LastIndex() (uint64, error)

    // GetLog gets a log entry at a given index.
    GetLog(index uint64, log *Log) error

    // StoreLog stores a log entry.
    StoreLog(log *Log) error

    // StoreLogs stores multiple log entries.
    StoreLogs(logs []*Log) error

    // DeleteRange deletes a range of log entries. The range is inclusive.
    DeleteRange(min, max uint64) error
}
```

../raft/log_cache.go

```
package raft

import (
    "fmt"
    "sync"
)

// LogCache wraps any LogStore implementation to provide an
// in-memory ring buffer. This is used to cache access to
// the recently written entries. For implementations that do not
// cache themselves, this can provide a substantial boost by
// avoiding disk I/O on recent entries.
type LogCache struct {
    store LogStore

    cache []*Log
    l     sync.RWMutex
}

// NewLogCache is used to create a new LogCache with the
// given capacity and backend store.
func NewLogCache(capacity int, store LogStore) (*LogCache, error) {
    if capacity <= 0 {
        return nil, fmt.Errorf("capacity must be positive")
    }
    c := &LogCache{
        store: store,
        cache: make([]*Log, capacity),
    }
    return c, nil
}

func (c *LogCache) GetLog(idx uint64, log *Log) error {
    // Check the buffer for an entry
    c.l.RLock()
    cached := c.cache[idx%uint64(len(c.cache))]
    c.l.RUnlock()

    // Check if entry is valid
    if cached != nil && cached.Index == idx {
        *log = *cached
        return nil
    }

    // Forward request on cache miss
    return c.store.GetLog(idx, log)
}

func (c *LogCache) StoreLog(log *Log) error {
    return c.StoreLogs([]*Log{log})
}
```

```go
func (c *LogCache) StoreLogs(logs []*Log) error {
    // Insert the logs into the ring buffer
    c.l.Lock()
    for _, l := range logs {
        c.cache[l.Index%uint64(len(c.cache))] = l
    }
    c.l.Unlock()

    return c.store.StoreLogs(logs)
}

func (c *LogCache) FirstIndex() (uint64, error) {
    return c.store.FirstIndex()
}

func (c *LogCache) LastIndex() (uint64, error) {
    return c.store.LastIndex()
}

func (c *LogCache) DeleteRange(min, max uint64) error {
    // Invalidate the cache on deletes
    c.l.Lock()
    c.cache = make([]*Log, len(c.cache))
    c.l.Unlock()

    return c.store.DeleteRange(min, max)
}
```

../raft/log_cache_test.go

```
package raft

import (
    "testing"
)

func TestLogCache(t *testing.T) {
    store := NewInmemStore()
    c, _ := NewLogCache(16, store)

    // Insert into the in-mem store
    for i := 0; i < 32; i++ {
        log := &Log{Index: uint64(i) + 1}
        store.StoreLog(log)
    }

    // Check the indexes
    if idx, _ := c.FirstIndex(); idx != 1 {
        t.Fatalf("bad: %d", idx)
    }
    if idx, _ := c.LastIndex(); idx != 32 {
        t.Fatalf("bad: %d", idx)
    }

    // Try get log with a miss
    var out Log
    err := c.GetLog(1, &out)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if out.Index != 1 {
        t.Fatalf("bad: %#v", out)
    }

    // Store logs
    l1 := &Log{Index: 33}
    l2 := &Log{Index: 34}
    err = c.StoreLogs([]*Log{l1, l2})
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    if idx, _ := c.LastIndex(); idx != 34 {
        t.Fatalf("bad: %d", idx)
    }

    // Check that it wrote-through
    err = store.GetLog(33, &out)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    err = store.GetLog(34, &out)
```

```
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Delete in the backend
    err = store.DeleteRange(33, 34)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Should be in the ring buffer
    err = c.GetLog(33, &out)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    err = c.GetLog(34, &out)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Purge the ring buffer
    err = c.DeleteRange(33, 34)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    // Should not be in the ring buffer
    err = c.GetLog(33, &out)
    if err != ErrLogNotFound {
        t.Fatalf("err: %v", err)
    }
    err = c.GetLog(34, &out)
    if err != ErrLogNotFound {
        t.Fatalf("err: %v", err)
    }
}
```

../raft/membership.md

Simon (@superfell) and I (@ongardie) talked through reworking this library's cluster membership changes last Friday. We don't see a way to split this into independent patches, so we're taking the next best approach: submitting the plan here for review, then working on an enormous PR. Your feedback would be appreciated. (@superfell is out this week, however, so don't expect him to respond quickly.)

These are the main goals:

- Bringing things in line with the description in my PhD dissertation;

- Catching up new servers prior to granting them a vote, as well as allowing permanent non-voting members; and
- Eliminating the `peers.json` file, to avoid issues of consistency between that and the log/snapshot.

Data-centric view

We propose to re-define a *configuration* as a set of servers, where each server includes an address (as it does today) and a mode that is either:

- *Voter*: a server whose vote is counted in elections and whose match index is used in advancing the leader's commit index.
- *Nonvoter*: a server that receives log entries but is not considered for elections or commitment purposes.
- *Staging*: a server that acts like a nonvoter with one exception: once a staging server receives enough log entries to catch up sufficiently to the leader's log, the leader will invoke a membership change to change the staging server to a voter.

All changes to the configuration will be done by writing a new configuration to the log. The new configuration will be in affect as soon as it is appended to the log (not when it is committed like a normal state machine command). Note that, per my dissertation, there can be at most one uncommitted configuration at a time (the next configuration may not be created until the prior one has been committed). It's not strictly necessary to follow these same rules for the nonvoter/staging servers, but we think its best to treat all changes uniformly.

Each server will track two configurations:

1. its *committed configuration*: the latest configuration in the log/snapshot that has been committed, along with its index.
2. its *latest configuration*: the latest configuration in the log/snapshot (may be committed or uncommitted), along with its index.

When there's no membership change happening, these two will be the same. The latest configuration is almost always the one used, except:

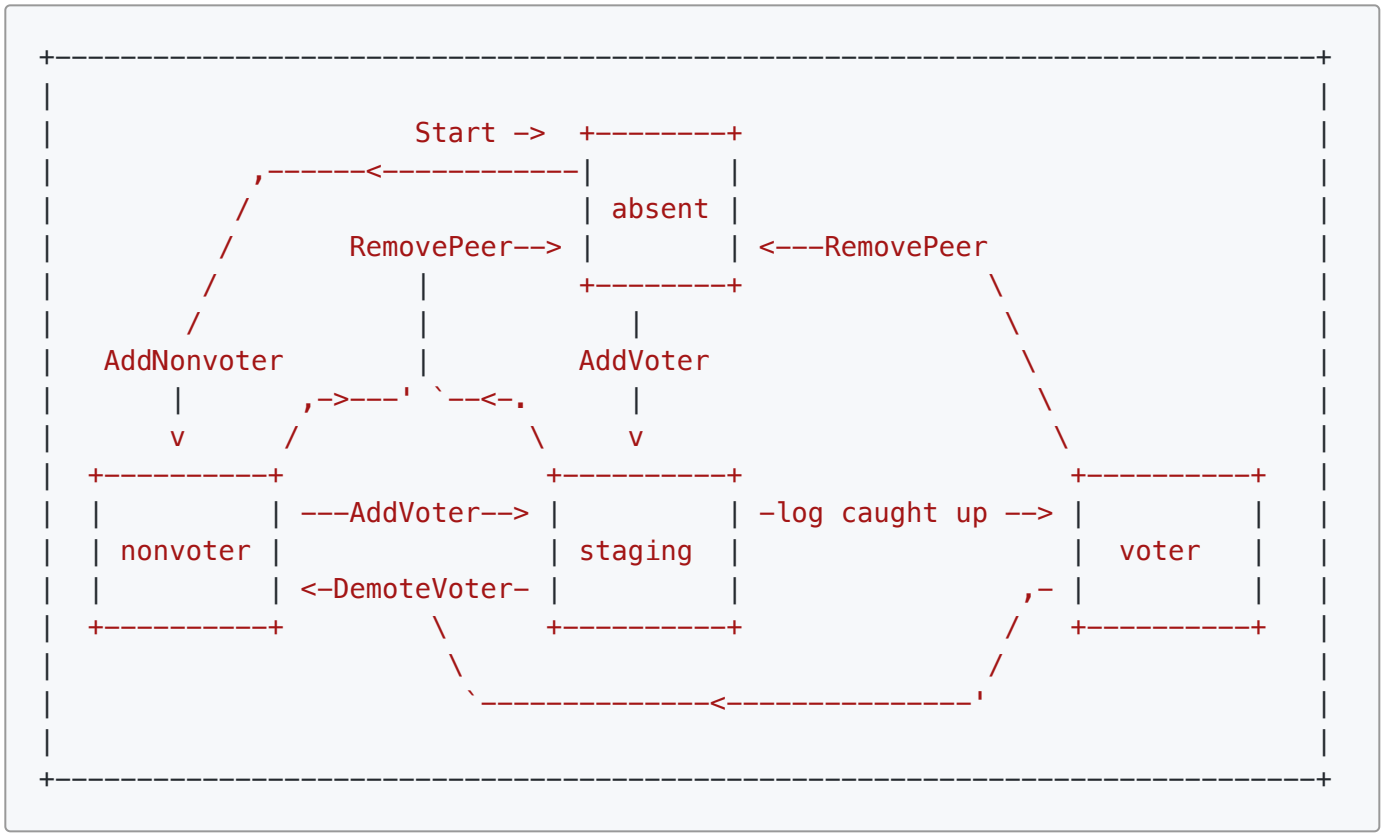
- When followers truncate the suffix of their logs, they may need to fall back to the committed configuration.
- When snapshotting, the committed configuration is written, to correspond with the committed log prefix that is being snapshotted.

Application API

We propose the following operations for clients to manipulate the cluster configuration:

- AddVoter: server becomes staging unless voter,
- AddNonvoter: server becomes nonvoter unless staging or voter,
- DemoteVoter: server becomes nonvoter unless absent,
- RemovePeer: server removed from configuration,
- GetConfiguration: waits for latest config to commit, returns committed config.

This diagram, of which I'm quite proud, shows the possible transitions:



While these operations aren't quite symmetric, we think they're a good set to capture the possible intent of the user. For example, if I want to make sure a server doesn't have a vote, but the server isn't part of the configuration at all, it probably shouldn't be added as a nonvoting server.

Each of these application-level operations will be interpreted by the leader and, if it has an effect, will cause the leader to write a new configuration entry to its log. Which particular application-level operation caused the log entry to be written need not be part of the log entry.

Code implications

This is a non-exhaustive list, but we came up with a few things:

- Remove the PeerStore: the `peers.json` file introduces the possibility of getting out of sync with the log and snapshot, and it's hard to maintain this atomically as the log

changes. It's not clear whether it's meant to track the committed or latest configuration, either.
- Servers will have to search their snapshot and log to find the committed configuration and the latest configuration on startup.
- Bootstrap will no longer use `peers.json` but should initialize the log or snapshot with an application-provided configuration entry.
- Snapshots should store the index of their configuration along with the configuration itself. In my experience with LogCabin, the original log index of the configuration is very useful to include in debug log messages.
- As noted in hashicorp/raft#84, configuration change requests should come in via a separate channel, and one may not proceed until the last has been committed.
- As to deciding when a log is sufficiently caught up, implementing a sophisticated algorithm *is* something that can be done in a separate PR. An easy and decent placeholder is: once the staging server has reached 95% of the leader's commit index, promote it.

Feedback

Again, we're looking for feedback here before we start working on this. Here are some questions to think about:

- Does this seem like where we want things to go?
- Is there anything here that should be left out?
- Is there anything else we're forgetting about?
- Is there a good way to break this up?
- What do we need to worry about in terms of backwards compatibility?
- What implication will this have on current tests?
- What's the best way to test this code, in particular the small changes that will be sprinkled all over the library?

../raft/net_transport.go

```
package raft

import (
    "bufio"
    "context"
    "errors"
    "fmt"
    "github.com/hashicorp/go-hclog"
    "io"
    "net"
    "os"
    "sync"
    "time"

    "github.com/hashicorp/go-msgpack/codec"
)

const (
    rpcAppendEntries uint8 = iota
    rpcRequestVote
    rpcInstallSnapshot
    rpcTimeoutNow

    // DefaultTimeoutScale is the default TimeoutScale in a NetworkTransport.
    DefaultTimeoutScale = 256 * 1024 // 256KB

    // rpcMaxPipeline controls the maximum number of outstanding
    // AppendEntries RPC calls.
    rpcMaxPipeline = 128
)

var (
    // ErrTransportShutdown is returned when operations on a transport are
    // invoked after it's been terminated.
    ErrTransportShutdown = errors.New("transport shutdown")

    // ErrPipelineShutdown is returned when the pipeline is closed.
    ErrPipelineShutdown = errors.New("append pipeline closed")
)

/*

NetworkTransport provides a network based transport that can be
used to communicate with Raft on remote machines. It requires
an underlying stream layer to provide a stream abstraction, which can
be simple TCP, TLS, etc.

This transport is very simple and lightweight. Each RPC request is
framed by sending a byte that indicates the message type, followed
by the MsgPack encoded request.

The response is an error string followed by the response object,
```

```
both are encoded using MsgPack.

InstallSnapshot is special, in that after the RPC request we stream
the entire state. That socket is not re-used as the connection state
is not known if there is an error.

*/
type NetworkTransport struct {
    connPool     map[ServerAddress][]*netConn
    connPoolLock sync.Mutex

    consumeCh chan RPC

    heartbeatFn     func(RPC)
    heartbeatFnLock sync.Mutex

    logger hclog.Logger

    maxPool int

    serverAddressProvider ServerAddressProvider

    shutdown     bool
    shutdownCh   chan struct{}
    shutdownLock sync.Mutex

    stream StreamLayer

    // streamCtx is used to cancel existing connection handlers.
    streamCtx     context.Context
    streamCancel  context.CancelFunc
    streamCtxLock sync.RWMutex

    timeout      time.Duration
    TimeoutScale int
}

// NetworkTransportConfig encapsulates configuration for the network transport
layer.
type NetworkTransportConfig struct {
    // ServerAddressProvider is used to override the target address when
establishing a connection to invoke an RPC
    ServerAddressProvider ServerAddressProvider

    Logger hclog.Logger

    // Dialer
    Stream StreamLayer

    // MaxPool controls how many connections we will pool
    MaxPool int

    // Timeout is used to apply I/O deadlines. For InstallSnapshot, we multiply
```

```
    // the timeout by (SnapshotSize / TimeoutScale).
    Timeout time.Duration
}

// ServerAddressProvider is a target address to which we invoke an RPC when
establishing a connection
type ServerAddressProvider interface {
    ServerAddr(id ServerID) (ServerAddress, error)
}

// StreamLayer is used with the NetworkTransport to provide
// the low level stream abstraction.
type StreamLayer interface {
    net.Listener

    // Dial is used to create a new outgoing connection
    Dial(address ServerAddress, timeout time.Duration) (net.Conn, error)
}

type netConn struct {
    target ServerAddress
    conn   net.Conn
    r      *bufio.Reader
    w      *bufio.Writer
    dec    *codec.Decoder
    enc    *codec.Encoder
}

func (n *netConn) Release() error {
    return n.conn.Close()
}

type netPipeline struct {
    conn  *netConn
    trans *NetworkTransport

    doneCh       chan AppendFuture
    inprogressCh chan *appendFuture

    shutdown     bool
    shutdownCh   chan struct{}
    shutdownLock sync.Mutex
}

// NewNetworkTransportWithConfig creates a new network transport with the given
config struct
func NewNetworkTransportWithConfig(
    config *NetworkTransportConfig,
) *NetworkTransport {
    if config.Logger == nil {
        config.Logger = hclog.New(&hclog.LoggerOptions{
            Name:   "raft-net",
            Output: hclog.DefaultOutput,
```

```
            Level:  hclog.DefaultLevel,
        })
    }
    trans := &NetworkTransport{
        connPool:              make(map[ServerAddress][]*netConn),
        consumeCh:             make(chan RPC),
        logger:                config.Logger,
        maxPool:               config.MaxPool,
        shutdownCh:            make(chan struct{}),
        stream:                config.Stream,
        timeout:               config.Timeout,
        TimeoutScale:          DefaultTimeoutScale,
        serverAddressProvider: config.ServerAddressProvider,
    }

    // Create the connection context and then start our listener.
    trans.setupStreamContext()
    go trans.listen()

    return trans
}

// NewNetworkTransport creates a new network transport with the given dialer
// and listener. The maxPool controls how many connections we will pool. The
// timeout is used to apply I/O deadlines. For InstallSnapshot, we multiply
// the timeout by (SnapshotSize / TimeoutScale).
func NewNetworkTransport(
    stream StreamLayer,
    maxPool int,
    timeout time.Duration,
    logOutput io.Writer,
) *NetworkTransport {
    if logOutput == nil {
        logOutput = os.Stderr
    }
    logger := hclog.New(&hclog.LoggerOptions{
        Name:   "raft-net",
        Output: logOutput,
        Level:  hclog.DefaultLevel,
    })
    config := &NetworkTransportConfig{Stream: stream, MaxPool: maxPool,
Timeout: timeout, Logger: logger}
    return NewNetworkTransportWithConfig(config)
}

// NewNetworkTransportWithLogger creates a new network transport with the given
logger, dialer
// and listener. The maxPool controls how many connections we will pool. The
// timeout is used to apply I/O deadlines. For InstallSnapshot, we multiply
// the timeout by (SnapshotSize / TimeoutScale).
func NewNetworkTransportWithLogger(
    stream StreamLayer,
    maxPool int,
```

```
    timeout time.Duration,
    logger hclog.Logger,
) *NetworkTransport {
    config := &NetworkTransportConfig{Stream: stream, MaxPool: maxPool,
Timeout: timeout, Logger: logger}
    return NewNetworkTransportWithConfig(config)
}

// setupStreamContext is used to create a new stream context. This should be
// called with the stream lock held.
func (n *NetworkTransport) setupStreamContext() {
    ctx, cancel := context.WithCancel(context.Background())
    n.streamCtx = ctx
    n.streamCancel = cancel
}

// getStreamContext is used retrieve the current stream context.
func (n *NetworkTransport) getStreamContext() context.Context {
    n.streamCtxLock.RLock()
    defer n.streamCtxLock.RUnlock()
    return n.streamCtx
}

// SetHeartbeatHandler is used to setup a heartbeat handler
// as a fast-pass. This is to avoid head-of-line blocking from
// disk IO.
func (n *NetworkTransport) SetHeartbeatHandler(cb func(rpc RPC)) {
    n.heartbeatFnLock.Lock()
    defer n.heartbeatFnLock.Unlock()
    n.heartbeatFn = cb
}

// CloseStreams closes the current streams.
func (n *NetworkTransport) CloseStreams() {
    n.connPoolLock.Lock()
    defer n.connPoolLock.Unlock()

    // Close all the connections in the connection pool and then remove their
    // entry.
    for k, e := range n.connPool {
        for _, conn := range e {
            conn.Release()
        }

        delete(n.connPool, k)
    }

    // Cancel the existing connections and create a new context. Both these
    // operations must always be done with the lock held otherwise we can
create
    // connection handlers that are holding a context that will never be
    // cancelable.
    n.streamCtxLock.Lock()
```

```
    n.streamCancel()
    n.setupStreamContext()
    n.streamCtxLock.Unlock()
}

// Close is used to stop the network transport.
func (n *NetworkTransport) Close() error {
    n.shutdownLock.Lock()
    defer n.shutdownLock.Unlock()

    if !n.shutdown {
        close(n.shutdownCh)
        n.stream.Close()
        n.shutdown = true
    }
    return nil
}

// Consumer implements the Transport interface.
func (n *NetworkTransport) Consumer() <-chan RPC {
    return n.consumeCh
}

// LocalAddr implements the Transport interface.
func (n *NetworkTransport) LocalAddr() ServerAddress {
    return ServerAddress(n.stream.Addr().String())
}

// IsShutdown is used to check if the transport is shutdown.
func (n *NetworkTransport) IsShutdown() bool {
    select {
    case <-n.shutdownCh:
        return true
    default:
        return false
    }
}

// getExistingConn is used to grab a pooled connection.
func (n *NetworkTransport) getPooledConn(target ServerAddress) *netConn {
    n.connPoolLock.Lock()
    defer n.connPoolLock.Unlock()

    conns, ok := n.connPool[target]
    if !ok || len(conns) == 0 {
        return nil
    }

    var conn *netConn
    num := len(conns)
    conn, conns[num-1] = conns[num-1], nil
    n.connPool[target] = conns[:num-1]
    return conn
```

```
}

// getConnFromAddressProvider returns a connection from the server address
provider if available, or defaults to a connection using the target server
address
func (n *NetworkTransport) getConnFromAddressProvider(id ServerID, target
ServerAddress) (*netConn, error) {
    address := n.getProviderAddressOrFallback(id, target)
    return n.getConn(address)
}

func (n *NetworkTransport) getProviderAddressOrFallback(id ServerID, target
ServerAddress) ServerAddress {
    if n.serverAddressProvider != nil {
        serverAddressOverride, err := n.serverAddressProvider.ServerAddr(id)
        if err != nil {
            n.logger.Warn("unable to get address for sever, using fallback
address", "id", id, "fallback", target, "error", err)
        } else {
            return serverAddressOverride
        }
    }
    return target
}

// getConn is used to get a connection from the pool.
func (n *NetworkTransport) getConn(target ServerAddress) (*netConn, error) {
    // Check for a pooled conn
    if conn := n.getPooledConn(target); conn != nil {
        return conn, nil
    }

    // Dial a new connection
    conn, err := n.stream.Dial(target, n.timeout)
    if err != nil {
        return nil, err
    }

    // Wrap the conn
    netConn := &netConn{
        target: target,
        conn:   conn,
        r:      bufio.NewReader(conn),
        w:      bufio.NewWriter(conn),
    }

    // Setup encoder/decoders
    netConn.dec = codec.NewDecoder(netConn.r, &codec.MsgpackHandle{})
    netConn.enc = codec.NewEncoder(netConn.w, &codec.MsgpackHandle{})

    // Done
    return netConn, nil
}
```

```
// returnConn returns a connection back to the pool.
func (n *NetworkTransport) returnConn(conn *netConn) {
    n.connPoolLock.Lock()
    defer n.connPoolLock.Unlock()

    key := conn.target
    conns, _ := n.connPool[key]

    if !n.IsShutdown() && len(conns) < n.maxPool {
        n.connPool[key] = append(conns, conn)
    } else {
        conn.Release()
    }
}

// AppendEntriesPipeline returns an interface that can be used to pipeline
// AppendEntries requests.
func (n *NetworkTransport) AppendEntriesPipeline(id ServerID, target
ServerAddress) (AppendPipeline, error) {
    // Get a connection
    conn, err := n.getConnFromAddressProvider(id, target)
    if err != nil {
        return nil, err
    }

    // Create the pipeline
    return newNetPipeline(n, conn), nil
}

// AppendEntries implements the Transport interface.
func (n *NetworkTransport) AppendEntries(id ServerID, target ServerAddress,
args *AppendEntriesRequest, resp *AppendEntriesResponse) error {
    return n.genericRPC(id, target, rpcAppendEntries, args, resp)
}

// RequestVote implements the Transport interface.
func (n *NetworkTransport) RequestVote(id ServerID, target ServerAddress, args
*RequestVoteRequest, resp *RequestVoteResponse) error {
    return n.genericRPC(id, target, rpcRequestVote, args, resp)
}

// genericRPC handles a simple request/response RPC.
func (n *NetworkTransport) genericRPC(id ServerID, target ServerAddress,
rpcType uint8, args interface{}, resp interface{}) error {
    // Get a conn
    conn, err := n.getConnFromAddressProvider(id, target)
    if err != nil {
        return err
    }

    // Set a deadline
    if n.timeout > 0 {
```

```
        conn.conn.SetDeadline(time.Now().Add(n.timeout))
    }

    // Send the RPC
    if err = sendRPC(conn, rpcType, args); err != nil {
        return err
    }

    // Decode the response
    canReturn, err := decodeResponse(conn, resp)
    if canReturn {
        n.returnConn(conn)
    }
    return err
}

// InstallSnapshot implements the Transport interface.
func (n *NetworkTransport) InstallSnapshot(id ServerID, target ServerAddress,
args *InstallSnapshotRequest, resp *InstallSnapshotResponse, data io.Reader)
error {
    // Get a conn, always close for InstallSnapshot
    conn, err := n.getConnFromAddressProvider(id, target)
    if err != nil {
        return err
    }
    defer conn.Release()

    // Set a deadline, scaled by request size
    if n.timeout > 0 {
        timeout := n.timeout * time.Duration(args.Size/int64(n.TimeoutScale))
        if timeout < n.timeout {
            timeout = n.timeout
        }
        conn.conn.SetDeadline(time.Now().Add(timeout))
    }

    // Send the RPC
    if err = sendRPC(conn, rpcInstallSnapshot, args); err != nil {
        return err
    }

    // Stream the state
    if _, err = io.Copy(conn.w, data); err != nil {
        return err
    }

    // Flush
    if err = conn.w.Flush(); err != nil {
        return err
    }

    // Decode the response, do not return conn
    _, err = decodeResponse(conn, resp)
```

```
    return err
}

// EncodePeer implements the Transport interface.
func (n *NetworkTransport) EncodePeer(id ServerID, p ServerAddress) []byte {
    address := n.getProviderAddressOrFallback(id, p)
    return []byte(address)
}

// DecodePeer implements the Transport interface.
func (n *NetworkTransport) DecodePeer(buf []byte) ServerAddress {
    return ServerAddress(buf)
}

// TimeoutNow implements the Transport interface.
func (n *NetworkTransport) TimeoutNow(id ServerID, target ServerAddress, args
*TimeoutNowRequest, resp *TimeoutNowResponse) error {
    return n.genericRPC(id, target, rpcTimeoutNow, args, resp)
}

// listen is used to handling incoming connections.
func (n *NetworkTransport) listen() {
    const baseDelay = 5 * time.Millisecond
    const maxDelay = 1 * time.Second

    var loopDelay time.Duration
    for {
        // Accept incoming connections
        conn, err := n.stream.Accept()
        if err != nil {
            if loopDelay == 0 {
                loopDelay = baseDelay
            } else {
                loopDelay *= 2
            }

            if loopDelay > maxDelay {
                loopDelay = maxDelay
            }

            if !n.IsShutdown() {
                n.logger.Error("failed to accept connection", "error", err)
            }

            select {
            case <-n.shutdownCh:
                return
            case <-time.After(loopDelay):
                continue
            }
        }
        // No error, reset loop delay
        loopDelay = 0
```

```
        n.logger.Debug("accepted connection", "local-address", n.LocalAddr(),
"remote-address", conn.RemoteAddr().String())

        // Handle the connection in dedicated routine
        go n.handleConn(n.getStreamContext(), conn)
    }
}

// handleConn is used to handle an inbound connection for its lifespan. The
// handler will exit when the passed context is cancelled or the connection is
// closed.
func (n *NetworkTransport) handleConn(connCtx context.Context, conn net.Conn) {
    defer conn.Close()
    r := bufio.NewReader(conn)
    w := bufio.NewWriter(conn)
    dec := codec.NewDecoder(r, &codec.MsgpackHandle{})
    enc := codec.NewEncoder(w, &codec.MsgpackHandle{})

    for {
        select {
        case <-connCtx.Done():
            n.logger.Debug("stream layer is closed")
            return
        default:
        }

        if err := n.handleCommand(r, dec, enc); err != nil {
            if err != io.EOF {
                n.logger.Error("failed to decode incoming command", "error",
err)
            }
            return
        }
        if err := w.Flush(); err != nil {
            n.logger.Error("failed to flush response", "error", err)
            return
        }
    }
}

// handleCommand is used to decode and dispatch a single command.
func (n *NetworkTransport) handleCommand(r *bufio.Reader, dec *codec.Decoder,
enc *codec.Encoder) error {
    // Get the rpc type
    rpcType, err := r.ReadByte()
    if err != nil {
        return err
    }

    // Create the RPC object
    respCh := make(chan RPCResponse, 1)
    rpc := RPC{
```

```
        RespChan: respCh,
    }

    // Decode the command
    isHeartbeat := false
    switch rpcType {
    case rpcAppendEntries:
        var req AppendEntriesRequest
        if err := dec.Decode(&req); err != nil {
            return err
        }
        rpc.Command = &req

        // Check if this is a heartbeat
        if req.Term != 0 && req.Leader != nil &&
            req.PrevLogEntry == 0 && req.PrevLogTerm == 0 &&
            len(req.Entries) == 0 && req.LeaderCommitIndex == 0 {
            isHeartbeat = true
        }

    case rpcRequestVote:
        var req RequestVoteRequest
        if err := dec.Decode(&req); err != nil {
            return err
        }
        rpc.Command = &req

    case rpcInstallSnapshot:
        var req InstallSnapshotRequest
        if err := dec.Decode(&req); err != nil {
            return err
        }
        rpc.Command = &req
        rpc.Reader = io.LimitReader(r, req.Size)

    case rpcTimeoutNow:
        var req TimeoutNowRequest
        if err := dec.Decode(&req); err != nil {
            return err
        }
        rpc.Command = &req

    default:
        return fmt.Errorf("unknown rpc type %d", rpcType)
    }

    // Check for heartbeat fast-path
    if isHeartbeat {
        n.heartbeatFnLock.Lock()
        fn := n.heartbeatFn
        n.heartbeatFnLock.Unlock()
        if fn != nil {
            fn(rpc)
```

```
            goto RESP
        }
    }

    // Dispatch the RPC
    select {
    case n.consumeCh <- rpc:
    case <-n.shutdownCh:
        return ErrTransportShutdown
    }

    // Wait for response
RESP:
    select {
    case resp := <-respCh:
        // Send the error first
        respErr := ""
        if resp.Error != nil {
            respErr = resp.Error.Error()
        }
        if err := enc.Encode(respErr); err != nil {
            return err
        }

        // Send the response
        if err := enc.Encode(resp.Response); err != nil {
            return err
        }
    case <-n.shutdownCh:
        return ErrTransportShutdown
    }
    return nil
}

// decodeResponse is used to decode an RPC response and reports whether
// the connection can be reused.
func decodeResponse(conn *netConn, resp interface{}) (bool, error) {
    // Decode the error if any
    var rpcError string
    if err := conn.dec.Decode(&rpcError); err != nil {
        conn.Release()
        return false, err
    }

    // Decode the response
    if err := conn.dec.Decode(resp); err != nil {
        conn.Release()
        return false, err
    }

    // Format an error if any
    if rpcError != "" {
        return true, fmt.Errorf(rpcError)
```

```
    }
    return true, nil
}

// sendRPC is used to encode and send the RPC.
func sendRPC(conn *netConn, rpcType uint8, args interface{}) error {
    // Write the request type
    if err := conn.w.WriteByte(rpcType); err != nil {
        conn.Release()
        return err
    }

    // Send the request
    if err := conn.enc.Encode(args); err != nil {
        conn.Release()
        return err
    }

    // Flush
    if err := conn.w.Flush(); err != nil {
        conn.Release()
        return err
    }
    return nil
}

// newNetPipeline is used to construct a netPipeline from a given
// transport and connection.
func newNetPipeline(trans *NetworkTransport, conn *netConn) *netPipeline {
    n := &netPipeline{
        conn:         conn,
        trans:        trans,
        doneCh:       make(chan AppendFuture, rpcMaxPipeline),
        inprogressCh: make(chan *appendFuture, rpcMaxPipeline),
        shutdownCh:   make(chan struct{}),
    }
    go n.decodeResponses()
    return n
}

// decodeResponses is a long running routine that decodes the responses
// sent on the connection.
func (n *netPipeline) decodeResponses() {
    timeout := n.trans.timeout
    for {
        select {
        case future := <-n.inprogressCh:
            if timeout > 0 {
                n.conn.conn.SetReadDeadline(time.Now().Add(timeout))
            }

            _, err := decodeResponse(n.conn, future.resp)
            future.respond(err)
```

```
            select {
            case n.doneCh <- future:
            case <-n.shutdownCh:
                return
            }
        case <-n.shutdownCh:
            return
        }
    }
}

// AppendEntries is used to pipeline a new append entries request.
func (n *netPipeline) AppendEntries(args *AppendEntriesRequest, resp
*AppendEntriesResponse) (AppendFuture, error) {
    // Create a new future
    future := &appendFuture{
        start: time.Now(),
        args:  args,
        resp:  resp,
    }
    future.init()

    // Add a send timeout
    if timeout := n.trans.timeout; timeout > 0 {
        n.conn.conn.SetWriteDeadline(time.Now().Add(timeout))
    }

    // Send the RPC
    if err := sendRPC(n.conn, rpcAppendEntries, future.args); err != nil {
        return nil, err
    }

    // Hand-off for decoding, this can also cause back-pressure
    // to prevent too many inflight requests
    select {
    case n.inprogressCh <- future:
        return future, nil
    case <-n.shutdownCh:
        return nil, ErrPipelineShutdown
    }
}

// Consumer returns a channel that can be used to consume complete futures.
func (n *netPipeline) Consumer() <-chan AppendFuture {
    return n.doneCh
}

// Closed is used to shutdown the pipeline connection.
func (n *netPipeline) Close() error {
    n.shutdownLock.Lock()
    defer n.shutdownLock.Unlock()
    if n.shutdown {
        return nil
```

```
    }

    // Release the connection
    n.conn.Release()

    n.shutdown = true
    close(n.shutdownCh)
    return nil
}
```

../raft/net_transport_test.go

```
package raft

import (
    "bytes"
    "fmt"
    "github.com/hashicorp/go-hclog"
    "github.com/stretchr/testify/require"
    "net"
    "reflect"
    "strings"
    "sync"
    "sync/atomic"
    "testing"
    "time"
)

type testAddrProvider struct {
    addr string
}

func (t *testAddrProvider) ServerAddr(id ServerID) (ServerAddress, error) {
    return ServerAddress(t.addr), nil
}

func TestNetworkTransport_CloseStreams(t *testing.T) {
    // Transport 1 is consumer
    trans1, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 2,
time.Second, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans1.Close()
    rpcCh := trans1.Consumer()

    // Make the RPC request
    args := AppendEntriesRequest{
        Term:         10,
        Leader:       []byte("cartman"),
        PrevLogEntry: 100,
        PrevLogTerm:  4,
        Entries: []*Log{
            {
                Index: 101,
                Term:  4,
                Type:  LogNoop,
            },
        },
        LeaderCommitIndex: 90,
    }
    resp := AppendEntriesResponse{
        Term:    4,
        LastLog: 90,
```

```
        Success: true,
    }

    // Listen for a request
    go func() {
        for {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*AppendEntriesRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }
                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                return
            }
        }
    }()

    // Transport 2 makes outbound request, 3 conn pool
    trans2, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 3,
time.Second, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans2.Close()
    var i int
    for i = 0; i < 2; i++ {
        // Create wait group
        wg := &sync.WaitGroup{}
        wg.Add(5)

        appendFunc := func() {
            defer wg.Done()
            var out AppendEntriesResponse
            if err := trans2.AppendEntries("id1", trans1.LocalAddr(), &args,
&out); err != nil {
                t.Fatalf("err: %v", err)
            }

            // Verify the response
            if !reflect.DeepEqual(resp, out) {
                t.Fatalf("command mismatch: %#v %#v", resp, out)
            }
        }

        // Try to do parallel appends, should stress the conn pool
        for i = 0; i < 5; i++ {
            go appendFunc()
        }
```

```
        // Wait for the routines to finish
        wg.Wait()

        // Check the conn pool size
        addr := trans1.LocalAddr()
        if len(trans2.connPool[addr]) != 3 {
            t.Fatalf("Expected 3 pooled conns!")
        }

        if i == 0 {
            trans2.CloseStreams()
            if len(trans2.connPool[addr]) != 0 {
                t.Fatalf("Expected no pooled conns after closing streams!")
            }
        }
    }
}

func TestNetworkTransport_StartStop(t *testing.T) {
    trans, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 2, time.Second,
newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    trans.Close()
}

func TestNetworkTransport_Heartbeat_FastPath(t *testing.T) {
    // Transport 1 is consumer
    trans1, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 2,
time.Second, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans1.Close()

    // Make the RPC request
    args := AppendEntriesRequest{
        Term:   10,
        Leader: []byte("cartman"),
    }
    resp := AppendEntriesResponse{
        Term:    4,
        LastLog: 90,
        Success: true,
    }

    invoked := false
    fastpath := func(rpc RPC) {
        // Verify the command
        req := rpc.Command.(*AppendEntriesRequest)
        if !reflect.DeepEqual(req, &args) {
            t.Fatalf("command mismatch: %#v %#v", *req, args)
```

```
        }

        rpc.Respond(&resp, nil)
        invoked = true
    }
    trans1.SetHeartbeatHandler(fastpath)

    // Transport 2 makes outbound request
    trans2, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 2,
time.Second, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans2.Close()

    var out AppendEntriesResponse
    if err := trans2.AppendEntries("id1", trans1.LocalAddr(), &args, &out); err
!= nil {
        t.Fatalf("err: %v", err)
    }

    // Verify the response
    if !reflect.DeepEqual(resp, out) {
        t.Fatalf("command mismatch: %#v %#v", resp, out)
    }

    // Ensure fast-path is used
    if !invoked {
        t.Fatalf("fast-path not used")
    }
}

func TestNetworkTransport_AppendEntries(t *testing.T) {

    for _, useAddrProvider := range []bool{true, false} {
        // Transport 1 is consumer
        trans1, err := makeTransport(t, useAddrProvider, "127.0.0.1:0")
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer trans1.Close()
        rpcCh := trans1.Consumer()

        // Make the RPC request
        args := AppendEntriesRequest{
            Term:         10,
            Leader:       []byte("cartman"),
            PrevLogEntry: 100,
            PrevLogTerm:  4,
            Entries: []*Log{
                {
                    Index: 101,
                    Term:  4,
```

```
                    Type:  LogNoop,
                },
            },
            LeaderCommitIndex: 90,
        }
        resp := AppendEntriesResponse{
            Term:    4,
            LastLog: 90,
            Success: true,
        }

        // Listen for a request
        go func() {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*AppendEntriesRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }

                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
            }
        }()

        // Transport 2 makes outbound request
        trans2, err := makeTransport(t, useAddrProvider,
string(trans1.LocalAddr()))
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer trans2.Close()

        var out AppendEntriesResponse
        if err := trans2.AppendEntries("id1", trans1.LocalAddr(), &args, &out);
err != nil {
            t.Fatalf("err: %v", err)
        }

        // Verify the response
        if !reflect.DeepEqual(resp, out) {
            t.Fatalf("command mismatch: %#v %#v", resp, out)
        }

    }
}

func TestNetworkTransport_AppendEntriesPipeline(t *testing.T) {

    for _, useAddrProvider := range []bool{true, false} {
```

```
        // Transport 1 is consumer
        trans1, err := makeTransport(t, useAddrProvider, "127.0.0.1:0")
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer trans1.Close()
        rpcCh := trans1.Consumer()

        // Make the RPC request
        args := AppendEntriesRequest{
            Term:         10,
            Leader:       []byte("cartman"),
            PrevLogEntry: 100,
            PrevLogTerm:  4,
            Entries: []*Log{
                {
                    Index: 101,
                    Term:  4,
                    Type:  LogNoop,
                },
            },
            LeaderCommitIndex: 90,
        }
        resp := AppendEntriesResponse{
            Term:    4,
            LastLog: 90,
            Success: true,
        }

        // Listen for a request
        go func() {
            for i := 0; i < 10; i++ {
                select {
                case rpc := <-rpcCh:
                    // Verify the command
                    req := rpc.Command.(*AppendEntriesRequest)
                    if !reflect.DeepEqual(req, &args) {
                        t.Fatalf("command mismatch: %#v %#v", *req, args)
                    }
                    rpc.Respond(&resp, nil)

                case <-time.After(200 * time.Millisecond):
                    t.Fatalf("timeout")
                }
            }
        }()

        // Transport 2 makes outbound request
        trans2, err := makeTransport(t, useAddrProvider,
string(trans1.LocalAddr()))
        if err != nil {
            t.Fatalf("err: %v", err)
        }
```

```
        defer trans2.Close()
        pipeline, err := trans2.AppendEntriesPipeline("id1",
trans1.LocalAddr())
        if err != nil {
            t.Fatalf("err: %v", err)
        }

        for i := 0; i < 10; i++ {
            out := new(AppendEntriesResponse)
            if _, err := pipeline.AppendEntries(&args, out); err != nil {
                t.Fatalf("err: %v", err)
            }
        }

        respCh := pipeline.Consumer()
        for i := 0; i < 10; i++ {
            select {
            case ready := <-respCh:
                // Verify the response
                if !reflect.DeepEqual(&resp, ready.Response()) {
                    t.Fatalf("command mismatch: %#v %#v", &resp,
ready.Response())
                }
            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
            }
        }
        pipeline.Close()

    }
}

func TestNetworkTransport_AppendEntriesPipeline_CloseStreams(t *testing.T) {
    // Transport 1 is consumer
    trans1, err := makeTransport(t, true, "127.0.0.1:0")
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans1.Close()
    rpcCh := trans1.Consumer()

    // Make the RPC request
    args := AppendEntriesRequest{
        Term:         10,
        Leader:       []byte("cartman"),
        PrevLogEntry: 100,
        PrevLogTerm:  4,
        Entries: []*Log{
            {
                Index: 101,
                Term:  4,
                Type:  LogNoop,
            },
```

```
        },
        LeaderCommitIndex: 90,
    }
    resp := AppendEntriesResponse{
        Term:    4,
        LastLog: 90,
        Success: true,
    }

    shutdownCh := make(chan struct{})
    defer close(shutdownCh)

    // Listen for a request
    go func() {
        for {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*AppendEntriesRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }
                rpc.Respond(&resp, nil)

            case <-shutdownCh:
                return
            }
        }
    }()

    // Transport 2 makes outbound request
    trans2, err := makeTransport(t, true, string(trans1.LocalAddr()))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans2.Close()

    for _, cancelStreams := range []bool{true, false} {
        pipeline, err := trans2.AppendEntriesPipeline("id1",
trans1.LocalAddr())
        if err != nil {
            t.Fatalf("err: %v", err)
        }

        for i := 0; i < 100; i++ {
            // On the last one, close the streams on the transport one.
            if cancelStreams && i == 10 {
                trans1.CloseStreams()
                time.Sleep(10 * time.Millisecond)
            }

            out := new(AppendEntriesResponse)
            if _, err := pipeline.AppendEntries(&args, out); err != nil {
```

```
                break
            }
        }

        var futureErr error
        respCh := pipeline.Consumer()
    OUTER:
        for i := 0; i < 100; i++ {
            select {
            case ready := <-respCh:
                if err := ready.Error(); err != nil {
                    futureErr = err
                    break OUTER
                }

                // Verify the response
                if !reflect.DeepEqual(&resp, ready.Response()) {
                    t.Fatalf("command mismatch: %#v %#v %v", &resp,
ready.Response(), ready.Error())
                }
            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout when cancel streams is %v", cancelStreams)
            }
        }

        if cancelStreams && futureErr == nil {
            t.Fatalf("expected an error due to the streams being closed")
        } else if !cancelStreams && futureErr != nil {
            t.Fatalf("unexpected error: %v", futureErr)
        }

        pipeline.Close()
    }
}

func TestNetworkTransport_RequestVote(t *testing.T) {

    for _, useAddrProvider := range []bool{true, false} {
        // Transport 1 is consumer
        trans1, err := makeTransport(t, useAddrProvider, "127.0.0.1:0")
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer trans1.Close()
        rpcCh := trans1.Consumer()

        // Make the RPC request
        args := RequestVoteRequest{
            Term:         20,
            Candidate:    []byte("butters"),
            LastLogIndex: 100,
            LastLogTerm:  19,
        }
```

```
        resp := RequestVoteResponse{
            Term:    100,
            Granted: false,
        }

        // Listen for a request
        go func() {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*RequestVoteRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }

                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
            }
        }()

        // Transport 2 makes outbound request
        trans2, err := makeTransport(t, useAddrProvider,
string(trans1.LocalAddr()))
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer trans2.Close()
        var out RequestVoteResponse
        if err := trans2.RequestVote("id1", trans1.LocalAddr(), &args, &out);
err != nil {
            t.Fatalf("err: %v", err)
        }

        // Verify the response
        if !reflect.DeepEqual(resp, out) {
            t.Fatalf("command mismatch: %#v %#v", resp, out)
        }

    }
}

func TestNetworkTransport_InstallSnapshot(t *testing.T) {

    for _, useAddrProvider := range []bool{true, false} {
        // Transport 1 is consumer
        trans1, err := makeTransport(t, useAddrProvider, "127.0.0.1:0")
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer trans1.Close()
        rpcCh := trans1.Consumer()
```

```
        // Make the RPC request
        args := InstallSnapshotRequest{
            Term:         10,
            Leader:       []byte("kyle"),
            LastLogIndex: 100,
            LastLogTerm:  9,
            Peers:        []byte("blah blah"),
            Size:         10,
        }
        resp := InstallSnapshotResponse{
            Term:    10,
            Success: true,
        }

        // Listen for a request
        go func() {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*InstallSnapshotRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }

                // Try to read the bytes
                buf := make([]byte, 10)
                rpc.Reader.Read(buf)

                // Compare
                if bytes.Compare(buf, []byte("0123456789")) != 0 {
                    t.Fatalf("bad buf %v", buf)
                }

                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
            }
        }()

        // Transport 2 makes outbound request
        trans2, err := makeTransport(t, useAddrProvider,
string(trans1.LocalAddr()))
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer trans2.Close()
        // Create a buffer
        buf := bytes.NewBuffer([]byte("0123456789"))

        var out InstallSnapshotResponse
        if err := trans2.InstallSnapshot("id1", trans1.LocalAddr(), &args,
```

```
&out, buf); err != nil {
            t.Fatalf("err: %v", err)
        }

        // Verify the response
        if !reflect.DeepEqual(resp, out) {
            t.Fatalf("command mismatch: %#v %#v", resp, out)
        }

    }
}

func TestNetworkTransport_EncodeDecode(t *testing.T) {
    // Transport 1 is consumer
    trans1, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 2,
time.Second, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans1.Close()

    local := trans1.LocalAddr()
    enc := trans1.EncodePeer("id1", local)
    dec := trans1.DecodePeer(enc)

    if dec != local {
        t.Fatalf("enc/dec fail: %v %v", dec, local)
    }
}

func TestNetworkTransport_EncodeDecode_AddressProvider(t *testing.T) {
    addressOverride := "127.0.0.1:11111"
    config := &NetworkTransportConfig{MaxPool: 2, Timeout: time.Second, Logger:
newTestLogger(t), ServerAddressProvider: &testAddrProvider{addressOverride}}
    trans1, err := NewTCPTransportWithConfig("127.0.0.1:0", nil, config)
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans1.Close()

    local := trans1.LocalAddr()
    enc := trans1.EncodePeer("id1", local)
    dec := trans1.DecodePeer(enc)

    if dec != ServerAddress(addressOverride) {
        t.Fatalf("enc/dec fail: %v %v", dec, addressOverride)
    }
}

func TestNetworkTransport_PooledConn(t *testing.T) {
    // Transport 1 is consumer
    trans1, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 2,
time.Second, newTestLogger(t))
```

```
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans1.Close()
    rpcCh := trans1.Consumer()

    // Make the RPC request
    args := AppendEntriesRequest{
        Term:         10,
        Leader:       []byte("cartman"),
        PrevLogEntry: 100,
        PrevLogTerm:  4,
        Entries: []*Log{
            {
                Index: 101,
                Term:  4,
                Type:  LogNoop,
            },
        },
        LeaderCommitIndex: 90,
    }
    resp := AppendEntriesResponse{
        Term:    4,
        LastLog: 90,
        Success: true,
    }

    // Listen for a request
    go func() {
        for {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*AppendEntriesRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }
                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                return
            }
        }
    }()

    // Transport 2 makes outbound request, 3 conn pool
    trans2, err := NewTCPTransportWithLogger("127.0.0.1:0", nil, 3,
time.Second, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer trans2.Close()
```

```
    // Create wait group
    wg := &sync.WaitGroup{}
    wg.Add(5)

    appendFunc := func() {
        defer wg.Done()
        var out AppendEntriesResponse
        if err := trans2.AppendEntries("id1", trans1.LocalAddr(), &args, &out);
err != nil {
            t.Fatalf("err: %v", err)
        }

        // Verify the response
        if !reflect.DeepEqual(resp, out) {
            t.Fatalf("command mismatch: %#v %#v", resp, out)
        }
    }

    // Try to do parallel appends, should stress the conn pool
    for i := 0; i < 5; i++ {
        go appendFunc()
    }

    // Wait for the routines to finish
    wg.Wait()

    // Check the conn pool size
    addr := trans1.LocalAddr()
    if len(trans2.connPool[addr]) != 3 {
        t.Fatalf("Expected 3 pooled conns!")
    }
}

func makeTransport(t *testing.T, useAddrProvider bool, addressOverride string)
(*NetworkTransport, error) {
    if useAddrProvider {
        config := &NetworkTransportConfig{MaxPool: 2, Timeout: time.Second,
Logger: newTestLogger(t), ServerAddressProvider:
&testAddrProvider{addressOverride}}
        return NewTCPTransportWithConfig("127.0.0.1:0", nil, config)
    }
    return NewTCPTransportWithLogger("127.0.0.1:0", nil, 2, time.Second,
newTestLogger(t))
}

type testCountingWriter struct {
    t        *testing.T
    numCalls *int32
}

func (tw testCountingWriter) Write(p []byte) (n int, err error) {
    atomic.AddInt32(tw.numCalls, 1)
    if !strings.Contains(string(p), "failed to accept connection") {
```

```
        tw.t.Error("did not receive expected log message")
    }
    tw.t.Log("countingWriter:", string(p))
    return len(p), nil
}

type testCountingStreamLayer struct {
    numCalls *int32
}

func (sl testCountingStreamLayer) Accept() (net.Conn, error) {
    *sl.numCalls++
    return nil, fmt.Errorf("intentional error in test")
}

func (sl testCountingStreamLayer) Close() error {
    return nil
}

func (sl testCountingStreamLayer) Addr() net.Addr {
    panic("not needed")
}

func (sl testCountingStreamLayer) Dial(address ServerAddress, timeout
time.Duration) (net.Conn, error) {
    return nil, fmt.Errorf("not needed")
}

// TestNetworkTransport_ListenBackoff tests that Accept() errors in
NetworkTransport#listen()
// do not result in a tight loop and spam the log. We verify this here by
counting the number
// of calls against Accept() and the logger
func TestNetworkTransport_ListenBackoff(t *testing.T) {

    // testTime is the amount of time we will allow NetworkTransport#listen()
to run
    // This needs to be long enough that to verify that maxDelay is in force,
    // but not so long as to be obnoxious when running the test suite.
    const testTime = 4 * time.Second

    var numAccepts int32
    var numLogs int32
    countingWriter := testCountingWriter{t, &numLogs}
    countingLogger := hclog.New(&hclog.LoggerOptions{
        Name:   "test",
        Output: countingWriter,
        Level:  hclog.DefaultLevel,
    })
    transport := NetworkTransport{
        logger:     countingLogger,
        stream:     testCountingStreamLayer{&numAccepts},
        shutdownCh: make(chan struct{}),
```

```
    }

    go transport.listen()

    // sleep (+yield) for testTime seconds before asking the accept loop to
shut down
    time.Sleep(testTime)
    transport.Close()

    // Verify that the method exited (but without block this test)
    // maxDelay == 1s, so we will give the routine 1.25s to loop around and
shut down.
    select {
    case <-transport.shutdownCh:
    case <-time.After(1250 * time.Millisecond):
        t.Error("timed out waiting for NetworkTransport to shut down")
    }
    require.True(t, transport.shutdown)

    // In testTime==4s, we expect to loop approximately 12 times
    // with the following delays (in ms):
    //   0+5+10+20+40+80+160+320+640+1000+1000+1000 == 4275 ms
    // Too few calls suggests that the minDelay is not in force; too many calls
suggests that the
    // maxDelay is not in force or that the back-off isn't working at all.
    // We'll leave a little flex; the important thing here is the asymptotic
behavior.
    // If the minDelay or maxDelay in NetworkTransport#listen() are modified,
this test may fail
    // and need to be adjusted.
    require.True(t, numAccepts > 10)
    require.True(t, numAccepts < 13)
    require.True(t, numLogs > 10)
    require.True(t, numLogs < 13)
}
```

../raft/observer.go

```
package raft

import (
    "sync/atomic"
)

// Observation is sent along the given channel to observers when an event
occurs.
type Observation struct {
    // Raft holds the Raft instance generating the observation.
    Raft *Raft
    // Data holds observation-specific data. Possible types are
    // *RequestVoteRequest
    // RaftState
    // PeerObservation
    // LeaderObservation
    Data interface{}
}

// LeaderObservation is used for the data when leadership changes.
type LeaderObservation struct {
    Leader ServerAddress
}

// PeerObservation is sent to observers when peers change.
type PeerObservation struct {
    Removed bool
    Peer    Server
}

// nextObserverId is used to provide a unique ID for each observer to aid in
// deregistration.
var nextObserverID uint64

// FilterFn is a function that can be registered in order to filter
observations.
// The function reports whether the observation should be included - if
// it returns false, the observation will be filtered out.
type FilterFn func(o *Observation) bool

// Observer describes what to do with a given observation.
type Observer struct {
    // numObserved and numDropped are performance counters for this observer.
    // 64 bit types must be 64 bit aligned to use with atomic operations on
    // 32 bit platforms, so keep them at the top of the struct.
    numObserved uint64
    numDropped  uint64

    // channel receives observations.
    channel chan Observation

    // blocking, if true, will cause Raft to block when sending an observation
```

```
    // to this observer. This should generally be set to false.
    blocking bool

    // filter will be called to determine if an observation should be sent to
    // the channel.
    filter FilterFn

    // id is the ID of this observer in the Raft map.
    id uint64
}

// NewObserver creates a new observer that can be registered
// to make observations on a Raft instance. Observations
// will be sent on the given channel if they satisfy the
// given filter.
//
// If blocking is true, the observer will block when it can't
// send on the channel, otherwise it may discard events.
func NewObserver(channel chan Observation, blocking bool, filter FilterFn)
*Observer {
    return &Observer{
        channel:  channel,
        blocking: blocking,
        filter:   filter,
        id:       atomic.AddUint64(&nextObserverID, 1),
    }
}

// GetNumObserved returns the number of observations.
func (or *Observer) GetNumObserved() uint64 {
    return atomic.LoadUint64(&or.numObserved)
}

// GetNumDropped returns the number of dropped observations due to blocking.
func (or *Observer) GetNumDropped() uint64 {
    return atomic.LoadUint64(&or.numDropped)
}

// RegisterObserver registers a new observer.
func (r *Raft) RegisterObserver(or *Observer) {
    r.observersLock.Lock()
    defer r.observersLock.Unlock()
    r.observers[or.id] = or
}

// DeregisterObserver deregisters an observer.
func (r *Raft) DeregisterObserver(or *Observer) {
    r.observersLock.Lock()
    defer r.observersLock.Unlock()
    delete(r.observers, or.id)
}

// observe sends an observation to every observer.
```

```
func (r *Raft) observe(o interface{}) {
    // In general observers should not block. But in any case this isn't
    // disastrous as we only hold a read lock, which merely prevents
    // registration / deregistration of observers.
    r.observersLock.RLock()
    defer r.observersLock.RUnlock()
    for _, or := range r.observers {
        // It's wasteful to do this in the loop, but for the common case
        // where there are no observers we won't create any objects.
        ob := Observation{Raft: r, Data: o}
        if or.filter != nil && !or.filter(&ob) {
            continue
        }
        if or.channel == nil {
            continue
        }
        if or.blocking {
            or.channel <- ob
            atomic.AddUint64(&or.numObserved, 1)
        } else {
            select {
            case or.channel <- ob:
                atomic.AddUint64(&or.numObserved, 1)
            default:
                atomic.AddUint64(&or.numDropped, 1)
            }
        }
    }
}
```

../raft/peersjson.go

```
package raft

import (
    "bytes"
    "encoding/json"
    "io/ioutil"
)

// ReadPeersJSON consumes a legacy peers.json file in the format of the old
JSON
// peer store and creates a new-style configuration structure. This can be used
// to migrate this data or perform manual recovery when running protocol
versions
// that can interoperate with older, unversioned Raft servers. This should not
be
// used once server IDs are in use, because the old peers.json file didn't have
// support for these, nor non-voter suffrage types.
func ReadPeersJSON(path string) (Configuration, error) {
    // Read in the file.
    buf, err := ioutil.ReadFile(path)
    if err != nil {
        return Configuration{}, err
    }

    // Parse it as JSON.
    var peers []string
    dec := json.NewDecoder(bytes.NewReader(buf))
    if err := dec.Decode(&peers); err != nil {
        return Configuration{}, err
    }

    // Map it into the new-style configuration structure. We can only specify
    // voter roles here, and the ID has to be the same as the address.
    var configuration Configuration
    for _, peer := range peers {
        server := Server{
            Suffrage: Voter,
            ID:       ServerID(peer),
            Address:  ServerAddress(peer),
        }
        configuration.Servers = append(configuration.Servers, server)
    }

    // We should only ingest valid configurations.
    if err := checkConfiguration(configuration); err != nil {
        return Configuration{}, err
    }
    return configuration, nil
}

// configEntry is used when decoding a new-style peers.json.
type configEntry struct {
```

```
    // ID is the ID of the server (a UUID, usually).
    ID ServerID `json:"id"`

    // Address is the host:port of the server.
    Address ServerAddress `json:"address"`

    // NonVoter controls the suffrage. We choose this sense so people
    // can leave this out and get a Voter by default.
    NonVoter bool `json:"non_voter"`
}

// ReadConfigJSON reads a new-style peers.json and returns a configuration
// structure. This can be used to perform manual recovery when running protocol
// versions that use server IDs.
func ReadConfigJSON(path string) (Configuration, error) {
    // Read in the file.
    buf, err := ioutil.ReadFile(path)
    if err != nil {
        return Configuration{}, err
    }

    // Parse it as JSON.
    var peers []configEntry
    dec := json.NewDecoder(bytes.NewReader(buf))
    if err := dec.Decode(&peers); err != nil {
        return Configuration{}, err
    }

    // Map it into the new-style configuration structure.
    var configuration Configuration
    for _, peer := range peers {
        suffrage := Voter
        if peer.NonVoter {
            suffrage = Nonvoter
        }
        server := Server{
            Suffrage: suffrage,
            ID:       peer.ID,
            Address:  peer.Address,
        }
        configuration.Servers = append(configuration.Servers, server)
    }

    // We should only ingest valid configurations.
    if err := checkConfiguration(configuration); err != nil {
        return Configuration{}, err
    }
    return configuration, nil
}
```

../raft/peersjson_test.go

```
package raft

import (
    "io/ioutil"
    "os"
    "path/filepath"
    "reflect"
    "strings"
    "testing"
)

func TestPeersJSON_BadConfiguration(t *testing.T) {
    var err error
    var base string
    base, err = ioutil.TempDir("", "")
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer os.RemoveAll(base)

    peers := filepath.Join(base, "peers.json")
    if err = ioutil.WriteFile(peers, []byte("null"), 0666); err != nil {
        t.Fatalf("err: %v", err)
    }

    _, err = ReadPeersJSON(peers)
    if err == nil || !strings.Contains(err.Error(), "at least one voter") {
        t.Fatalf("err: %v", err)
    }
}

func TestPeersJSON_ReadPeersJSON(t *testing.T) {
    var err error
    var base string
    base, err = ioutil.TempDir("", "")
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer os.RemoveAll(base)

    content := []byte(`
["127.0.0.1:123",
 "127.0.0.2:123",
 "127.0.0.3:123"]
`)
    peers := filepath.Join(base, "peers.json")
    if err = ioutil.WriteFile(peers, content, 0666); err != nil {
        t.Fatalf("err: %v", err)
    }
    var configuration Configuration
    configuration, err = ReadPeersJSON(peers)
    if err != nil {
```

```
        t.Fatalf("err: %v", err)
    }

    expected := Configuration{
        Servers: []Server{
            {
                Suffrage: Voter,
                ID:       ServerID("127.0.0.1:123"),
                Address:  ServerAddress("127.0.0.1:123"),
            },
            {
                Suffrage: Voter,
                ID:       ServerID("127.0.0.2:123"),
                Address:  ServerAddress("127.0.0.2:123"),
            },
            {
                Suffrage: Voter,
                ID:       ServerID("127.0.0.3:123"),
                Address:  ServerAddress("127.0.0.3:123"),
            },
        },
    }
    if !reflect.DeepEqual(configuration, expected) {
        t.Fatalf("bad configuration: %+v != %+v", configuration, expected)
    }
}

func TestPeersJSON_ReadConfigJSON(t *testing.T) {
    var err error
    var base string
    base, err = ioutil.TempDir("", "")
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    defer os.RemoveAll(base)

    content := []byte(`
[
  {
    "id": "adf4238a-882b-9ddc-4a9d-5b6758e4159e",
    "address": "127.0.0.1:123",
    "non_voter": false
  },
  {
    "id": "8b6dda82-3103-11e7-93ae-92361f002671",
    "address": "127.0.0.2:123"
  },
  {
    "id": "97e17742-3103-11e7-93ae-92361f002671",
    "address": "127.0.0.3:123",
    "non_voter": true
  }
]
```

```
`)
    peers := filepath.Join(base, "peers.json")
    if err = ioutil.WriteFile(peers, content, 0666); err != nil {
        t.Fatalf("err: %v", err)
    }

    var configuration Configuration
    configuration, err = ReadConfigJSON(peers)
    if err != nil {
        t.Fatalf("err: %v", err)
    }

    expected := Configuration{
        Servers: []Server{
            {
                Suffrage: Voter,
                ID:       ServerID("adf4238a-882b-9ddc-4a9d-5b6758e4159e"),
                Address:  ServerAddress("127.0.0.1:123"),
            },
            {
                Suffrage: Voter,
                ID:       ServerID("8b6dda82-3103-11e7-93ae-92361f002671"),
                Address:  ServerAddress("127.0.0.2:123"),
            },
            {
                Suffrage: Nonvoter,
                ID:       ServerID("97e17742-3103-11e7-93ae-92361f002671"),
                Address:  ServerAddress("127.0.0.3:123"),
            },
        },
    }
    if !reflect.DeepEqual(configuration, expected) {
        t.Fatalf("bad configuration: %+v != %+v", configuration, expected)
    }
}
```

../raft/raft.go

```
package raft

import (
    "bytes"
    "container/list"
    "fmt"
    "io"
    "io/ioutil"
    "sync/atomic"
    "time"

    "github.com/hashicorp/go-hclog"

    "github.com/armon/go-metrics"
)

const (
    minCheckInterval = 10 * time.Millisecond
)

var (
    keyCurrentTerm  = []byte("CurrentTerm")
    keyLastVoteTerm = []byte("LastVoteTerm")
    keyLastVoteCand = []byte("LastVoteCand")
)

// getRPCHeader returns an initialized RPCHeader struct for the given
// Raft instance. This structure is sent along with RPC requests and
// responses.
func (r *Raft) getRPCHeader() RPCHeader {
    return RPCHeader{
        ProtocolVersion: r.conf.ProtocolVersion,
    }
}

// checkRPCHeader houses logic about whether this instance of Raft can process
// the given RPC message.
func (r *Raft) checkRPCHeader(rpc RPC) error {
    // Get the header off the RPC message.
    wh, ok := rpc.Command.(WithRPCHeader)
    if !ok {
        return fmt.Errorf("RPC does not have a header")
    }
    header := wh.GetRPCHeader()

    // First check is to just make sure the code can understand the
    // protocol at all.
    if header.ProtocolVersion < ProtocolVersionMin ||
        header.ProtocolVersion > ProtocolVersionMax {
        return ErrUnsupportedProtocol
    }
```

```
    // Second check is whether we should support this message, given the
    // current protocol we are configured to run. This will drop support
    // for protocol version 0 starting at protocol version 2, which is
    // currently what we want, and in general support one version back. We
    // may need to revisit this policy depending on how future protocol
    // changes evolve.
    if header.ProtocolVersion < r.conf.ProtocolVersion-1 {
        return ErrUnsupportedProtocol
    }

    return nil
}

// getSnapshotVersion returns the snapshot version that should be used when
// creating snapshots, given the protocol version in use.
func getSnapshotVersion(protocolVersion ProtocolVersion) SnapshotVersion {
    // Right now we only have two versions and they are backwards compatible
    // so we don't need to look at the protocol version.
    return 1
}

// commitTuple is used to send an index that was committed,
// with an optional associated future that should be invoked.
type commitTuple struct {
    log    *Log
    future *logFuture
}

// leaderState is state that is used while we are a leader.
type leaderState struct {
    leadershipTransferInProgress int32 // indicates that a leadership transfer
is in progress.
    commitCh                     chan struct{}
    commitment                   *commitment
    inflight                     *list.List // list of logFuture in log index
order
    replState                    map[ServerID]*followerReplication
    notify                       map[*verifyFuture]struct{}
    stepDown                     chan struct{}
}

// setLeader is used to modify the current leader of the cluster
func (r *Raft) setLeader(leader ServerAddress) {
    r.leaderLock.Lock()
    oldLeader := r.leader
    r.leader = leader
    r.leaderLock.Unlock()
    if oldLeader != leader {
        r.observe(LeaderObservation{Leader: leader})
    }
}

// requestConfigChange is a helper for the above functions that make
```

```
// configuration change requests. 'req' describes the change. For timeout,
// see AddVoter.
func (r *Raft) requestConfigChange(req configurationChangeRequest, timeout
time.Duration) IndexFuture {
    var timer <-chan time.Time
    if timeout > 0 {
        timer = time.After(timeout)
    }
    future := &configurationChangeFuture{
        req: req,
    }
    future.init()
    select {
    case <-timer:
        return errorFuture{ErrEnqueueTimeout}
    case r.configurationChangeCh <- future:
        return future
    case <-r.shutdownCh:
        return errorFuture{ErrRaftShutdown}
    }
}

// run is a long running goroutine that runs the Raft FSM.
func (r *Raft) run() {
    for {
        // Check if we are doing a shutdown
        select {
        case <-r.shutdownCh:
            // Clear the leader to prevent forwarding
            r.setLeader("")
            return
        default:
        }

        // Enter into a sub-FSM
        switch r.getState() {
        case Follower:
            r.runFollower()
        case Candidate:
            r.runCandidate()
        case Leader:
            r.runLeader()
        }
    }
}

// runFollower runs the FSM for a follower.
func (r *Raft) runFollower() {
    didWarn := false
    r.logger.Info("entering follower state", "follower", r, "leader",
r.Leader())
    metrics.IncrCounter([]string{"raft", "state", "follower"}, 1)
    heartbeatTimer := randomTimeout(r.conf.HeartbeatTimeout)
```

```
    for r.getState() == Follower {
        select {
        case rpc := <-r.rpcCh:
            r.processRPC(rpc)

        case c := <-r.configurationChangeCh:
            // Reject any operations since we are not the leader
            c.respond(ErrNotLeader)

        case a := <-r.applyCh:
            // Reject any operations since we are not the leader
            a.respond(ErrNotLeader)

        case v := <-r.verifyCh:
            // Reject any operations since we are not the leader
            v.respond(ErrNotLeader)

        case r := <-r.userRestoreCh:
            // Reject any restores since we are not the leader
            r.respond(ErrNotLeader)

        case r := <-r.leadershipTransferCh:
            // Reject any operations since we are not the leader
            r.respond(ErrNotLeader)

        case c := <-r.configurationsCh:
            c.configurations = r.configurations.Clone()
            c.respond(nil)

        case b := <-r.bootstrapCh:
            b.respond(r.liveBootstrap(b.configuration))

        case <-heartbeatTimer:
            // Restart the heartbeat timer
            heartbeatTimer = randomTimeout(r.conf.HeartbeatTimeout)

            // Check if we have had a successful contact
            lastContact := r.LastContact()
            if time.Now().Sub(lastContact) < r.conf.HeartbeatTimeout {
                continue
            }

            // Heartbeat failed! Transition to the candidate state
            lastLeader := r.Leader()
            r.setLeader("")

            if r.configurations.latestIndex == 0 {
                if !didWarn {
                    r.logger.Warn("no known peers, aborting election")
                    didWarn = true
                }
            } else if r.configurations.latestIndex ==
```

```
r.configurations.committedIndex &&
                !hasVote(r.configurations.latest, r.localID) {
                if !didWarn {
                    r.logger.Warn("not part of stable configuration, aborting
election")
                    didWarn = true
                }
            } else {
                r.logger.Warn("heartbeat timeout reached, starting election",
"last-leader", lastLeader)
                metrics.IncrCounter([]string{"raft", "transition",
"heartbeat_timeout"}, 1)
                r.setState(Candidate)
                return
            }

        case <-r.shutdownCh:
            return
        }
    }
}

// liveBootstrap attempts to seed an initial configuration for the cluster. See
// the Raft object's member BootstrapCluster for more details. This must only
be
// called on the main thread, and only makes sense in the follower state.
func (r *Raft) liveBootstrap(configuration Configuration) error {
    // Use the pre-init API to make the static updates.
    err := BootstrapCluster(&r.conf, r.logs, r.stable, r.snapshots,
        r.trans, configuration)
    if err != nil {
        return err
    }

    // Make the configuration live.
    var entry Log
    if err := r.logs.GetLog(1, &entry); err != nil {
        panic(err)
    }
    r.setCurrentTerm(1)
    r.setLastLog(entry.Index, entry.Term)
    r.processConfigurationLogEntry(&entry)
    return nil
}

// runCandidate runs the FSM for a candidate.
func (r *Raft) runCandidate() {
    r.logger.Info("entering candidate state", "node", r, "term",
r.getCurrentTerm()+1)
    metrics.IncrCounter([]string{"raft", "state", "candidate"}, 1)

    // Start vote for us, and set a timeout
    voteCh := r.electSelf()
```

```
    // Make sure the leadership transfer flag is reset after each run. Having
this
    // flag will set the field LeadershipTransfer in a RequestVoteRequst to
true,
    // which will make other servers vote even though they have a leader
already.
    // It is important to reset that flag, because this privilege could be
abused
    // otherwise.
    defer func() { r.candidateFromLeadershipTransfer = false }()

    electionTimer := randomTimeout(r.conf.ElectionTimeout)

    // Tally the votes, need a simple majority
    grantedVotes := 0
    votesNeeded := r.quorumSize()
    r.logger.Debug("votes", "needed", votesNeeded)

    for r.getState() == Candidate {
        select {
        case rpc := <-r.rpcCh:
            r.processRPC(rpc)

        case vote := <-voteCh:
            // Check if the term is greater than ours, bail
            if vote.Term > r.getCurrentTerm() {
                r.logger.Debug("newer term discovered, fallback to follower")
                r.setState(Follower)
                r.setCurrentTerm(vote.Term)
                return
            }

            // Check if the vote is granted
            if vote.Granted {
                grantedVotes++
                r.logger.Debug("vote granted", "from", vote.voterID, "term",
vote.Term, "tally", grantedVotes)
            }

            // Check if we've become the leader
            if grantedVotes >= votesNeeded {
                r.logger.Info("election won", "tally", grantedVotes)
                r.setState(Leader)
                r.setLeader(r.localAddr)
                return
            }

        case c := <-r.configurationChangeCh:
            // Reject any operations since we are not the leader
            c.respond(ErrNotLeader)

        case a := <-r.applyCh:
```

```
            // Reject any operations since we are not the leader
            a.respond(ErrNotLeader)

        case v := <-r.verifyCh:
            // Reject any operations since we are not the leader
            v.respond(ErrNotLeader)

        case r := <-r.userRestoreCh:
            // Reject any restores since we are not the leader
            r.respond(ErrNotLeader)

        case c := <-r.configurationsCh:
            c.configurations = r.configurations.Clone()
            c.respond(nil)

        case b := <-r.bootstrapCh:
            b.respond(ErrCantBootstrap)

        case <-electionTimer:
            // Election failed! Restart the election. We simply return,
            // which will kick us back into runCandidate
            r.logger.Warn("Election timeout reached, restarting election")
            return

        case <-r.shutdownCh:
            return
        }
    }
}

func (r *Raft) setLeadershipTransferInProgress(v bool) {
    if v {
        atomic.StoreInt32(&r.leaderState.leadershipTransferInProgress, 1)
    } else {
        atomic.StoreInt32(&r.leaderState.leadershipTransferInProgress, 0)
    }
}

func (r *Raft) getLeadershipTransferInProgress() bool {
    v := atomic.LoadInt32(&r.leaderState.leadershipTransferInProgress)
    if v == 1 {
        return true
    }
    return false
}

func (r *Raft) setupLeaderState() {
    r.leaderState.commitCh = make(chan struct{}, 1)
    r.leaderState.commitment = newCommitment(r.leaderState.commitCh,
        r.configurations.latest,
        r.getLastIndex()+1 /* first index that may be committed in this term
*/)
    r.leaderState.inflight = list.New()
```

```
    r.leaderState.replState = make(map[ServerID]*followerReplication)
    r.leaderState.notify = make(map[*verifyFuture]struct{})
    r.leaderState.stepDown = make(chan struct{}, 1)
}

// runLeader runs the FSM for a leader. Do the setup here and drop into
// the leaderLoop for the hot loop.
func (r *Raft) runLeader() {
    r.logger.Info("entering leader state", "leader", r)
    metrics.IncrCounter([]string{"raft", "state", "leader"}, 1)

    // Notify that we are the leader
    asyncNotifyBool(r.leaderCh, true)

    // Push to the notify channel if given
    if notify := r.conf.NotifyCh; notify != nil {
        select {
        case notify <- true:
        case <-r.shutdownCh:
        }
    }

    // setup leader state. This is only supposed to be accessed within the
    // leaderloop.
    r.setupLeaderState()

    // Cleanup state on step down
    defer func() {
        // Since we were the leader previously, we update our
        // last contact time when we step down, so that we are not
        // reporting a last contact time from before we were the
        // leader. Otherwise, to a client it would seem our data
        // is extremely stale.
        r.setLastContact()

        // Stop replication
        for _, p := range r.leaderState.replState {
            close(p.stopCh)
        }

        // Respond to all inflight operations
        for e := r.leaderState.inflight.Front(); e != nil; e = e.Next() {
            e.Value.(*logFuture).respond(ErrLeadershipLost)
        }

        // Respond to any pending verify requests
        for future := range r.leaderState.notify {
            future.respond(ErrLeadershipLost)
        }

        // Clear all the state
        r.leaderState.commitCh = nil
        r.leaderState.commitment = nil
```

```
        r.leaderState.inflight = nil
        r.leaderState.replState = nil
        r.leaderState.notify = nil
        r.leaderState.stepDown = nil

        // If we are stepping down for some reason, no known leader.
        // We may have stepped down due to an RPC call, which would
        // provide the leader, so we cannot always blank this out.
        r.leaderLock.Lock()
        if r.leader == r.localAddr {
            r.leader = ""
        }
        r.leaderLock.Unlock()

        // Notify that we are not the leader
        asyncNotifyBool(r.leaderCh, false)

        // Push to the notify channel if given
        if notify := r.conf.NotifyCh; notify != nil {
            select {
            case notify <- false:
            case <-r.shutdownCh:
                // On shutdown, make a best effort but do not block
                select {
                case notify <- false:
                default:
                }
            }
        }
    }()

    // Start a replication routine for each peer
    r.startStopReplication()

    // Dispatch a no-op log entry first. This gets this leader up to the latest
    // possible commit index, even in the absence of client commands. This used
    // to append a configuration entry instead of a noop. However, that permits
    // an unbounded number of uncommitted configurations in the log. We now
    // maintain that there exists at most one uncommitted configuration entry
in
    // any log, so we have to do proper no-ops here.
    noop := &logFuture{
        log: Log{
            Type: LogNoop,
        },
    }
    r.dispatchLogs([]*logFuture{noop})

    // Sit in the leader loop until we step down
    r.leaderLoop()
}

// startStopReplication will set up state and start asynchronous replication to
```

```
// new peers, and stop replication to removed peers. Before removing a peer,
// it'll instruct the replication routines to try to replicate to the current
// index. This must only be called from the main thread.
func (r *Raft) startStopReplication() {
    inConfig := make(map[ServerID]bool, len(r.configurations.latest.Servers))
    lastIdx := r.getLastIndex()

    // Start replication goroutines that need starting
    for _, server := range r.configurations.latest.Servers {
        if server.ID == r.localID {
            continue
        }
        inConfig[server.ID] = true
        if _, ok := r.leaderState.replState[server.ID]; !ok {
            r.logger.Info("added peer, starting replication", "peer",
server.ID)
            s := &followerReplication{
                peer:                server,
                commitment:          r.leaderState.commitment,
                stopCh:              make(chan uint64, 1),
                triggerCh:           make(chan struct{}, 1),
                triggerDeferErrorCh: make(chan *deferError, 1),
                currentTerm:         r.getCurrentTerm(),
                nextIndex:           lastIdx + 1,
                lastContact:         time.Now(),
                notify:              make(map[*verifyFuture]struct{}),
                notifyCh:            make(chan struct{}, 1),
                stepDown:            r.leaderState.stepDown,
            }
            r.leaderState.replState[server.ID] = s
            r.goFunc(func() { r.replicate(s) })
            asyncNotifyCh(s.triggerCh)
            r.observe(PeerObservation{Peer: server, Removed: false})
        }
    }

    // Stop replication goroutines that need stopping
    for serverID, repl := range r.leaderState.replState {
        if inConfig[serverID] {
            continue
        }
        // Replicate up to lastIdx and stop
        r.logger.Info("removed peer, stopping replication", "peer", serverID,
"last-index", lastIdx)
        repl.stopCh <- lastIdx
        close(repl.stopCh)
        delete(r.leaderState.replState, serverID)
        r.observe(PeerObservation{Peer: repl.peer, Removed: true})
    }
}

// configurationChangeChIfStable returns r.configurationChangeCh if it's safe
// to process requests from it, or nil otherwise. This must only be called
```

```
// from the main thread.
//
// Note that if the conditions here were to change outside of leaderLoop to
take
// this from nil to non-nil, we would need leaderLoop to be kicked.
func (r *Raft) configurationChangeChIfStable() chan *configurationChangeFuture
{
    // Have to wait until:
    // 1. The latest configuration is committed, and
    // 2. This leader has committed some entry (the noop) in this term
    //    https://groups.google.com/forum/#!msg/raft-
dev/t4xj6dJTP6E/d2D9LrWRza8J
    if r.configurations.latestIndex == r.configurations.committedIndex &&
        r.getCommitIndex() >= r.leaderState.commitment.startIndex {
        return r.configurationChangeCh
    }
    return nil
}

// leaderLoop is the hot loop for a leader. It is invoked
// after all the various leader setup is done.
func (r *Raft) leaderLoop() {
    // stepDown is used to track if there is an inflight log that
    // would cause us to lose leadership (specifically a RemovePeer of
    // ourselves). If this is the case, we must not allow any logs to
    // be processed in parallel, otherwise we are basing commit on
    // only a single peer (ourself) and replicating to an undefined set
    // of peers.
    stepDown := false
    lease := time.After(r.conf.LeaderLeaseTimeout)

    for r.getState() == Leader {
        select {
        case rpc := <-r.rpcCh:
            r.processRPC(rpc)

        case <-r.leaderState.stepDown:
            r.setState(Follower)

        case future := <-r.leadershipTransferCh:
            if r.getLeadershipTransferInProgress() {
                r.logger.Debug(ErrLeadershipTransferInProgress.Error())
                future.respond(ErrLeadershipTransferInProgress)
                continue
            }

            r.logger.Debug("starting leadership transfer", "id", future.ID,
"address", future.Address)

            // When we are leaving leaderLoop, we are no longer
            // leader, so we should stop transferring.
            leftLeaderLoop := make(chan struct{})
            defer func() { close(leftLeaderLoop) }()
```

```
            stopCh := make(chan struct{})
            doneCh := make(chan error, 1)

            // This is intentionally being setup outside of the
            // leadershipTransfer function. Because the TimeoutNow
            // call is blocking and there is no way to abort that
            // in case eg the timer expires.
            // The leadershipTransfer function is controlled with
            // the stopCh and doneCh.
            go func() {
                select {
                case <-time.After(r.conf.ElectionTimeout):
                    close(stopCh)
                    err := fmt.Errorf("leadership transfer timeout")
                    r.logger.Debug(err.Error())
                    future.respond(err)
                    <-doneCh
                case <-leftLeaderLoop:
                    close(stopCh)
                    err := fmt.Errorf("lost leadership during transfer
(expected)")
                    r.logger.Debug(err.Error())
                    future.respond(nil)
                    <-doneCh
                case err := <-doneCh:
                    if err != nil {
                        r.logger.Debug(err.Error())
                    }
                    future.respond(err)
                }
            }()

            // leaderState.replState is accessed here before
            // starting leadership transfer asynchronously because
            // leaderState is only supposed to be accessed in the
            // leaderloop.
            id := future.ID
            address := future.Address
            if id == nil {
                s := r.pickServer()
                if s != nil {
                    id = &s.ID
                    address = &s.Address
                } else {
                    doneCh <- fmt.Errorf("cannot find peer")
                    continue
                }
            }
            state, ok := r.leaderState.replState[*id]
            if !ok {
                doneCh <- fmt.Errorf("cannot find replication state for %v",
id)
```

```
                continue
            }

            go r.leadershipTransfer(*id, *address, state, stopCh, doneCh)

        case <-r.leaderState.commitCh:
            // Process the newly committed entries
            oldCommitIndex := r.getCommitIndex()
            commitIndex := r.leaderState.commitment.getCommitIndex()
            r.setCommitIndex(commitIndex)

            // New configration has been committed, set it as the committed
            // value.
            if r.configurations.latestIndex > oldCommitIndex &&
                r.configurations.latestIndex <= commitIndex {
                r.configurations.committed = r.configurations.latest
                r.configurations.committedIndex = r.configurations.latestIndex
                if !hasVote(r.configurations.committed, r.localID) {
                    stepDown = true
                }
            }

            start := time.Now()
            var groupReady []*list.Element
            var groupFutures = make(map[uint64]*logFuture)
            var lastIdxInGroup uint64

            // Pull all inflight logs that are committed off the queue.
            for e := r.leaderState.inflight.Front(); e != nil; e = e.Next() {
                commitLog := e.Value.(*logFuture)
                idx := commitLog.log.Index
                if idx > commitIndex {
                    // Don't go past the committed index
                    break
                }

                // Measure the commit time
                metrics.MeasureSince([]string{"raft", "commitTime"},
commitLog.dispatch)
                groupReady = append(groupReady, e)
                groupFutures[idx] = commitLog
                lastIdxInGroup = idx
            }

            // Process the group
            if len(groupReady) != 0 {
                r.processLogs(lastIdxInGroup, groupFutures)

                for _, e := range groupReady {
                    r.leaderState.inflight.Remove(e)
                }
            }
```

```
            // Measure the time to enqueue batch of logs for FSM to apply
            metrics.MeasureSince([]string{"raft", "fsm", "enqueue"}, start)

            // Count the number of logs enqueued
            metrics.SetGauge([]string{"raft", "commitNumLogs"},
float32(len(groupReady)))

            if stepDown {
                if r.conf.ShutdownOnRemove {
                    r.logger.Info("removed ourself, shutting down")
                    r.Shutdown()
                } else {
                    r.logger.Info("removed ourself, transitioning to follower")
                    r.setState(Follower)
                }
            }

        case v := <-r.verifyCh:
            if v.quorumSize == 0 {
                // Just dispatched, start the verification
                r.verifyLeader(v)

            } else if v.votes < v.quorumSize {
                // Early return, means there must be a new leader
                r.logger.Warn("new leader elected, stepping down")
                r.setState(Follower)
                delete(r.leaderState.notify, v)
                for _, repl := range r.leaderState.replState {
                    repl.cleanNotify(v)
                }
                v.respond(ErrNotLeader)

            } else {
                // Quorum of members agree, we are still leader
                delete(r.leaderState.notify, v)
                for _, repl := range r.leaderState.replState {
                    repl.cleanNotify(v)
                }
                v.respond(nil)
            }

        case future := <-r.userRestoreCh:
            if r.getLeadershipTransferInProgress() {
                r.logger.Debug(ErrLeadershipTransferInProgress.Error())
                future.respond(ErrLeadershipTransferInProgress)
                continue
            }
            err := r.restoreUserSnapshot(future.meta, future.reader)
            future.respond(err)

        case future := <-r.configurationsCh:
            if r.getLeadershipTransferInProgress() {
                r.logger.Debug(ErrLeadershipTransferInProgress.Error())
```

```
                future.respond(ErrLeadershipTransferInProgress)
                continue
            }
            future.configurations = r.configurations.Clone()
            future.respond(nil)

        case future := <-r.configurationChangeChIfStable():
            if r.getLeadershipTransferInProgress() {
                r.logger.Debug(ErrLeadershipTransferInProgress.Error())
                future.respond(ErrLeadershipTransferInProgress)
                continue
            }
            r.appendConfigurationEntry(future)

        case b := <-r.bootstrapCh:
            b.respond(ErrCantBootstrap)

        case newLog := <-r.applyCh:
            if r.getLeadershipTransferInProgress() {
                r.logger.Debug(ErrLeadershipTransferInProgress.Error())
                newLog.respond(ErrLeadershipTransferInProgress)
                continue
            }
            // Group commit, gather all the ready commits
            ready := []*logFuture{newLog}
        GROUP_COMMIT_LOOP:
            for i := 0; i < r.conf.MaxAppendEntries; i++ {
                select {
                case newLog := <-r.applyCh:
                    ready = append(ready, newLog)
                default:
                    break GROUP_COMMIT_LOOP
                }
            }

            // Dispatch the logs
            if stepDown {
                // we're in the process of stepping down as leader, don't
process anything new
                for i := range ready {
                    ready[i].respond(ErrNotLeader)
                }
            } else {
                r.dispatchLogs(ready)
            }

        case <-lease:
            // Check if we've exceeded the lease, potentially stepping down
            maxDiff := r.checkLeaderLease()

            // Next check interval should adjust for the last node we've
            // contacted, without going negative
            checkInterval := r.conf.LeaderLeaseTimeout - maxDiff
```

```
            if checkInterval < minCheckInterval {
                checkInterval = minCheckInterval
            }

            // Renew the lease timer
            lease = time.After(checkInterval)

        case <-r.shutdownCh:
            return
        }
    }
}

// verifyLeader must be called from the main thread for safety.
// Causes the followers to attempt an immediate heartbeat.
func (r *Raft) verifyLeader(v *verifyFuture) {
    // Current leader always votes for self
    v.votes = 1

    // Set the quorum size, hot-path for single node
    v.quorumSize = r.quorumSize()
    if v.quorumSize == 1 {
        v.respond(nil)
        return
    }

    // Track this request
    v.notifyCh = r.verifyCh
    r.leaderState.notify[v] = struct{}{}

    // Trigger immediate heartbeats
    for _, repl := range r.leaderState.replState {
        repl.notifyLock.Lock()
        repl.notify[v] = struct{}{}
        repl.notifyLock.Unlock()
        asyncNotifyCh(repl.notifyCh)
    }
}

// leadershipTransfer is doing the heavy lifting for the leadership transfer.
func (r *Raft) leadershipTransfer(id ServerID, address ServerAddress, repl
*followerReplication, stopCh chan struct{}, doneCh chan error) {

    // make sure we are not already stopped
    select {
    case <-stopCh:
        doneCh <- nil
        return
    default:
    }

    // Step 1: set this field which stops this leader from responding to any
client requests.
```

```
    r.setLeadershipTransferInProgress(true)
    defer func() { r.setLeadershipTransferInProgress(false) }()

    for atomic.LoadUint64(&repl.nextIndex) <= r.getLastIndex() {
        err := &deferError{}
        err.init()
        repl.triggerDeferErrorCh <- err
        select {
        case err := <-err.errCh:
            if err != nil {
                doneCh <- err
                return
            }
        case <-stopCh:
            doneCh <- nil
            return
        }
    }

    // Step ?: the thesis describes in chap 6.4.1: Using clocks to reduce
    // messaging for read-only queries. If this is implemented, the lease
    // has to be reset as well, in case leadership is transferred. This
    // implementation also has a lease, but it serves another purpose and
    // doesn't need to be reset. The lease mechanism in our raft lib, is
    // setup in a similar way to the one in the thesis, but in practice
    // it's a timer that just tells the leader how often to check
    // heartbeats are still coming in.

    // Step 3: send TimeoutNow message to target server.
    err := r.trans.TimeoutNow(id, address, &TimeoutNowRequest{RPCHeader:
r.getRPCHeader()}, &TimeoutNowResponse{})
    if err != nil {
        err = fmt.Errorf("failed to make TimeoutNow RPC to %v: %v", id, err)
    }
    doneCh <- err
}

// checkLeaderLease is used to check if we can contact a quorum of nodes
// within the last leader lease interval. If not, we need to step down,
// as we may have lost connectivity. Returns the maximum duration without
// contact. This must only be called from the main thread.
func (r *Raft) checkLeaderLease() time.Duration {
    // Track contacted nodes, we can always contact ourself
    contacted := 0

    // Check each follower
    var maxDiff time.Duration
    now := time.Now()
    for _, server := range r.configurations.latest.Servers {
        if server.Suffrage == Voter {
            if server.ID == r.localID {
                contacted++
                continue
```

```
            }
            f := r.leaderState.replState[server.ID]
            diff := now.Sub(f.LastContact())
            if diff <= r.conf.LeaderLeaseTimeout {
                contacted++
                if diff > maxDiff {
                    maxDiff = diff
                }
            } else {
                // Log at least once at high value, then debug. Otherwise it
gets very verbose.
                if diff <= 3*r.conf.LeaderLeaseTimeout {
                    r.logger.Warn("failed to contact", "server-id", server.ID,
"time", diff)
                } else {
                    r.logger.Debug("failed to contact", "server-id", server.ID,
"time", diff)
                }
            }
            metrics.AddSample([]string{"raft", "leader", "lastContact"},
float32(diff/time.Millisecond))
        }
    }

    // Verify we can contact a quorum
    quorum := r.quorumSize()
    if contacted < quorum {
        r.logger.Warn("failed to contact quorum of nodes, stepping down")
        r.setState(Follower)
        metrics.IncrCounter([]string{"raft", "transition",
"leader_lease_timeout"}, 1)
    }
    return maxDiff
}

// quorumSize is used to return the quorum size. This must only be called on
// the main thread.
// TODO: revisit usage
func (r *Raft) quorumSize() int {
    voters := 0
    for _, server := range r.configurations.latest.Servers {
        if server.Suffrage == Voter {
            voters++
        }
    }
    return voters/2 + 1
}

// restoreUserSnapshot is used to manually consume an external snapshot, such
// as if restoring from a backup. We will use the current Raft configuration,
// not the one from the snapshot, so that we can restore into a new cluster. We
// will also use the higher of the index of the snapshot, or the current index,
// and then add 1 to that, so we force a new state with a hole in the Raft log,
```

```
// so that the snapshot will be sent to followers and used for any new joiners.
// This can only be run on the leader, and returns a future that can be used to
// block until complete.
func (r *Raft) restoreUserSnapshot(meta *SnapshotMeta, reader io.Reader) error
{
    defer metrics.MeasureSince([]string{"raft", "restoreUserSnapshot"},
time.Now())

    // Sanity check the version.
    version := meta.Version
    if version < SnapshotVersionMin || version > SnapshotVersionMax {
        return fmt.Errorf("unsupported snapshot version %d", version)
    }

    // We don't support snapshots while there's a config change
    // outstanding since the snapshot doesn't have a means to
    // represent this state.
    committedIndex := r.configurations.committedIndex
    latestIndex := r.configurations.latestIndex
    if committedIndex != latestIndex {
        return fmt.Errorf("cannot restore snapshot now, wait until the
configuration entry at %v has been applied (have applied %v)",
            latestIndex, committedIndex)
    }

    // Cancel any inflight requests.
    for {
        e := r.leaderState.inflight.Front()
        if e == nil {
            break
        }
        e.Value.(*logFuture).respond(ErrAbortedByRestore)
        r.leaderState.inflight.Remove(e)
    }

    // We will overwrite the snapshot metadata with the current term,
    // an index that's greater than the current index, or the last
    // index in the snapshot. It's important that we leave a hole in
    // the index so we know there's nothing in the Raft log there and
    // replication will fault and send the snapshot.
    term := r.getCurrentTerm()
    lastIndex := r.getLastIndex()
    if meta.Index > lastIndex {
        lastIndex = meta.Index
    }
    lastIndex++

    // Dump the snapshot. Note that we use the latest configuration,
    // not the one that came with the snapshot.
    sink, err := r.snapshots.Create(version, lastIndex, term,
        r.configurations.latest, r.configurations.latestIndex, r.trans)
    if err != nil {
        return fmt.Errorf("failed to create snapshot: %v", err)
```

```
    }
    n, err := io.Copy(sink, reader)
    if err != nil {
        sink.Cancel()
        return fmt.Errorf("failed to write snapshot: %v", err)
    }
    if n != meta.Size {
        sink.Cancel()
        return fmt.Errorf("failed to write snapshot, size didn't match (%d !=
%d)", n, meta.Size)
    }
    if err := sink.Close(); err != nil {
        return fmt.Errorf("failed to close snapshot: %v", err)
    }
    r.logger.Info("copied to local snapshot", "bytes", n)

    // Restore the snapshot into the FSM. If this fails we are in a
    // bad state so we panic to take ourselves out.
    fsm := &restoreFuture{ID: sink.ID()}
    fsm.init()
    select {
    case r.fsmMutateCh <- fsm:
    case <-r.shutdownCh:
        return ErrRaftShutdown
    }
    if err := fsm.Error(); err != nil {
        panic(fmt.Errorf("failed to restore snapshot: %v", err))
    }

    // We set the last log so it looks like we've stored the empty
    // index we burned. The last applied is set because we made the
    // FSM take the snapshot state, and we store the last snapshot
    // in the stable store since we created a snapshot as part of
    // this process.
    r.setLastLog(lastIndex, term)
    r.setLastApplied(lastIndex)
    r.setLastSnapshot(lastIndex, term)

    r.logger.Info("restored user snapshot", "index", latestIndex)
    return nil
}

// appendConfigurationEntry changes the configuration and adds a new
// configuration entry to the log. This must only be called from the
// main thread.
func (r *Raft) appendConfigurationEntry(future *configurationChangeFuture) {
    configuration, err := nextConfiguration(r.configurations.latest,
r.configurations.latestIndex, future.req)
    if err != nil {
        future.respond(err)
        return
    }
```

```
    r.logger.Info("updating configuration",
        "command", future.req.command,
        "server-id", future.req.serverID,
        "server-addr", future.req.serverAddress,
        "servers", hclog.Fmt("%+v", configuration.Servers))

    // In pre-ID compatibility mode we translate all configuration changes
    // in to an old remove peer message, which can handle all supported
    // cases for peer changes in the pre-ID world (adding and removing
    // voters). Both add peer and remove peer log entries are handled
    // similarly on old Raft servers, but remove peer does extra checks to
    // see if a leader needs to step down. Since they both assert the full
    // configuration, then we can safely call remove peer for everything.
    if r.protocolVersion < 2 {
        future.log = Log{
            Type: LogRemovePeerDeprecated,
            Data: encodePeers(configuration, r.trans),
        }
    } else {
        future.log = Log{
            Type: LogConfiguration,
            Data: EncodeConfiguration(configuration),
        }
    }

    r.dispatchLogs([]*logFuture{&future.logFuture})
    index := future.Index()
    r.configurations.latest = configuration
    r.configurations.latestIndex = index
    r.leaderState.commitment.setConfiguration(configuration)
    r.startStopReplication()
}

// dispatchLog is called on the leader to push a log to disk, mark it
// as inflight and begin replication of it.
func (r *Raft) dispatchLogs(applyLogs []*logFuture) {
    now := time.Now()
    defer metrics.MeasureSince([]string{"raft", "leader", "dispatchLog"}, now)

    term := r.getCurrentTerm()
    lastIndex := r.getLastIndex()

    n := len(applyLogs)
    logs := make([]*Log, n)
    metrics.SetGauge([]string{"raft", "leader", "dispatchNumLogs"}, float32(n))

    for idx, applyLog := range applyLogs {
        applyLog.dispatch = now
        lastIndex++
        applyLog.log.Index = lastIndex
        applyLog.log.Term = term
        logs[idx] = &applyLog.log
        r.leaderState.inflight.PushBack(applyLog)
```

```
    }

    // Write the log entry locally
    if err := r.logs.StoreLogs(logs); err != nil {
        r.logger.Error("failed to commit logs", "error", err)
        for _, applyLog := range applyLogs {
            applyLog.respond(err)
        }
        r.setState(Follower)
        return
    }
    r.leaderState.commitment.match(r.localID, lastIndex)

    // Update the last log since it's on disk now
    r.setLastLog(lastIndex, term)

    // Notify the replicators of the new log
    for _, f := range r.leaderState.replState {
        asyncNotifyCh(f.triggerCh)
    }
}

// processLogs is used to apply all the committed entries that haven't been
// applied up to the given index limit.
// This can be called from both leaders and followers.
// Followers call this from AppendEntries, for n entries at a time, and always
// pass futures=nil.
// Leaders call this when entries are committed. They pass the futures from any
// inflight logs.
func (r *Raft) processLogs(index uint64, futures map[uint64]*logFuture) {
    // Reject logs we've applied already
    lastApplied := r.getLastApplied()
    if index <= lastApplied {
        r.logger.Warn("skipping application of old log", "index", index)
        return
    }

    applyBatch := func(batch []*commitTuple) {
        select {
        case r.fsmMutateCh <- batch:
        case <-r.shutdownCh:
            for _, cl := range batch {
                if cl.future != nil {
                    cl.future.respond(ErrRaftShutdown)
                }
            }
        }
    }

    batch := make([]*commitTuple, 0, r.conf.MaxAppendEntries)

    // Apply all the preceding logs
    for idx := lastApplied + 1; idx <= index; idx++ {
```

```
        var preparedLog *commitTuple
        // Get the log, either from the future or from our log store
        future, futureOk := futures[idx]
        if futureOk {
            preparedLog = r.prepareLog(&future.log, future)
        } else {
            l := new(Log)
            if err := r.logs.GetLog(idx, l); err != nil {
                r.logger.Error("failed to get log", "index", idx, "error", err)
                panic(err)
            }
            preparedLog = r.prepareLog(l, nil)
        }

        switch {
        case preparedLog != nil:
            // If we have a log ready to send to the FSM add it to the batch.
            // The FSM thread will respond to the future.
            batch = append(batch, preparedLog)

            // If we have filled up a batch, send it to the FSM
            if len(batch) >= r.conf.MaxAppendEntries {
                applyBatch(batch)
                batch = make([]*commitTuple, 0, r.conf.MaxAppendEntries)
            }

        case futureOk:
            // Invoke the future if given.
            future.respond(nil)
        }
    }

    // If there are any remaining logs in the batch apply them
    if len(batch) != 0 {
        applyBatch(batch)
    }

    // Update the lastApplied index and term
    r.setLastApplied(index)
}

// processLog is invoked to process the application of a single committed log
entry.
func (r *Raft) prepareLog(l *Log, future *logFuture) *commitTuple {
    switch l.Type {
    case LogBarrier:
        // Barrier is handled by the FSM
        fallthrough

    case LogCommand:
        return &commitTuple{l, future}

    case LogConfiguration:
```

```
            // Only support this with the v2 configuration format
            if r.protocolVersion > 2 {
                return &commitTuple{l, future}
            }
        case LogAddPeerDeprecated:
        case LogRemovePeerDeprecated:
        case LogNoop:
            // Ignore the no-op

        default:
            panic(fmt.Errorf("unrecognized log type: %#v", l))
        }

        return nil
    }

    // processRPC is called to handle an incoming RPC request. This must only be
    // called from the main thread.
    func (r *Raft) processRPC(rpc RPC) {
        if err := r.checkRPCHeader(rpc); err != nil {
            rpc.Respond(nil, err)
            return
        }

        switch cmd := rpc.Command.(type) {
        case *AppendEntriesRequest:
            r.appendEntries(rpc, cmd)
        case *RequestVoteRequest:
            r.requestVote(rpc, cmd)
        case *InstallSnapshotRequest:
            r.installSnapshot(rpc, cmd)
        case *TimeoutNowRequest:
            r.timeoutNow(rpc, cmd)
        default:
            r.logger.Error("got unexpected command",
                "command", hclog.Fmt("%#v", rpc.Command))
            rpc.Respond(nil, fmt.Errorf("unexpected command"))
        }
    }

    // processHeartbeat is a special handler used just for heartbeat requests
    // so that they can be fast-pathed if a transport supports it. This must only
    // be called from the main thread.
    func (r *Raft) processHeartbeat(rpc RPC) {
        defer metrics.MeasureSince([]string{"raft", "rpc", "processHeartbeat"},
    time.Now())

        // Check if we are shutdown, just ignore the RPC
        select {
        case <-r.shutdownCh:
            return
        default:
        }
```

```
    // Ensure we are only handling a heartbeat
    switch cmd := rpc.Command.(type) {
    case *AppendEntriesRequest:
        r.appendEntries(rpc, cmd)
    default:
        r.logger.Error("expected heartbeat, got", "command", hclog.Fmt("%#v",
rpc.Command))
        rpc.Respond(nil, fmt.Errorf("unexpected command"))
    }
}

// appendEntries is invoked when we get an append entries RPC call. This must
// only be called from the main thread.
func (r *Raft) appendEntries(rpc RPC, a *AppendEntriesRequest) {
    defer metrics.MeasureSince([]string{"raft", "rpc", "appendEntries"},
time.Now())
    // Setup a response
    resp := &AppendEntriesResponse{
        RPCHeader:      r.getRPCHeader(),
        Term:           r.getCurrentTerm(),
        LastLog:        r.getLastIndex(),
        Success:        false,
        NoRetryBackoff: false,
    }
    var rpcErr error
    defer func() {
        rpc.Respond(resp, rpcErr)
    }()

    // Ignore an older term
    if a.Term < r.getCurrentTerm() {
        return
    }

    // Increase the term if we see a newer one, also transition to follower
    // if we ever get an appendEntries call
    if a.Term > r.getCurrentTerm() || r.getState() != Follower {
        // Ensure transition to follower
        r.setState(Follower)
        r.setCurrentTerm(a.Term)
        resp.Term = a.Term
    }

    // Save the current leader
    r.setLeader(ServerAddress(r.trans.DecodePeer(a.Leader)))

    // Verify the last log entry
    if a.PrevLogEntry > 0 {
        lastIdx, lastTerm := r.getLastEntry()

        var prevLogTerm uint64
        if a.PrevLogEntry == lastIdx {
```

```
            prevLogTerm = lastTerm

        } else {
            var prevLog Log
            if err := r.logs.GetLog(a.PrevLogEntry, &prevLog); err != nil {
                r.logger.Warn("failed to get previous log",
                    "previous-index", a.PrevLogEntry,
                    "last-index", lastIdx,
                    "error", err)
                resp.NoRetryBackoff = true
                return
            }
            prevLogTerm = prevLog.Term
        }

        if a.PrevLogTerm != prevLogTerm {
            r.logger.Warn("previous log term mis-match",
                "ours", prevLogTerm,
                "remote", a.PrevLogTerm)
            resp.NoRetryBackoff = true
            return
        }
    }

    // Process any new entries
    if len(a.Entries) > 0 {
        start := time.Now()

        // Delete any conflicting entries, skip any duplicates
        lastLogIdx, _ := r.getLastLog()
        var newEntries []*Log
        for i, entry := range a.Entries {
            if entry.Index > lastLogIdx {
                newEntries = a.Entries[i:]
                break
            }
            var storeEntry Log
            if err := r.logs.GetLog(entry.Index, &storeEntry); err != nil {
                r.logger.Warn("failed to get log entry",
                    "index", entry.Index,
                    "error", err)
                return
            }
            if entry.Term != storeEntry.Term {
                r.logger.Warn("clearing log suffix",
                    "from", entry.Index,
                    "to", lastLogIdx)
                if err := r.logs.DeleteRange(entry.Index, lastLogIdx); err !=
nil {
                    r.logger.Error("failed to clear log suffix", "error", err)
                    return
                }
                if entry.Index <= r.configurations.latestIndex {
```

```
                    r.configurations.latest = r.configurations.committed
                    r.configurations.latestIndex =
r.configurations.committedIndex
                }
                newEntries = a.Entries[i:]
                break
            }
        }

        if n := len(newEntries); n > 0 {
            // Append the new entries
            if err := r.logs.StoreLogs(newEntries); err != nil {
                r.logger.Error("failed to append to logs", "error", err)
                // TODO: leaving r.getLastLog() in the wrong
                // state if there was a truncation above
                return
            }

            // Handle any new configuration changes
            for _, newEntry := range newEntries {
                r.processConfigurationLogEntry(newEntry)
            }

            // Update the lastLog
            last := newEntries[n-1]
            r.setLastLog(last.Index, last.Term)
        }

        metrics.MeasureSince([]string{"raft", "rpc", "appendEntries",
"storeLogs"}, start)
    }

    // Update the commit index
    if a.LeaderCommitIndex > 0 && a.LeaderCommitIndex > r.getCommitIndex() {
        start := time.Now()
        idx := min(a.LeaderCommitIndex, r.getLastIndex())
        r.setCommitIndex(idx)
        if r.configurations.latestIndex <= idx {
            r.configurations.committed = r.configurations.latest
            r.configurations.committedIndex = r.configurations.latestIndex
        }
        r.processLogs(idx, nil)
        metrics.MeasureSince([]string{"raft", "rpc", "appendEntries",
"processLogs"}, start)
    }

    // Everything went well, set success
    resp.Success = true
    r.setLastContact()
    return
}

// processConfigurationLogEntry takes a log entry and updates the latest
```

```
// configuration if the entry results in a new configuration. This must only be
// called from the main thread, or from NewRaft() before any threads have
begun.
func (r *Raft) processConfigurationLogEntry(entry *Log) {
    if entry.Type == LogConfiguration {
        r.configurations.committed = r.configurations.latest
        r.configurations.committedIndex = r.configurations.latestIndex
        r.configurations.latest = DecodeConfiguration(entry.Data)
        r.configurations.latestIndex = entry.Index
    } else if entry.Type == LogAddPeerDeprecated || entry.Type ==
LogRemovePeerDeprecated {
        r.configurations.committed = r.configurations.latest
        r.configurations.committedIndex = r.configurations.latestIndex
        r.configurations.latest = decodePeers(entry.Data, r.trans)
        r.configurations.latestIndex = entry.Index
    }
}

// requestVote is invoked when we get an request vote RPC call.
func (r *Raft) requestVote(rpc RPC, req *RequestVoteRequest) {
    defer metrics.MeasureSince([]string{"raft", "rpc", "requestVote"},
time.Now())
    r.observe(*req)

    // Setup a response
    resp := &RequestVoteResponse{
        RPCHeader: r.getRPCHeader(),
        Term:      r.getCurrentTerm(),
        Granted:   false,
    }
    var rpcErr error
    defer func() {
        rpc.Respond(resp, rpcErr)
    }()

    // Version 0 servers will panic unless the peers is present. It's only
    // used on them to produce a warning message.
    if r.protocolVersion < 2 {
        resp.Peers = encodePeers(r.configurations.latest, r.trans)
    }

    // Check if we have an existing leader [who's not the candidate] and also
    // check the LeadershipTransfer flag is set. Usually votes are rejected if
    // there is a known leader. But if the leader initiated a leadership
transfer,
    // vote!
    candidate := r.trans.DecodePeer(req.Candidate)
    if leader := r.Leader(); leader != "" && leader != candidate &&
!req.LeadershipTransfer {
        r.logger.Warn("rejecting vote request since we have a leader",
            "from", candidate,
            "leader", leader)
        return
```

```
    }

    // Ignore an older term
    if req.Term < r.getCurrentTerm() {
        return
    }

    // Increase the term if we see a newer one
    if req.Term > r.getCurrentTerm() {
        // Ensure transition to follower
        r.logger.Debug("lost leadership because received a requestVote with a
newer term")
        r.setState(Follower)
        r.setCurrentTerm(req.Term)
        resp.Term = req.Term
    }

    // Check if we have voted yet
    lastVoteTerm, err := r.stable.GetUint64(keyLastVoteTerm)
    if err != nil && err != ErrKeyNotFound {
        r.logger.Error("failed to get last vote term", "error", err)
        return
    }
    lastVoteCandBytes, err := r.stable.Get(keyLastVoteCand)
    if err != nil && err != ErrKeyNotFound {
        r.logger.Error("failed to get last vote candidate", "error", err)
        return
    }

    // Check if we've voted in this election before
    if lastVoteTerm == req.Term && lastVoteCandBytes != nil {
        r.logger.Info("duplicate requestVote for same term", "term", req.Term)
        if bytes.Compare(lastVoteCandBytes, req.Candidate) == 0 {
            r.logger.Warn("duplicate requestVote from", "candidate",
req.Candidate)
            resp.Granted = true
        }
        return
    }

    // Reject if their term is older
    lastIdx, lastTerm := r.getLastEntry()
    if lastTerm > req.LastLogTerm {
        r.logger.Warn("rejecting vote request since our last term is greater",
            "candidate", candidate,
            "last-term", lastTerm,
            "last-candidate-term", req.LastLogTerm)
        return
    }

    if lastTerm == req.LastLogTerm && lastIdx > req.LastLogIndex {
        r.logger.Warn("rejecting vote request since our last index is greater",
            "candidate", candidate,
```

```
            "last-index", lastIdx,
            "last-candidate-index", req.LastLogIndex)
        return
    }

    // Persist a vote for safety
    if err := r.persistVote(req.Term, req.Candidate); err != nil {
        r.logger.Error("failed to persist vote", "error", err)
        return
    }

    resp.Granted = true
    r.setLastContact()
    return
}

// installSnapshot is invoked when we get a InstallSnapshot RPC call.
// We must be in the follower state for this, since it means we are
// too far behind a leader for log replay. This must only be called
// from the main thread.
func (r *Raft) installSnapshot(rpc RPC, req *InstallSnapshotRequest) {
    defer metrics.MeasureSince([]string{"raft", "rpc", "installSnapshot"},
time.Now())
    // Setup a response
    resp := &InstallSnapshotResponse{
        Term:    r.getCurrentTerm(),
        Success: false,
    }
    var rpcErr error
    defer func() {
        io.Copy(ioutil.Discard, rpc.Reader) // ensure we always consume all the
snapshot data from the stream [see issue #212]
        rpc.Respond(resp, rpcErr)
    }()

    // Sanity check the version
    if req.SnapshotVersion < SnapshotVersionMin ||
        req.SnapshotVersion > SnapshotVersionMax {
        rpcErr = fmt.Errorf("unsupported snapshot version %d",
req.SnapshotVersion)
        return
    }

    // Ignore an older term
    if req.Term < r.getCurrentTerm() {
        r.logger.Info("ignoring installSnapshot request with older term than
current term",
            "request-term", req.Term,
            "current-term", r.getCurrentTerm())
        return
    }

    // Increase the term if we see a newer one
```

```
    if req.Term > r.getCurrentTerm() {
        // Ensure transition to follower
        r.setState(Follower)
        r.setCurrentTerm(req.Term)
        resp.Term = req.Term
    }

    // Save the current leader
    r.setLeader(ServerAddress(r.trans.DecodePeer(req.Leader)))

    // Create a new snapshot
    var reqConfiguration Configuration
    var reqConfigurationIndex uint64
    if req.SnapshotVersion > 0 {
        reqConfiguration = DecodeConfiguration(req.Configuration)
        reqConfigurationIndex = req.ConfigurationIndex
    } else {
        reqConfiguration = decodePeers(req.Peers, r.trans)
        reqConfigurationIndex = req.LastLogIndex
    }
    version := getSnapshotVersion(r.protocolVersion)
    sink, err := r.snapshots.Create(version, req.LastLogIndex, req.LastLogTerm,
        reqConfiguration, reqConfigurationIndex, r.trans)
    if err != nil {
        r.logger.Error("failed to create snapshot to install", "error", err)
        rpcErr = fmt.Errorf("failed to create snapshot: %v", err)
        return
    }

    // Spill the remote snapshot to disk
    n, err := io.Copy(sink, rpc.Reader)
    if err != nil {
        sink.Cancel()
        r.logger.Error("failed to copy snapshot", "error", err)
        rpcErr = err
        return
    }

    // Check that we received it all
    if n != req.Size {
        sink.Cancel()
        r.logger.Error("failed to receive whole snapshot",
            "received", hclog.Fmt("%d / %d", n, req.Size))
        rpcErr = fmt.Errorf("short read")
        return
    }

    // Finalize the snapshot
    if err := sink.Close(); err != nil {
        r.logger.Error("failed to finalize snapshot", "error", err)
        rpcErr = err
        return
    }
```

```
    r.logger.Info("copied to local snapshot", "bytes", n)

    // Restore snapshot
    future := &restoreFuture{ID: sink.ID()}
    future.init()
    select {
    case r.fsmMutateCh <- future:
    case <-r.shutdownCh:
        future.respond(ErrRaftShutdown)
        return
    }

    // Wait for the restore to happen
    if err := future.Error(); err != nil {
        r.logger.Error("failed to restore snapshot", "error", err)
        rpcErr = err
        return
    }

    // Update the lastApplied so we don't replay old logs
    r.setLastApplied(req.LastLogIndex)

    // Update the last stable snapshot info
    r.setLastSnapshot(req.LastLogIndex, req.LastLogTerm)

    // Restore the peer set
    r.configurations.latest = reqConfiguration
    r.configurations.latestIndex = reqConfigurationIndex
    r.configurations.committed = reqConfiguration
    r.configurations.committedIndex = reqConfigurationIndex

    // Compact logs, continue even if this fails
    if err := r.compactLogs(req.LastLogIndex); err != nil {
        r.logger.Error("failed to compact logs", "error", err)
    }

    r.logger.Info("Installed remote snapshot")
    resp.Success = true
    r.setLastContact()
    return
}

// setLastContact is used to set the last contact time to now
func (r *Raft) setLastContact() {
    r.lastContactLock.Lock()
    r.lastContact = time.Now()
    r.lastContactLock.Unlock()
}

type voteResult struct {
    RequestVoteResponse
    voterID ServerID
}
```

```
// electSelf is used to send a RequestVote RPC to all peers, and vote for
// ourself. This has the side affecting of incrementing the current term. The
// response channel returned is used to wait for all the responses (including a
// vote for ourself). This must only be called from the main thread.
func (r *Raft) electSelf() <-chan *voteResult {
    // Create a response channel
    respCh := make(chan *voteResult, len(r.configurations.latest.Servers))

    // Increment the term
    r.setCurrentTerm(r.getCurrentTerm() + 1)

    // Construct the request
    lastIdx, lastTerm := r.getLastEntry()
    req := &RequestVoteRequest{
        RPCHeader:          r.getRPCHeader(),
        Term:               r.getCurrentTerm(),
        Candidate:          r.trans.EncodePeer(r.localID, r.localAddr),
        LastLogIndex:       lastIdx,
        LastLogTerm:        lastTerm,
        LeadershipTransfer: r.candidateFromLeadershipTransfer,
    }

    // Construct a function to ask for a vote
    askPeer := func(peer Server) {
        r.goFunc(func() {
            defer metrics.MeasureSince([]string{"raft", "candidate",
"electSelf"}, time.Now())
            resp := &voteResult{voterID: peer.ID}
            err := r.trans.RequestVote(peer.ID, peer.Address, req,
&resp.RequestVoteResponse)
            if err != nil {
                r.logger.Error("failed to make requestVote RPC",
                    "target", peer,
                    "error", err)
                resp.Term = req.Term
                resp.Granted = false
            }
            respCh <- resp
        })
    }

    // For each peer, request a vote
    for _, server := range r.configurations.latest.Servers {
        if server.Suffrage == Voter {
            if server.ID == r.localID {
                // Persist a vote for ourselves
                if err := r.persistVote(req.Term, req.Candidate); err != nil {
                    r.logger.Error("failed to persist vote", "error", err)
                    return nil
                }
                // Include our own vote
                respCh <- &voteResult{
```

```
                    RequestVoteResponse: RequestVoteResponse{
                        RPCHeader: r.getRPCHeader(),
                        Term:      req.Term,
                        Granted:   true,
                    },
                    voterID: r.localID,
                }
            } else {
                askPeer(server)
            }
        }
    }

    return respCh
}

// persistVote is used to persist our vote for safety.
func (r *Raft) persistVote(term uint64, candidate []byte) error {
    if err := r.stable.SetUint64(keyLastVoteTerm, term); err != nil {
        return err
    }
    if err := r.stable.Set(keyLastVoteCand, candidate); err != nil {
        return err
    }
    return nil
}

// setCurrentTerm is used to set the current term in a durable manner.
func (r *Raft) setCurrentTerm(t uint64) {
    // Persist to disk first
    if err := r.stable.SetUint64(keyCurrentTerm, t); err != nil {
        panic(fmt.Errorf("failed to save current term: %v", err))
    }
    r.raftState.setCurrentTerm(t)
}

// setState is used to update the current state. Any state
// transition causes the known leader to be cleared. This means
// that leader should be set only after updating the state.
func (r *Raft) setState(state RaftState) {
    r.setLeader("")
    oldState := r.raftState.getState()
    r.raftState.setState(state)
    if oldState != state {
        r.observe(state)
    }
}

// LookupServer looks up a server by ServerID.
func (r *Raft) lookupServer(id ServerID) *Server {
    for _, server := range r.configurations.latest.Servers {
        if server.ID != r.localID {
            return &server
```

```
        }
    }
    return nil
}

// pickServer returns the follower that is most up to date. Because it accesses
// leaderstate, it should only be called from the leaderloop.
func (r *Raft) pickServer() *Server {
    var pick *Server
    var current uint64
    for _, server := range r.configurations.latest.Servers {
        if server.ID == r.localID {
            continue
        }
        state, ok := r.leaderState.replState[server.ID]
        if !ok {
            continue
        }
        nextIdx := atomic.LoadUint64(&state.nextIndex)
        if nextIdx > current {
            current = nextIdx
            tmp := server
            pick = &tmp
        }
    }
    return pick
}

// initiateLeadershipTransfer starts the leadership on the leader side, by
// sending a message to the leadershipTransferCh, to make sure it runs in the
// mainloop.
func (r *Raft) initiateLeadershipTransfer(id *ServerID, address *ServerAddress)
LeadershipTransferFuture {
    future := &leadershipTransferFuture{ID: id, Address: address}
    future.init()

    if id != nil && *id == r.localID {
        err := fmt.Errorf("cannot transfer leadership to itself")
        r.logger.Info(err.Error())
        future.respond(err)
        return future
    }

    select {
    case r.leadershipTransferCh <- future:
        return future
    case <-r.shutdownCh:
        return errorFuture{ErrRaftShutdown}
    default:
        return errorFuture{ErrEnqueueTimeout}
    }
}
```

```
// timeoutNow is what happens when a server receives a TimeoutNowRequest.
func (r *Raft) timeoutNow(rpc RPC, req *TimeoutNowRequest) {
    r.setLeader("")
    r.setState(Candidate)
    r.candidateFromLeadershipTransfer = true
    rpc.Respond(&TimeoutNowResponse{}, nil)
}
```

../raft/raft_test.go

```
package raft

import (
    "bytes"
    "fmt"
    "io/ioutil"
    "os"
    "path/filepath"
    "reflect"
    "strings"
    "sync"
    "sync/atomic"
    "testing"
    "time"
)

func TestRaft_StartStop(t *testing.T) {
    c := MakeCluster(1, t, nil)
    c.Close()
}

func TestRaft_AfterShutdown(t *testing.T) {
    c := MakeCluster(1, t, nil)
    c.Close()
    raft := c.rafts[0]

    // Everything should fail now
    if f := raft.Apply(nil, 0); f.Error() != ErrRaftShutdown {
        c.FailNowf("should be shutdown: %v", f.Error())
    }

    // TODO (slackpad) - Barrier, VerifyLeader, and GetConfiguration can get
    // stuck if the buffered channel consumes the future but things are shut
    // down so they never get processed.
    if f := raft.AddVoter(ServerID("id"), ServerAddress("addr"), 0, 0);
f.Error() != ErrRaftShutdown {
        c.FailNowf("should be shutdown: %v", f.Error())
    }
    if f := raft.AddNonvoter(ServerID("id"), ServerAddress("addr"), 0, 0);
f.Error() != ErrRaftShutdown {
        c.FailNowf("should be shutdown: %v", f.Error())
    }
    if f := raft.RemoveServer(ServerID("id"), 0, 0); f.Error() !=
ErrRaftShutdown {
        c.FailNowf("should be shutdown: %v", f.Error())
    }
    if f := raft.DemoteVoter(ServerID("id"), 0, 0); f.Error() !=
ErrRaftShutdown {
        c.FailNowf("should be shutdown: %v", f.Error())
    }
    if f := raft.Snapshot(); f.Error() != ErrRaftShutdown {
        c.FailNowf("should be shutdown: %v", f.Error())
```

```
    }

    // Should be idempotent
    if f := raft.Shutdown(); f.Error() != nil {
        c.FailNowf("shutdown should be idempotent")
    }

}

func TestRaft_LiveBootstrap(t *testing.T) {
    // Make the cluster.
    c := MakeClusterNoBootstrap(3, t, nil)
    defer c.Close()

    // Build the configuration.
    configuration := Configuration{}
    for _, r := range c.rafts {
        server := Server{
            ID:      r.localID,
            Address: r.localAddr,
        }
        configuration.Servers = append(configuration.Servers, server)
    }

    // Bootstrap one of the nodes live.
    boot := c.rafts[0].BootstrapCluster(configuration)
    if err := boot.Error(); err != nil {
        c.FailNowf("bootstrap err: %v", err)
    }

    // Should be one leader.
    c.Followers()
    leader := c.Leader()
    c.EnsureLeader(t, leader.localAddr)

    // Should be able to apply.
    future := leader.Apply([]byte("test"), c.conf.CommitTimeout)
    if err := future.Error(); err != nil {
        c.FailNowf("apply err: %v", err)
    }
    c.WaitForReplication(1)

    // Make sure the live bootstrap fails now that things are started up.
    boot = c.rafts[0].BootstrapCluster(configuration)
    if err := boot.Error(); err != ErrCantBootstrap {
        c.FailNowf("bootstrap should have failed: %v", err)
    }
}

func TestRaft_RecoverCluster_NoState(t *testing.T) {
    c := MakeClusterNoBootstrap(1, t, nil)
    defer c.Close()
```

```
    r := c.rafts[0]
    configuration := Configuration{
        Servers: []Server{
            {
                ID:      r.localID,
                Address: r.localAddr,
            },
        },
    }
    err := RecoverCluster(&r.conf, &MockFSM{}, r.logs, r.stable,
        r.snapshots, r.trans, configuration)
    if err == nil || !strings.Contains(err.Error(), "no initial state") {
        c.FailNowf("should have failed for no initial state: %v", err)
    }
}

func TestRaft_RecoverCluster(t *testing.T) {
    // Run with different number of applies which will cover no snapshot and
    // snapshot + log scenarios. By sweeping through the trailing logs value
    // we will also hit the case where we have a snapshot only.
    var err error
    runRecover := func(applies int) {
        conf := inmemConfig(t)
        conf.TrailingLogs = 10
        c := MakeCluster(3, t, conf)
        defer c.Close()

        // Perform some commits.
        c.logger.Debug("running with", "applies", applies)
        leader := c.Leader()
        for i := 0; i < applies; i++ {
            future := leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
            if err = future.Error(); err != nil {
                c.FailNowf("[ERR] apply err: %v", err)
            }
        }

        // Snap the configuration.
        future := leader.GetConfiguration()
        if err = future.Error(); err != nil {
            c.FailNowf("[ERR] get configuration err: %v", err)
        }
        configuration := future.Configuration()

        // Shut down the cluster.
        for _, sec := range c.rafts {
            if err = sec.Shutdown().Error(); err != nil {
                c.FailNowf("[ERR] shutdown err: %v", err)
            }
        }

        // Recover the cluster. We need to replace the transport and we
        // replace the FSM so no state can carry over.
```

```
        for i, r := range c.rafts {
            var before []*SnapshotMeta
            before, err = r.snapshots.List()
            if err != nil {
                c.FailNowf("snapshot list err: %v", err)
            }
            if err = RecoverCluster(&r.conf, &MockFSM{}, r.logs, r.stable,
                r.snapshots, r.trans, configuration); err != nil {
                c.FailNowf("recover err: %v", err)
            }

            // Make sure the recovery looks right.
            var after []*SnapshotMeta
            after, err = r.snapshots.List()
            if err != nil {
                c.FailNowf("snapshot list err: %v", err)
            }
            if len(after) != len(before)+1 {
                c.FailNowf("expected a new snapshot, %d vs. %d", len(before),
len(after))
            }
            var first uint64
            first, err = r.logs.FirstIndex()
            if err != nil {
                c.FailNowf("first log index err: %v", err)
            }
            var last uint64
            last, err = r.logs.LastIndex()
            if err != nil {
                c.FailNowf("last log index err: %v", err)
            }
            if first != 0 || last != 0 {
                c.FailNowf("expected empty logs, got %d/%d", first, last)
            }

            // Fire up the recovered Raft instance. We have to patch
            // up the cluster state manually since this is an unusual
            // operation.
            _, trans := NewInmemTransport(r.localAddr)
            var r2 *Raft
            r2, err = NewRaft(&r.conf, &MockFSM{}, r.logs, r.stable,
r.snapshots, trans)
            if err != nil {
                c.FailNowf("new raft err: %v", err)
            }
            c.rafts[i] = r2
            c.trans[i] = r2.trans.(*InmemTransport)
            c.fsms[i] = r2.fsm.(*MockFSM)
        }
        c.FullyConnect()
        time.Sleep(c.propagateTimeout)

        // Let things settle and make sure we recovered.
```

```
        c.EnsureLeader(t, c.Leader().localAddr)
        c.EnsureSame(t)
        c.EnsureSamePeers(t)
    }
    for applies := 0; applies < 20; applies++ {
        runRecover(applies)
    }
}

func TestRaft_HasExistingState(t *testing.T) {
    var err error
    // Make a cluster.
    c := MakeCluster(2, t, nil)
    defer c.Close()

    // Make a new cluster of 1.
    c1 := MakeClusterNoBootstrap(1, t, nil)

    // Make sure the initial state is clean.
    var hasState bool
    hasState, err = HasExistingState(c1.rafts[0].logs, c1.rafts[0].stable,
c1.rafts[0].snapshots)
    if err != nil || hasState {
        c.FailNowf("should not have any existing state, %v", err)
    }

    // Merge clusters.
    c.Merge(c1)
    c.FullyConnect()

    // Join the new node in.
    future := c.Leader().AddVoter(c1.rafts[0].localID, c1.rafts[0].localAddr,
0, 0)
    if err = future.Error(); err != nil {
        c.FailNowf("[ERR] err: %v", err)
    }

    // Check the FSMs.
    c.EnsureSame(t)

    // Check the peers.
    c.EnsureSamePeers(t)

    // Ensure one leader.
    c.EnsureLeader(t, c.Leader().localAddr)

    // Make sure it's not clean.
    hasState, err = HasExistingState(c1.rafts[0].logs, c1.rafts[0].stable,
c1.rafts[0].snapshots)
    if err != nil || !hasState {
        c.FailNowf("should have some existing state, %v", err)
    }
}
```

```go
func TestRaft_SingleNode(t *testing.T) {
    conf := inmemConfig(t)
    c := MakeCluster(1, t, conf)
    defer c.Close()
    raft := c.rafts[0]

    // Watch leaderCh for change
    select {
    case v := <-raft.LeaderCh():
        if !v {
            c.FailNowf("should become leader")
        }
    case <-time.After(conf.HeartbeatTimeout * 3):
        c.FailNowf("timeout becoming leader")
    }

    // Should be leader
    if s := raft.State(); s != Leader {
        c.FailNowf("expected leader: %v", s)
    }

    // Should be able to apply
    future := raft.Apply([]byte("test"), c.conf.HeartbeatTimeout)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Check the response
    if future.Response().(int) != 1 {
        c.FailNowf("bad response: %v", future.Response())
    }

    // Check the index
    if idx := future.Index(); idx == 0 {
        c.FailNowf("bad index: %d", idx)
    }

    // Check that it is applied to the FSM
    if len(getMockFSM(c.fsms[0]).logs) != 1 {
        c.FailNowf("did not apply to FSM!")
    }
}

func TestRaft_TripleNode(t *testing.T) {
    // Make the cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Should be one leader
    c.Followers()
    leader := c.Leader()
    c.EnsureLeader(t, leader.localAddr)
```

```
    // Should be able to apply
    future := leader.Apply([]byte("test"), c.conf.CommitTimeout)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
    c.WaitForReplication(1)
}

func TestRaft_LeaderFail(t *testing.T) {
    // Make the cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Should be one leader
    c.Followers()
    leader := c.Leader()

    // Should be able to apply
    future := leader.Apply([]byte("test"), c.conf.CommitTimeout)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
    c.WaitForReplication(1)

    // Disconnect the leader now
    t.Logf("[INFO] Disconnecting %v", leader)
    leaderTerm := leader.getCurrentTerm()
    c.Disconnect(leader.localAddr)

    // Wait for new leader
    limit := time.Now().Add(c.longstopTimeout)
    var newLead *Raft
    for time.Now().Before(limit) && newLead == nil {
        c.WaitEvent(nil, c.conf.CommitTimeout)
        leaders := c.GetInState(Leader)
        if len(leaders) == 1 && leaders[0] != leader {
            newLead = leaders[0]
        }
    }
    if newLead == nil {
        c.FailNowf("expected new leader")
    }

    // Ensure the term is greater
    if newLead.getCurrentTerm() <= leaderTerm {
        c.FailNowf("expected newer term! %d %d (%v, %v)",
newLead.getCurrentTerm(), leaderTerm, newLead, leader)
    }

    // Apply should work not work on old leader
    future1 := leader.Apply([]byte("fail"), c.conf.CommitTimeout)
```

```
    // Apply should work on newer leader
    future2 := newLead.Apply([]byte("apply"), c.conf.CommitTimeout)

    // Future2 should work
    if err := future2.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Reconnect the networks
    t.Logf("[INFO] Reconnecting %v", leader)
    c.FullyConnect()

    // Future1 should fail
    if err := future1.Error(); err != ErrLeadershipLost && err != ErrNotLeader
{
        c.FailNowf("err: %v", err)
    }

    // Wait for log replication
    c.EnsureSame(t)

    // Check two entries are applied to the FSM
    for _, fsmRaw := range c.fsms {
        fsm := getMockFSM(fsmRaw)
        fsm.Lock()
        if len(fsm.logs) != 2 {
            c.FailNowf("did not apply both to FSM! %v", fsm.logs)
        }
        if bytes.Compare(fsm.logs[0], []byte("test")) != 0 {
            c.FailNowf("first entry should be 'test'")
        }
        if bytes.Compare(fsm.logs[1], []byte("apply")) != 0 {
            c.FailNowf("second entry should be 'apply'")
        }
        fsm.Unlock()
    }
}

func TestRaft_BehindFollower(t *testing.T) {
    // Make the cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Disconnect one follower
    leader := c.Leader()
    followers := c.Followers()
    behind := followers[0]
    c.Disconnect(behind.localAddr)

    // Commit a lot of things
    var future Future
    for i := 0; i < 100; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
```

```
    }

    // Wait for the last future to apply
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    } else {
        t.Logf("[INFO] Finished apply without behind follower")
    }

    // Check that we have a non zero last contact
    if behind.LastContact().IsZero() {
        c.FailNowf("expected previous contact")
    }

    // Reconnect the behind node
    c.FullyConnect()

    // Ensure all the logs are the same
    c.EnsureSame(t)

    // Ensure one leader
    leader = c.Leader()
    c.EnsureLeader(t, leader.localAddr)
}

func TestRaft_ApplyNonLeader(t *testing.T) {
    // Make the cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Wait for a leader
    c.Leader()

    // Try to apply to them
    followers := c.GetInState(Follower)
    if len(followers) != 2 {
        c.FailNowf("Expected 2 followers")
    }
    follower := followers[0]

    // Try to apply
    future := follower.Apply([]byte("test"), c.conf.CommitTimeout)
    if future.Error() != ErrNotLeader {
        c.FailNowf("should not apply on follower")
    }

    // Should be cached
    if future.Error() != ErrNotLeader {
        c.FailNowf("should not apply on follower")
    }
}

func TestRaft_ApplyConcurrent(t *testing.T) {
```

```
    // Make the cluster
    conf := inmemConfig(t)
    conf.HeartbeatTimeout = 2 * conf.HeartbeatTimeout
    conf.ElectionTimeout = 2 * conf.ElectionTimeout
    c := MakeCluster(3, t, conf)
    defer c.Close()

    // Wait for a leader
    leader := c.Leader()

    // Create a wait group
    const sz = 100
    var group sync.WaitGroup
    group.Add(sz)

    applyF := func(i int) {
        defer group.Done()
        future := leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
        if err := future.Error(); err != nil {
            c.Failf("[ERR] err: %v", err)
        }
    }

    // Concurrently apply
    for i := 0; i < sz; i++ {
        go applyF(i)
    }

    // Wait to finish
    doneCh := make(chan struct{})
    go func() {
        group.Wait()
        close(doneCh)
    }()
    select {
    case <-doneCh:
    case <-time.After(c.longstopTimeout):
        c.FailNowf("timeout")
    }

    // If anything failed up to this point then bail now, rather than do a
    // confusing compare.
    if t.Failed() {
        c.FailNowf("One or more of the apply operations failed")
    }

    // Check the FSMs
    c.EnsureSame(t)
}

func TestRaft_ApplyConcurrent_Timeout(t *testing.T) {
    // Make the cluster
    conf := inmemConfig(t)
```

```
    conf.CommitTimeout = 1 * time.Millisecond
    conf.HeartbeatTimeout = 2 * conf.HeartbeatTimeout
    conf.ElectionTimeout = 2 * conf.ElectionTimeout
    c := MakeCluster(1, t, conf)
    defer c.Close()

    // Wait for a leader
    leader := c.Leader()

    // Enough enqueues should cause at least one timeout...
    var didTimeout int32
    for i := 0; (i < 5000) && (atomic.LoadInt32(&didTimeout) == 0); i++ {
        go func(i int) {
            future := leader.Apply([]byte(fmt.Sprintf("test%d", i)),
time.Microsecond)
            if future.Error() == ErrEnqueueTimeout {
                atomic.StoreInt32(&didTimeout, 1)
            }
        }(i)

        // Give the leader loop some other things to do in order to
        // increase the odds of a timeout.
        if i%5 == 0 {
            leader.VerifyLeader()
        }
    }

    // Loop until we see a timeout, or give up.
    limit := time.Now().Add(c.longstopTimeout)
    for time.Now().Before(limit) {
        if atomic.LoadInt32(&didTimeout) != 0 {
            return
        }
        c.WaitEvent(nil, c.propagateTimeout)
    }
    c.FailNowf("Timeout waiting to detect apply timeouts")
}

func TestRaft_JoinNode(t *testing.T) {
    // Make a cluster
    c := MakeCluster(2, t, nil)
    defer c.Close()

    // Make a new cluster of 1
    c1 := MakeClusterNoBootstrap(1, t, nil)

    // Merge clusters
    c.Merge(c1)
    c.FullyConnect()

    // Join the new node in
    future := c.Leader().AddVoter(c1.rafts[0].localID, c1.rafts[0].localAddr,
0, 0)
```

```
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Ensure one leader
    c.EnsureLeader(t, c.Leader().localAddr)

    // Check the FSMs
    c.EnsureSame(t)

    // Check the peers
    c.EnsureSamePeers(t)
}

func TestRaft_JoinNode_ConfigStore(t *testing.T) {
    // Make a cluster
    conf := inmemConfig(t)
    c := makeCluster(t, &MakeClusterOpts{
        Peers:          1,
        Bootstrap:      true,
        Conf:           conf,
        ConfigStoreFSM: true,
    })
    defer c.Close()

    // Make a new nodes
    c1 := makeCluster(t, &MakeClusterOpts{
        Peers:          1,
        Bootstrap:      false,
        Conf:           conf,
        ConfigStoreFSM: true,
    })
    c2 := makeCluster(t, &MakeClusterOpts{
        Peers:          1,
        Bootstrap:      false,
        Conf:           conf,
        ConfigStoreFSM: true,
    })

    // Merge clusters
    c.Merge(c1)
    c.Merge(c2)
    c.FullyConnect()

    // Join the new node in
    future := c.Leader().AddVoter(c1.rafts[0].localID, c1.rafts[0].localAddr,
0, 0)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
    // Join the new node in
    future = c.Leader().AddVoter(c2.rafts[0].localID, c2.rafts[0].localAddr, 0,
0)
```

```
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Ensure one leader
    c.EnsureLeader(t, c.Leader().localAddr)

    // Check the FSMs
    c.EnsureSame(t)

    // Check the peers
    c.EnsureSamePeers(t)

    // Check the fsm holds the correct config logs
    for _, fsmRaw := range c.fsms {
        fsm := getMockFSM(fsmRaw)
        if len(fsm.configurations) != 3 {
            c.FailNowf("unexpected number of configuration changes: %d",
len(fsm.configurations))
        }
        if len(fsm.configurations[0].Servers) != 1 {
            c.FailNowf("unexpected number of servers in config change: %v",
fsm.configurations[0].Servers)
        }
        if len(fsm.configurations[1].Servers) != 2 {
            c.FailNowf("unexpected number of servers in config change: %v",
fsm.configurations[1].Servers)
        }
        if len(fsm.configurations[2].Servers) != 3 {
            c.FailNowf("unexpected number of servers in config change: %v",
fsm.configurations[2].Servers)
        }
    }

}

func TestRaft_RemoveFollower(t *testing.T) {
    // Make a cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Get the leader
    leader := c.Leader()

    // Wait until we have 2 followers
    limit := time.Now().Add(c.longstopTimeout)
    var followers []*Raft
    for time.Now().Before(limit) && len(followers) != 2 {
        c.WaitEvent(nil, c.conf.CommitTimeout)
        followers = c.GetInState(Follower)
    }
    if len(followers) != 2 {
        c.FailNowf("expected two followers: %v", followers)
```

```
    }

    // Remove a follower
    follower := followers[0]
    future := leader.RemoveServer(follower.localID, 0, 0)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Wait a while
    time.Sleep(c.propagateTimeout)

    // Other nodes should have fewer peers
    if configuration := c.getConfiguration(leader); len(configuration.Servers)
!= 2 {
        c.FailNowf("too many peers")
    }
    if configuration := c.getConfiguration(followers[1]);
len(configuration.Servers) != 2 {
        c.FailNowf("too many peers")
    }
}

func TestRaft_RemoveLeader(t *testing.T) {
    // Make a cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Get the leader
    leader := c.Leader()

    // Wait until we have 2 followers
    limit := time.Now().Add(c.longstopTimeout)
    var followers []*Raft
    for time.Now().Before(limit) && len(followers) != 2 {
        c.WaitEvent(nil, c.conf.CommitTimeout)
        followers = c.GetInState(Follower)
    }
    if len(followers) != 2 {
        c.FailNowf("expected two followers: %v", followers)
    }

    // Remove the leader
    f := leader.RemoveServer(leader.localID, 0, 0)

    // Wait for the future to complete
    if f.Error() != nil {
        c.FailNowf("RemoveServer() returned error %v", f.Error())
    }

    // Wait a bit for log application
    time.Sleep(c.propagateTimeout)
```

```
    // Should have a new leader
    time.Sleep(c.propagateTimeout)
    newLeader := c.Leader()
    if newLeader == leader {
        c.FailNowf("removed leader is still leader")
    }

    // Other nodes should have fewer peers
    if configuration := c.getConfiguration(newLeader);
len(configuration.Servers) != 2 {
        c.FailNowf("wrong number of peers %d", len(configuration.Servers))
    }

    // Old leader should be shutdown
    if leader.State() != Shutdown {
        c.FailNowf("old leader should be shutdown")
    }
}

func TestRaft_RemoveLeader_NoShutdown(t *testing.T) {
    // Make a cluster
    conf := inmemConfig(t)
    conf.ShutdownOnRemove = false
    c := MakeCluster(3, t, conf)
    defer c.Close()

    // Get the leader
    c.Followers()
    leader := c.Leader()

    // Remove the leader
    for i := byte(0); i < 100; i++ {
        if i == 80 {
            removeFuture := leader.RemoveServer(leader.localID, 0, 0)
            if err := removeFuture.Error(); err != nil {
                c.FailNowf("err: %v, remove leader failed", err)
            }
        }
        future := leader.Apply([]byte{i}, 0)
        if i > 80 {
            if err := future.Error(); err == nil || err != ErrNotLeader {
                c.FailNowf("err: %v, future entries should fail", err)
            }
        }
    }

    // Wait a while
    time.Sleep(c.propagateTimeout)

    // Should have a new leader
    newLeader := c.Leader()

    // Wait a bit for log application
```

```
    time.Sleep(c.propagateTimeout)

    // Other nodes should have pulled the leader.
    configuration := c.getConfiguration(newLeader)
    if len(configuration.Servers) != 2 {
        c.FailNowf("too many peers")
    }
    if hasVote(configuration, leader.localID) {
        c.FailNowf("old leader should no longer have a vote")
    }

    // Old leader should be a follower.
    if leader.State() != Follower {
        c.FailNowf("leader should be follower")
    }

    // Old leader should not include itself in its peers.
    configuration = c.getConfiguration(leader)
    if len(configuration.Servers) != 2 {
        c.FailNowf("too many peers")
    }
    if hasVote(configuration, leader.localID) {
        c.FailNowf("old leader should no longer have a vote")
    }

    // Other nodes should have the same state
    c.EnsureSame(t)
}

func TestRaft_RemoveFollower_SplitCluster(t *testing.T) {
    // Make a cluster.
    conf := inmemConfig(t)
    c := MakeCluster(4, t, conf)
    defer c.Close()

    // Wait for a leader to get elected.
    leader := c.Leader()

    // Wait to make sure knowledge of the 4th server is known to all the
    // peers.
    numServers := 0
    limit := time.Now().Add(c.longstopTimeout)
    for time.Now().Before(limit) && numServers != 4 {
        time.Sleep(c.propagateTimeout)
        configuration := c.getConfiguration(leader)
        numServers = len(configuration.Servers)
    }
    if numServers != 4 {
        c.FailNowf("Leader should have 4 servers, got %d", numServers)
    }
    c.EnsureSamePeers(t)

    // Isolate two of the followers.
```

```
    followers := c.Followers()
    if len(followers) != 3 {
        c.FailNowf("Expected 3 followers, got %d", len(followers))
    }
    c.Partition([]ServerAddress{followers[0].localAddr,
followers[1].localAddr})

    // Try to remove the remaining follower that was left with the leader.
    future := leader.RemoveServer(followers[2].localID, 0, 0)
    if err := future.Error(); err == nil {
        c.FailNowf("Should not have been able to make peer change")
    }
}

func TestRaft_AddKnownPeer(t *testing.T) {
    // Make a cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Get the leader
    leader := c.Leader()
    followers := c.GetInState(Follower)

    configReq := &configurationsFuture{}
    configReq.init()
    leader.configurationsCh <- configReq
    if err := configReq.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
    startingConfig := configReq.configurations.committed
    startingConfigIdx := configReq.configurations.committedIndex

    // Add a follower
    future := leader.AddVoter(followers[0].localID, followers[0].localAddr, 0,
0)
    if err := future.Error(); err != nil {
        c.FailNowf("AddVoter() err: %v", err)
    }
    configReq = &configurationsFuture{}
    configReq.init()
    leader.configurationsCh <- configReq
    if err := configReq.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
    newConfig := configReq.configurations.committed
    newConfigIdx := configReq.configurations.committedIndex
    if newConfigIdx <= startingConfigIdx {
        c.FailNowf("AddVoter should have written a new config entry, but
configurations.commitedIndex still %d", newConfigIdx)
    }
    if !reflect.DeepEqual(newConfig, startingConfig) {
        c.FailNowf("[ERR} AddVoter with existing peer shouldn't have changed
config, was %#v, but now %#v", startingConfig, newConfig)
```

```
    }
}

func TestRaft_RemoveUnknownPeer(t *testing.T) {
    // Make a cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Get the leader
    leader := c.Leader()
    configReq := &configurationsFuture{}
    configReq.init()
    leader.configurationsCh <- configReq
    if err := configReq.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
    startingConfig := configReq.configurations.committed
    startingConfigIdx := configReq.configurations.committedIndex

    // Remove unknown
    future := leader.RemoveServer(ServerID(NewInmemAddr()), 0, 0)

    // nothing to do, should be a new config entry that's the same as before
    if err := future.Error(); err != nil {
        c.FailNowf("RemoveServer() err: %v", err)
    }
    configReq = &configurationsFuture{}
    configReq.init()
    leader.configurationsCh <- configReq
    if err := configReq.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
    newConfig := configReq.configurations.committed
    newConfigIdx := configReq.configurations.committedIndex
    if newConfigIdx <= startingConfigIdx {
        c.FailNowf("RemoveServer should have written a new config entry, but
configurations.commitedIndex still %d", newConfigIdx)
    }
    if !reflect.DeepEqual(newConfig, startingConfig) {
        c.FailNowf("[ERR} RemoveServer with unknown peer shouldn't of changed
config, was %#v, but now %#v", startingConfig, newConfig)
    }
}

func TestRaft_SnapshotRestore(t *testing.T) {
    // Make the cluster
    conf := inmemConfig(t)
    conf.TrailingLogs = 10
    c := MakeCluster(1, t, conf)
    defer c.Close()

    // Commit a lot of things
    leader := c.Leader()
```

```
    var future Future
    for i := 0; i < 100; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for the last future to apply
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Take a snapshot
    snapFuture := leader.Snapshot()
    if err := snapFuture.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Check for snapshot
    snaps, _ := leader.snapshots.List()
    if len(snaps) != 1 {
        c.FailNowf("should have a snapshot")
    }
    snap := snaps[0]

    // Logs should be trimmed
    if idx, _ := leader.logs.FirstIndex(); idx != snap.Index-
conf.TrailingLogs+1 {
        c.FailNowf("should trim logs to %d: but is %d", snap.Index-
conf.TrailingLogs+1, idx)
    }

    // Shutdown
    shutdown := leader.Shutdown()
    if err := shutdown.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Restart the Raft
    r := leader
    // Can't just reuse the old transport as it will be closed
    _, trans2 := NewInmemTransport(r.trans.LocalAddr())
    r, err := NewRaft(&r.conf, r.fsm, r.logs, r.stable, r.snapshots, trans2)
    if err != nil {
        c.FailNowf("err: %v", err)
    }
    c.rafts[0] = r

    // We should have restored from the snapshot!
    if last := r.getLastApplied(); last != snap.Index {
        c.FailNowf("bad last index: %d, expecting %d", last, snap.Index)
    }
}

// TODO: Need a test that has a previous format Snapshot and check that it can
```

```
// be read/installed on the new code.

// TODO: Need a test to process old-style entries in the Raft log when starting
// up.

func TestRaft_NoRestoreOnStart(t *testing.T) {
    conf := inmemConfig(t)
    conf.TrailingLogs = 10
    conf.NoSnapshotRestoreOnStart = true
    c := MakeCluster(1, t, conf)

    // Commit a lot of things.
    leader := c.Leader()
    var future Future
    for i := 0; i < 100; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for the last future to apply
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Take a snapshot.
    snapFuture := leader.Snapshot()
    if err := snapFuture.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Shutdown.
    shutdown := leader.Shutdown()
    if err := shutdown.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    _, trans := NewInmemTransport(leader.localAddr)
    newFSM := &MockFSM{}
    _, err := NewRaft(&leader.conf, newFSM, leader.logs, leader.stable,
leader.snapshots, trans)
    if err != nil {
        c.FailNowf("err: %v", err)
    }

    if len(newFSM.logs) != 0 {
        c.FailNowf("expected empty FSM, got %v", newFSM)
    }
}

func TestRaft_SnapshotRestore_PeerChange(t *testing.T) {
    var err error
    // Make the cluster.
    conf := inmemConfig(t)
    conf.ProtocolVersion = 1
```

```
    conf.TrailingLogs = 10
    c := MakeCluster(3, t, conf)
    defer c.Close()

    // Commit a lot of things.
    leader := c.Leader()
    var future Future
    for i := 0; i < 100; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for the last future to apply
    if err = future.Error(); err != nil {
        c.FailNowf("[ERR] err: %v", err)
    }

    // Take a snapshot.
    snapFuture := leader.Snapshot()
    if err = snapFuture.Error(); err != nil {
        c.FailNowf("[ERR] err: %v", err)
    }

    // Shutdown.
    shutdown := leader.Shutdown()
    if err = shutdown.Error(); err != nil {
        c.FailNowf("[ERR] err: %v", err)
    }

    // Make a separate cluster.
    c2 := MakeClusterNoBootstrap(2, t, conf)
    defer c2.Close()

    // Kill the old cluster.
    for _, sec := range c.rafts {
        if sec != leader {
            if err = sec.Shutdown().Error(); err != nil {
                c.FailNowf("[ERR] shutdown err: %v", err)
            }
        }
    }

    // Restart the Raft with new peers.
    r := leader

    // Gather the new peer address list.
    var peers []string
    peers = append(peers, fmt.Sprintf("%q", leader.trans.LocalAddr()))
    for _, sec := range c2.rafts {
        peers = append(peers, fmt.Sprintf("%q", sec.trans.LocalAddr()))
    }
    content := []byte(fmt.Sprintf("[%s]", strings.Join(peers, ",")))

    // Perform a manual recovery on the cluster.
```

```
    base, err := ioutil.TempDir("", "")
    if err != nil {
        c.FailNowf("err: %v", err)
    }
    defer os.RemoveAll(base)
    peersFile := filepath.Join(base, "peers.json")
    if err = ioutil.WriteFile(peersFile, content, 0666); err != nil {
        c.FailNowf("[ERR] err: %v", err)
    }
    configuration, err := ReadPeersJSON(peersFile)
    if err != nil {
        c.FailNowf("err: %v", err)
    }
    if err = RecoverCluster(&r.conf, &MockFSM{}, r.logs, r.stable,
        r.snapshots, r.trans, configuration); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Can't just reuse the old transport as it will be closed. We also start
    // with a fresh FSM for good measure so no state can carry over.
    _, trans := NewInmemTransport(r.localAddr)
    r, err = NewRaft(&r.conf, &MockFSM{}, r.logs, r.stable, r.snapshots, trans)
    if err != nil {
        c.FailNowf("err: %v", err)
    }
    c.rafts[0] = r
    c2.rafts = append(c2.rafts, r)
    c2.trans = append(c2.trans, r.trans.(*InmemTransport))
    c2.fsms = append(c2.fsms, r.fsm.(*MockFSM))
    c2.FullyConnect()

    // Wait a while.
    time.Sleep(c.propagateTimeout)

    // Ensure we elect a leader, and that we replicate to our new followers.
    c2.EnsureSame(t)

    // We should have restored from the snapshot! Note that there's one
    // index bump from the noop the leader tees up when it takes over.
    if last := r.getLastApplied(); last != 103 {
        c.FailNowf("bad last: %v", last)
    }

    // Check the peers.
    c2.EnsureSamePeers(t)
}

func TestRaft_AutoSnapshot(t *testing.T) {
    // Make the cluster
    conf := inmemConfig(t)
    conf.SnapshotInterval = conf.CommitTimeout * 2
    conf.SnapshotThreshold = 50
    conf.TrailingLogs = 10
```

```
    c := MakeCluster(1, t, conf)
    defer c.Close()

    // Commit a lot of things
    leader := c.Leader()
    var future Future
    for i := 0; i < 100; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for the last future to apply
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Wait for a snapshot to happen
    time.Sleep(c.propagateTimeout)

    // Check for snapshot
    if snaps, _ := leader.snapshots.List(); len(snaps) == 0 {
        c.FailNowf("should have a snapshot")
    }
}

func TestRaft_UserSnapshot(t *testing.T) {
    // Make the cluster.
    conf := inmemConfig(t)
    conf.SnapshotThreshold = 50
    conf.TrailingLogs = 10
    c := MakeCluster(1, t, conf)
    defer c.Close()

    // With nothing committed, asking for a snapshot should return an error.
    leader := c.Leader()
    if userSnapshotErrorsOnNoData {
        if err := leader.Snapshot().Error(); err != ErrNothingNewToSnapshot {
            c.FailNowf("Request for Snapshot failed: %v", err)
        }
    }

    // Commit some things.
    var future Future
    for i := 0; i < 10; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test %d", i)), 0)
    }
    if err := future.Error(); err != nil {
        c.FailNowf("Error Apply new log entries: %v", err)
    }

    // Now we should be able to ask for a snapshot without getting an error.
    if err := leader.Snapshot().Error(); err != nil {
        c.FailNowf("Request for Snapshot failed: %v", err)
    }
```

```
    // Check for snapshot
    if snaps, _ := leader.snapshots.List(); len(snaps) == 0 {
        c.FailNowf("should have a snapshot")
    }
}

// snapshotAndRestore does a snapshot and restore sequence and applies the
given
// offset to the snapshot index, so we can try out different situations.
func snapshotAndRestore(t *testing.T, offset uint64) {
    // Make the cluster.
    conf := inmemConfig(t)
    c := MakeCluster(3, t, conf)
    defer c.Close()

    // Wait for things to get stable and commit some things.
    leader := c.Leader()
    var future Future
    for i := 0; i < 10; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test %d", i)), 0)
    }
    if err := future.Error(); err != nil {
        c.FailNowf("Error Apply new log entries: %v", err)
    }

    // Take a snapshot.
    snap := leader.Snapshot()
    if err := snap.Error(); err != nil {
        c.FailNowf("Request for Snapshot failed: %v", err)
    }

    // Commit some more things.
    for i := 10; i < 20; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test %d", i)), 0)
    }
    if err := future.Error(); err != nil {
        c.FailNowf("Error Apply new log entries: %v", err)
    }

    // Get the last index before the restore.
    preIndex := leader.getLastIndex()

    // Restore the snapshot, twiddling the index with the offset.
    meta, reader, err := snap.Open()
    meta.Index += offset
    if err != nil {
        c.FailNowf("Snapshot open failed: %v", err)
    }
    defer reader.Close()
    if err := leader.Restore(meta, reader, 5*time.Second); err != nil {
        c.FailNowf("Restore failed: %v", err)
    }
```

```
    // Make sure the index was updated correctly. We add 2 because we burn
    // an index to create a hole, and then we apply a no-op after the
    // restore.
    var expected uint64
    if meta.Index < preIndex {
        expected = preIndex + 2
    } else {
        expected = meta.Index + 2
    }
    lastIndex := leader.getLastIndex()
    if lastIndex != expected {
        c.FailNowf("Index was not updated correctly: %d vs. %d", lastIndex,
expected)
    }

    // Ensure all the logs are the same and that we have everything that was
    // part of the original snapshot, and that the contents after were
    // reverted.
    c.EnsureSame(t)
    fsm := getMockFSM(c.fsms[0])
    fsm.Lock()
    if len(fsm.logs) != 10 {
        c.FailNowf("Log length bad: %d", len(fsm.logs))
    }
    for i, entry := range fsm.logs {
        expected := []byte(fmt.Sprintf("test %d", i))
        if bytes.Compare(entry, expected) != 0 {
            c.FailNowf("Log entry bad: %v", entry)
        }
    }
    fsm.Unlock()

    // Commit some more things.
    for i := 20; i < 30; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test %d", i)), 0)
    }
    if err := future.Error(); err != nil {
        c.FailNowf("Error Apply new log entries: %v", err)
    }
    c.EnsureSame(t)
}

func TestRaft_UserRestore(t *testing.T) {
    // Snapshots from the past.
    snapshotAndRestore(t, 0)
    snapshotAndRestore(t, 1)
    snapshotAndRestore(t, 2)

    // Snapshots from the future.
    snapshotAndRestore(t, 100)
    snapshotAndRestore(t, 1000)
    snapshotAndRestore(t, 10000)
```

```
}

func TestRaft_SendSnapshotFollower(t *testing.T) {
    // Make the cluster
    conf := inmemConfig(t)
    conf.TrailingLogs = 10
    c := MakeCluster(3, t, conf)
    defer c.Close()

    // Disconnect one follower
    followers := c.Followers()
    leader := c.Leader()
    behind := followers[0]
    c.Disconnect(behind.localAddr)

    // Commit a lot of things
    var future Future
    for i := 0; i < 100; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for the last future to apply
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    } else {
        t.Logf("[INFO] Finished apply without behind follower")
    }

    // Snapshot, this will truncate logs!
    for _, r := range c.rafts {
        future = r.Snapshot()
        // the disconnected node will have nothing to snapshot, so that's
expected
        if err := future.Error(); err != nil && err != ErrNothingNewToSnapshot
{
            c.FailNowf("err: %v", err)
        }
    }

    // Reconnect the behind node
    c.FullyConnect()

    // Ensure all the logs are the same
    c.EnsureSame(t)
}

func TestRaft_SendSnapshotAndLogsFollower(t *testing.T) {
    // Make the cluster
    conf := inmemConfig(t)
    conf.TrailingLogs = 10
    c := MakeCluster(3, t, conf)
    defer c.Close()
```

```
    // Disconnect one follower
    followers := c.Followers()
    leader := c.Leader()
    behind := followers[0]
    c.Disconnect(behind.localAddr)

    // Commit a lot of things
    var future Future
    for i := 0; i < 100; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for the last future to apply
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    } else {
        t.Logf("[INFO] Finished apply without behind follower")
    }

    // Snapshot, this will truncate logs!
    for _, r := range c.rafts {
        future = r.Snapshot()
        // the disconnected node will have nothing to snapshot, so that's
expected
        if err := future.Error(); err != nil && err != ErrNothingNewToSnapshot
{
            c.FailNowf("err: %v", err)
        }
    }

    // Commit more logs past the snapshot.
    for i := 100; i < 200; i++ {
        future = leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for the last future to apply
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    } else {
        t.Logf("[INFO] Finished apply without behind follower")
    }

    // Reconnect the behind node
    c.FullyConnect()

    // Ensure all the logs are the same
    c.EnsureSame(t)
}

func TestRaft_ReJoinFollower(t *testing.T) {
    // Enable operation after a remove.
    conf := inmemConfig(t)
    conf.ShutdownOnRemove = false
```

```
    c := MakeCluster(3, t, conf)
    defer c.Close()

    // Get the leader.
    leader := c.Leader()

    // Wait until we have 2 followers.
    limit := time.Now().Add(c.longstopTimeout)
    var followers []*Raft
    for time.Now().Before(limit) && len(followers) != 2 {
        c.WaitEvent(nil, c.conf.CommitTimeout)
        followers = c.GetInState(Follower)
    }
    if len(followers) != 2 {
        c.FailNowf("expected two followers: %v", followers)
    }

    // Remove a follower.
    follower := followers[0]
    future := leader.RemoveServer(follower.localID, 0, 0)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Other nodes should have fewer peers.
    time.Sleep(c.propagateTimeout)
    if configuration := c.getConfiguration(leader); len(configuration.Servers)
!= 2 {
        c.FailNowf("too many peers: %v", configuration)
    }
    if configuration := c.getConfiguration(followers[1]);
len(configuration.Servers) != 2 {
        c.FailNowf("too many peers: %v", configuration)
    }

    // Get the leader. We can't use the normal stability checker here because
    // the removed server will be trying to run an election but will be
    // ignored. The stability check will think this is off nominal because
    // the RequestVote RPCs won't stop firing.
    limit = time.Now().Add(c.longstopTimeout)
    var leaders []*Raft
    for time.Now().Before(limit) && len(leaders) != 1 {
        c.WaitEvent(nil, c.conf.CommitTimeout)
        leaders, _ = c.pollState(Leader)
    }
    if len(leaders) != 1 {
        c.FailNowf("expected a leader")
    }
    leader = leaders[0]

    // Rejoin. The follower will have a higher term than the leader,
    // this will cause the leader to step down, and a new round of elections
    // to take place. We should eventually re-stabilize.
```

```
    future = leader.AddVoter(follower.localID, follower.localAddr, 0, 0)
    if err := future.Error(); err != nil && err != ErrLeadershipLost {
        c.FailNowf("err: %v", err)
    }

    // We should level back up to the proper number of peers. We add a
    // stability check here to make sure the cluster gets to a state where
    // there's a solid leader.
    leader = c.Leader()
    if configuration := c.getConfiguration(leader); len(configuration.Servers)
!= 3 {
        c.FailNowf("missing peers: %v", configuration)
    }
    if configuration := c.getConfiguration(followers[1]);
len(configuration.Servers) != 3 {
        c.FailNowf("missing peers: %v", configuration)
    }

    // Should be a follower now.
    if follower.State() != Follower {
        c.FailNowf("bad state: %v", follower.State())
    }
}

func TestRaft_LeaderLeaseExpire(t *testing.T) {
    // Make a cluster
    conf := inmemConfig(t)
    c := MakeCluster(2, t, conf)
    defer c.Close()

    // Get the leader
    leader := c.Leader()

    // Wait until we have a followers
    limit := time.Now().Add(c.longstopTimeout)
    var followers []*Raft
    for time.Now().Before(limit) && len(followers) != 1 {
        c.WaitEvent(nil, c.conf.CommitTimeout)
        followers = c.GetInState(Follower)
    }
    if len(followers) != 1 {
        c.FailNowf("expected a followers: %v", followers)
    }

    // Disconnect the follower now
    follower := followers[0]
    t.Logf("[INFO] Disconnecting %v", follower)
    c.Disconnect(follower.localAddr)

    // Watch the leaderCh
    select {
    case v := <-leader.LeaderCh():
        if v {
```

```
            c.FailNowf("should step down as leader")
        }
    case <-time.After(conf.LeaderLeaseTimeout * 2):
        c.FailNowf("timeout stepping down as leader")
    }

    // Ensure the last contact of the leader is non-zero
    if leader.LastContact().IsZero() {
        c.FailNowf("expected non-zero contact time")
    }

    // Should be no leaders
    if len(c.GetInState(Leader)) != 0 {
        c.FailNowf("expected step down")
    }

    // Verify no further contact
    last := follower.LastContact()
    time.Sleep(c.propagateTimeout)

    // Check that last contact has not changed
    if last != follower.LastContact() {
        c.FailNowf("unexpected further contact")
    }

    // Ensure both have cleared their leader
    if l := leader.Leader(); l != "" {
        c.FailNowf("bad: %v", l)
    }
    if l := follower.Leader(); l != "" {
        c.FailNowf("bad: %v", l)
    }
}

func TestRaft_Barrier(t *testing.T) {
    // Make the cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Get the leader
    leader := c.Leader()

    // Commit a lot of things
    for i := 0; i < 100; i++ {
        leader.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // Wait for a barrier complete
    barrier := leader.Barrier(0)

    // Wait for the barrier future to apply
    if err := barrier.Error(); err != nil {
        c.FailNowf("err: %v", err)
```

```
    }

    // Ensure all the logs are the same
    c.EnsureSame(t)
    if len(getMockFSM(c.fsms[0]).logs) != 100 {
        c.FailNowf(fmt.Sprintf("Bad log length: %d",
len(getMockFSM(c.fsms[0]).logs)))
    }
}

func TestRaft_VerifyLeader(t *testing.T) {
    // Make the cluster
    c := MakeCluster(3, t, nil)
    defer c.Close()

    // Get the leader
    leader := c.Leader()

    // Verify we are leader
    verify := leader.VerifyLeader()

    // Wait for the verify to apply
    if err := verify.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
}

func TestRaft_VerifyLeader_Single(t *testing.T) {
    // Make the cluster
    c := MakeCluster(1, t, nil)
    defer c.Close()

    // Get the leader
    leader := c.Leader()

    // Verify we are leader
    verify := leader.VerifyLeader()

    // Wait for the verify to apply
    if err := verify.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
}

func TestRaft_VerifyLeader_Fail(t *testing.T) {
    // Make a cluster
    conf := inmemConfig(t)
    c := MakeCluster(2, t, conf)
    defer c.Close()

    // Get the leader
    leader := c.Leader()
```

```
    // Wait until we have a followers
    followers := c.Followers()

    // Force follower to different term
    follower := followers[0]
    follower.setCurrentTerm(follower.getCurrentTerm() + 1)

    // Verify we are leader
    verify := leader.VerifyLeader()

    // Wait for the leader to step down
    if err := verify.Error(); err != ErrNotLeader && err != ErrLeadershipLost {
        c.FailNowf("err: %v", err)
    }

    // Ensure the known leader is cleared
    if l := leader.Leader(); l != "" {
        c.FailNowf("bad: %v", l)
    }
}

func TestRaft_VerifyLeader_PartialConnect(t *testing.T) {
    // Make a cluster
    conf := inmemConfig(t)
    c := MakeCluster(3, t, conf)
    defer c.Close()

    // Get the leader
    leader := c.Leader()

    // Wait until we have a followers
    limit := time.Now().Add(c.longstopTimeout)
    var followers []*Raft
    for time.Now().Before(limit) && len(followers) != 2 {
        c.WaitEvent(nil, c.conf.CommitTimeout)
        followers = c.GetInState(Follower)
    }
    if len(followers) != 2 {
        c.FailNowf("expected two followers but got: %v", followers)
    }

    // Force partial disconnect
    follower := followers[0]
    t.Logf("[INFO] Disconnecting %v", follower)
    c.Disconnect(follower.localAddr)

    // Verify we are leader
    verify := leader.VerifyLeader()

    // Wait for the leader to step down
    if err := verify.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }
```

```
}

func TestRaft_StartAsLeader(t *testing.T) {
    conf := inmemConfig(t)
    conf.StartAsLeader = true
    c := MakeCluster(1, t, conf)
    defer c.Close()
    raft := c.rafts[0]

    // Watch leaderCh for change
    select {
    case v := <-raft.LeaderCh():
        if !v {
            c.FailNowf("should become leader")
        }
    case <-time.After(c.conf.HeartbeatTimeout * 4):
        // Longer than you think as possibility of multiple elections
        c.FailNowf("timeout becoming leader")
    }

    // Should be leader
    if s := raft.State(); s != Leader {
        c.FailNowf("expected leader: %v", s)
    }

    // Should be able to apply
    future := raft.Apply([]byte("test"), c.conf.CommitTimeout)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Check the response
    if future.Response().(int) != 1 {
        c.FailNowf("bad response: %v", future.Response())
    }

    // Check the index
    if idx := future.Index(); idx == 0 {
        c.FailNowf("bad index: %d", idx)
    }

    // Check that it is applied to the FSM
    if len(getMockFSM(c.fsms[0]).logs) != 1 {
        c.FailNowf("did not apply to FSM!")
    }
}

func TestRaft_NotifyCh(t *testing.T) {
    ch := make(chan bool, 1)
    conf := inmemConfig(t)
    conf.NotifyCh = ch
    c := MakeCluster(1, t, conf)
    defer c.Close()
```

```
    // Watch leaderCh for change
    select {
    case v := <-ch:
        if !v {
            c.FailNowf("should become leader")
        }
    case <-time.After(conf.HeartbeatTimeout * 8):
        c.FailNowf("timeout becoming leader")
    }

    // Close the cluster
    c.Close()

    // Watch leaderCh for change
    select {
    case v := <-ch:
        if v {
            c.FailNowf("should step down as leader")
        }
    case <-time.After(conf.HeartbeatTimeout * 6):
        c.FailNowf("timeout on step down as leader")
    }
}

func TestRaft_Voting(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()
    followers := c.Followers()
    ldr := c.Leader()
    ldrT := c.trans[c.IndexOf(ldr)]

    reqVote := RequestVoteRequest{
        RPCHeader:          ldr.getRPCHeader(),
        Term:               ldr.getCurrentTerm() + 10,
        Candidate:          ldrT.EncodePeer(ldr.localID, ldr.localAddr),
        LastLogIndex:       ldr.LastIndex(),
        LastLogTerm:        ldr.getCurrentTerm(),
        LeadershipTransfer: false,
    }
    // a follower that thinks there's a leader should vote for that leader.
    var resp RequestVoteResponse
    if err := ldrT.RequestVote(followers[0].localID, followers[0].localAddr,
&reqVote, &resp); err != nil {
        c.FailNowf("RequestVote RPC failed %v", err)
    }
    if !resp.Granted {
        c.FailNowf("expected vote to be granted, but wasn't %+v", resp)
    }
    // a follower that thinks there's a leader shouldn't vote for a different
candidate
    reqVote.Candidate = ldrT.EncodePeer(followers[0].localID,
followers[0].localAddr)
```

```
    if err := ldrT.RequestVote(followers[1].localID, followers[1].localAddr,
&reqVote, &resp); err != nil {
        c.FailNowf("RequestVote RPC failed %v", err)
    }
    if resp.Granted {
        c.FailNowf("expected vote not to be granted, but was %+v", resp)
    }
    // a follower that thinks there's a leader, but the request has the
leadership transfer flag, should
    // vote for a different candidate
    reqVote.LeadershipTransfer = true
    reqVote.Candidate = ldrT.EncodePeer(followers[0].localID,
followers[0].localAddr)
    if err := ldrT.RequestVote(followers[1].localID, followers[1].localAddr,
&reqVote, &resp); err != nil {
        c.FailNowf("RequestVote RPC failed %v", err)
    }
    if !resp.Granted {
        c.FailNowf("expected vote to be granted, but wasn't %+v", resp)
    }
}

func TestRaft_ProtocolVersion_RejectRPC(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()
    followers := c.Followers()
    ldr := c.Leader()
    ldrT := c.trans[c.IndexOf(ldr)]

    reqVote := RequestVoteRequest{
        RPCHeader: RPCHeader{
            ProtocolVersion: ProtocolVersionMax + 1,
        },
        Term:         ldr.getCurrentTerm() + 10,
        Candidate:    ldrT.EncodePeer(ldr.localID, ldr.localAddr),
        LastLogIndex: ldr.LastIndex(),
        LastLogTerm:  ldr.getCurrentTerm(),
    }

    // Reject a message from a future version we don't understand.
    var resp RequestVoteResponse
    err := ldrT.RequestVote(followers[0].localID, followers[0].localAddr,
&reqVote, &resp)
    if err == nil || !strings.Contains(err.Error(), "protocol version") {
        c.FailNowf("expected RPC to get rejected: %v", err)
    }

    // Reject a message that's too old.
    reqVote.RPCHeader.ProtocolVersion = followers[0].protocolVersion - 2
    err = ldrT.RequestVote(followers[0].localID, followers[0].localAddr,
&reqVote, &resp)
    if err == nil || !strings.Contains(err.Error(), "protocol version") {
        c.FailNowf("expected RPC to get rejected: %v", err)
```

```
    }
}

func TestRaft_ProtocolVersion_Upgrade_1_2(t *testing.T) {
    // Make a cluster back on protocol version 1.
    conf := inmemConfig(t)
    conf.ProtocolVersion = 1
    c := MakeCluster(2, t, conf)
    defer c.Close()

    // Set up another server speaking protocol version 2.
    conf = inmemConfig(t)
    conf.ProtocolVersion = 2
    c1 := MakeClusterNoBootstrap(1, t, conf)

    // Merge clusters.
    c.Merge(c1)
    c.FullyConnect()

    // Make sure the new ID-based operations aren't supported in the old
    // protocol.
    future := c.Leader().AddNonvoter(c1.rafts[0].localID,
c1.rafts[0].localAddr, 0, 1*time.Second)
    if err := future.Error(); err != ErrUnsupportedProtocol {
        c.FailNowf("err: %v", err)
    }
    future = c.Leader().DemoteVoter(c1.rafts[0].localID, 0, 1*time.Second)
    if err := future.Error(); err != ErrUnsupportedProtocol {
        c.FailNowf("err: %v", err)
    }

    // Now do the join using the old address-based API.
    if future := c.Leader().AddPeer(c1.rafts[0].localAddr); future.Error() !=
nil {
        c.FailNowf("err: %v", future.Error())
    }

    // Sanity check the cluster.
    c.EnsureSame(t)
    c.EnsureSamePeers(t)
    c.EnsureLeader(t, c.Leader().localAddr)

    // Now do the remove using the old address-based API.
    if future := c.Leader().RemovePeer(c1.rafts[0].localAddr); future.Error()
!= nil {
        c.FailNowf("err: %v", future.Error())
    }
}

func TestRaft_ProtocolVersion_Upgrade_2_3(t *testing.T) {
    // Make a cluster back on protocol version 2.
    conf := inmemConfig(t)
    conf.ProtocolVersion = 2
```

```
    c := MakeCluster(2, t, conf)
    defer c.Close()
    oldAddr := c.Followers()[0].localAddr

    // Set up another server speaking protocol version 3.
    conf = inmemConfig(t)
    conf.ProtocolVersion = 3
    c1 := MakeClusterNoBootstrap(1, t, conf)

    // Merge clusters.
    c.Merge(c1)
    c.FullyConnect()

    // Use the new ID-based API to add the server with its ID.
    future := c.Leader().AddVoter(c1.rafts[0].localID, c1.rafts[0].localAddr,
0, 1*time.Second)
    if err := future.Error(); err != nil {
        c.FailNowf("err: %v", err)
    }

    // Sanity check the cluster.
    c.EnsureSame(t)
    c.EnsureSamePeers(t)
    c.EnsureLeader(t, c.Leader().localAddr)

    // Remove an old server using the old address-based API.
    if future := c.Leader().RemovePeer(oldAddr); future.Error() != nil {
        c.FailNowf("err: %v", future.Error())
    }
}

func TestRaft_LeadershipTransferInProgress(t *testing.T) {
    r := &Raft{leaderState: leaderState{}}
    r.setupLeaderState()

    if r.getLeadershipTransferInProgress() != false {
        t.Errorf("should be true after setup")
    }

    r.setLeadershipTransferInProgress(true)
    if r.getLeadershipTransferInProgress() != true {
        t.Errorf("should be true because we set it before")
    }
    r.setLeadershipTransferInProgress(false)
    if r.getLeadershipTransferInProgress() != false {
        t.Errorf("should be false because we set it before")
    }
}

func pointerToString(s string) *string {
    return &s
}
```

```
func TestRaft_LeadershipTransferPickServer(t *testing.T) {
    type variant struct {
        lastLogIndex int
        servers      map[string]uint64
        expected     *string
    }
    leaderID := "z"
    variants := []variant{
        {lastLogIndex: 10, servers: map[string]uint64{}, expected: nil},
        {lastLogIndex: 10, servers: map[string]uint64{leaderID: 11, "a": 9},
expected: pointerToString("a")},
        {lastLogIndex: 10, servers: map[string]uint64{leaderID: 11, "a": 9,
"b": 8}, expected: pointerToString("a")},
        {lastLogIndex: 10, servers: map[string]uint64{leaderID: 11, "c": 9,
"b": 8, "a": 8}, expected: pointerToString("c")},
        {lastLogIndex: 10, servers: map[string]uint64{leaderID: 11, "a": 7,
"b": 11, "c": 8}, expected: pointerToString("b")},
    }
    for i, v := range variants {
        servers := []Server{}
        replState := map[ServerID]*followerReplication{}
        for id, idx := range v.servers {
            servers = append(servers, Server{ID: ServerID(id)})
            replState[ServerID(id)] = &followerReplication{nextIndex: idx}
        }
        r := Raft{leaderState: leaderState{}, localID: ServerID(leaderID),
configurations: configurations{latest: Configuration{Servers: servers}}}
        r.lastLogIndex = uint64(v.lastLogIndex)
        r.leaderState.replState = replState

        actual := r.pickServer()
        if v.expected == nil && actual == nil {
            continue
        } else if v.expected == nil && actual != nil {
            t.Errorf("case %d: actual: %v doesn't match expected: %v", i,
actual, v.expected)
        } else if actual == nil && v.expected != nil {
            t.Errorf("case %d: actual: %v doesn't match expected: %v", i,
actual, v.expected)
        } else if string(actual.ID) != *v.expected {
            t.Errorf("case %d: actual: %v doesn't match expected: %v", i,
actual.ID, *v.expected)
        }
    }
}

func TestRaft_LeadershipTransfer(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()

    oldLeader := string(c.Leader().localID)
    err := c.Leader().LeadershipTransfer()
    if err.Error() != nil {
```

```
        t.Fatalf("Didn't expect error: %v", err.Error())
    }
    newLeader := string(c.Leader().localID)
    if oldLeader == newLeader {
        t.Error("Leadership should have been transitioned to another peer.")
    }
}

func TestRaft_LeadershipTransferWithOneNode(t *testing.T) {
    c := MakeCluster(1, t, nil)
    defer c.Close()

    future := c.Leader().LeadershipTransfer()
    if future.Error() == nil {
        t.Fatal("leadership transfer should err")
    }

    expected := "cannot find peer"
    actual := future.Error().Error()
    if !strings.Contains(actual, expected) {
        t.Errorf("leadership transfer should err with: %s", expected)
    }
}

func TestRaft_LeadershipTransferWithSevenNodes(t *testing.T) {
    c := MakeCluster(7, t, nil)
    defer c.Close()

    oldLeader := c.Leader().localID
    follower := c.GetInState(Follower)[0]
    future := c.Leader().LeadershipTransferToServer(follower.localID,
follower.localAddr)
    if future.Error() != nil {
        t.Fatalf("Didn't expect error: %v", future.Error())
    }
    if oldLeader == c.Leader().localID {
        t.Error("Leadership should have been transitioned to specified
server.")
    }
}

func TestRaft_LeadershipTransferToInvalidID(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()

    future := c.Leader().LeadershipTransferToServer(ServerID("abc"),
ServerAddress("127.0.0.1"))
    if future.Error() == nil {
        t.Fatal("leadership transfer should err")
    }

    expected := "cannot find replication state"
    actual := future.Error().Error()
```

```
    if !strings.Contains(actual, expected) {
        t.Errorf("leadership transfer should err with: %s", expected)
    }
}

func TestRaft_LeadershipTransferToInvalidAddress(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()

    follower := c.GetInState(Follower)[0]
    future := c.Leader().LeadershipTransferToServer(follower.localID,
ServerAddress("127.0.0.1"))
    if future.Error() == nil {
        t.Fatal("leadership transfer should err")
    }
    expected := "failed to make TimeoutNow RPC"
    actual := future.Error().Error()
    if !strings.Contains(actual, expected) {
        t.Errorf("leadership transfer should err with: %s", expected)
    }
}

func TestRaft_LeadershipTransferToBehindServer(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()

    l := c.Leader()
    behind := c.GetInState(Follower)[0]

    // Commit a lot of things
    for i := 0; i < 1000; i++ {
        l.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    future := l.LeadershipTransferToServer(behind.localID, behind.localAddr)
    if future.Error() != nil {
        t.Fatalf("This is not supposed to error: %v", future.Error())
    }
    if c.Leader().localID != behind.localID {
        t.Fatal("Behind server did not get leadership")
    }
}

func TestRaft_LeadershipTransferToItself(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()

    l := c.Leader()

    future := l.LeadershipTransferToServer(l.localID, l.localAddr)
    if future.Error() == nil {
        t.Fatal("leadership transfer should err")
    }
```

```
    expected := "cannot transfer leadership to itself"
    actual := future.Error().Error()
    if !strings.Contains(actual, expected) {
        t.Errorf("leadership transfer should err with: %s", expected)
    }
}

func TestRaft_LeadershipTransferLeaderRejectsClientRequests(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()
    l := c.Leader()
    l.setLeadershipTransferInProgress(true)

    // tests for API > protocol version 3 is missing here because leadership
transfer
    // is only available for protocol version >= 3
    // TODO: is something missing here?
    futures := []Future{
        l.AddNonvoter(ServerID(""), ServerAddress(""), 0, 0),
        l.AddVoter(ServerID(""), ServerAddress(""), 0, 0),
        l.Apply([]byte("test"), 0),
        l.Barrier(0),
        l.DemoteVoter(ServerID(""), 0, 0),
        l.GetConfiguration(),

        // the API is tested, but here we are making sure we reject any config
change.
        l.requestConfigChange(configurationChangeRequest{},
100*time.Millisecond),
    }
    futures = append(futures, l.LeadershipTransfer())
    select {
    case <-l.leadershipTransferCh:
    default:
    }

    futures = append(futures, l.LeadershipTransferToServer(ServerID(""),
ServerAddress("")))

    for i, f := range futures {
        if f.Error() != ErrLeadershipTransferInProgress {
            t.Errorf("case %d: should have errored with: %s, instead of %s", i,
ErrLeadershipTransferInProgress, f.Error())
        }
    }
}

func TestRaft_LeadershipTransferLeaderReplicationTimeout(t *testing.T) {
    c := MakeCluster(3, t, nil)
    defer c.Close()

    l := c.Leader()
    behind := c.GetInState(Follower)[0]
```

```
    // Commit a lot of things, so that the timeout can kick in
    for i := 0; i < 10000; i++ {
        l.Apply([]byte(fmt.Sprintf("test%d", i)), 0)
    }

    // set ElectionTimeout really short because this is used to determine
    // how long a transfer is allowed to take.
    l.conf.ElectionTimeout = 1 * time.Nanosecond

    future := l.LeadershipTransferToServer(behind.localID, behind.localAddr)
    if future.Error() == nil {
        t.Log("This test is fishing for a replication timeout, but this is not
guaranteed to happen.")
    } else {
        expected := "leadership transfer timeout"
        actual := future.Error().Error()
        if !strings.Contains(actual, expected) {
            t.Errorf("leadership transfer should err with: %s", expected)
        }
    }
}

func TestRaft_LeadershipTransferStopRightAway(t *testing.T) {
    r := Raft{leaderState: leaderState{}}
    r.setupLeaderState()

    stopCh := make(chan struct{})
    doneCh := make(chan error, 1)
    close(stopCh)
    r.leadershipTransfer(ServerID("a"), ServerAddress(""),
&followerReplication{}, stopCh, doneCh)
    err := <-doneCh
    if err != nil {
        t.Errorf("leadership shouldn't have started, but instead it error with:
%v", err)
    }
}
func TestRaft_GetConfigurationNoBootstrap(t *testing.T) {
    c := MakeCluster(2, t, nil)
    defer c.Close()

    // Should be one leader
    c.Followers()
    leader := c.Leader()
    c.EnsureLeader(t, leader.localAddr)

    // Should be able to apply
    future := leader.Apply([]byte("test"), c.conf.CommitTimeout)
    if err := future.Error(); err != nil {
        c.FailNowf("[ERR] err: %v", err)
    }
    c.WaitForReplication(1)
```

```
    // Get configuration via GetConfiguration of a running node
    cfgf := c.rafts[0].GetConfiguration()
    if err := cfgf.Error(); err != nil {
        t.Fatal(err)
    }
    expected := cfgf.Configuration()

    // Obtain the same configuration via GetConfig
    logs := c.stores[0]
    store := c.stores[0]
    snap := c.snaps[0]
    trans := c.trans[0]
    observed, err := GetConfiguration(c.conf, c.fsms[0], logs, store, snap,
trans)
    if err != nil {
        t.Fatal(err)
    }
    if !reflect.DeepEqual(observed, expected) {
        t.Errorf("GetConfiguration result differ from Raft.GetConfiguration:
observed %+v, expected %+v", observed, expected)
    }
}

// TODO: These are test cases we'd like to write for appendEntries().
// Unfortunately, it's difficult to do so with the current way this file is
// tested.
//
// Term check:
// - m.term is too small: no-op.
// - m.term is too large: update term, become follower, process request.
// - m.term is right but we're candidate: become follower, process request.
//
// Previous entry check:
// - prev is within the snapshot, before the snapshot's index: assume match.
// - prev is within the snapshot, exactly the snapshot's index: check
//   snapshot's term.
// - prev is a log entry: check entry's term.
// - prev is past the end of the log: return fail.
//
// New entries:
// - new entries are all new: add them all.
// - new entries are all duplicate: ignore them all without ever removing dups.
// - new entries some duplicate, some new: add the new ones without ever
//   removing dups.
// - new entries all conflict: remove the conflicting ones, add their
//   replacements.
// - new entries some duplicate, some conflict: remove the conflicting ones,
//   add their replacement, without ever removing dups.
//
// Storage errors handled properly.
// Commit index updated properly.
```

../raft/replication.go

```
package raft

import (
    "errors"
    "fmt"
    "sync"
    "sync/atomic"
    "time"

    "github.com/armon/go-metrics"
)

const (
    maxFailureScale = 12
    failureWait     = 10 * time.Millisecond
)

var (
    // ErrLogNotFound indicates a given log entry is not available.
    ErrLogNotFound = errors.New("log not found")

    // ErrPipelineReplicationNotSupported can be returned by the transport to
    // signal that pipeline replication is not supported in general, and that
    // no error message should be produced.
    ErrPipelineReplicationNotSupported = errors.New("pipeline replication not
supported")
)

// followerReplication is in charge of sending snapshots and log entries from
// this leader during this particular term to a remote follower.
type followerReplication struct {
    // currentTerm and nextIndex must be kept at the top of the struct so
    // they're 64 bit aligned which is a requirement for atomic ops on 32 bit
    // platforms.

    // currentTerm is the term of this leader, to be included in AppendEntries
    // requests.
    currentTerm uint64

    // nextIndex is the index of the next log entry to send to the follower,
    // which may fall past the end of the log.
    nextIndex uint64

    // peer contains the network address and ID of the remote follower.
    peer Server

    // commitment tracks the entries acknowledged by followers so that the
    // leader's commit index can advance. It is updated on successful
    // AppendEntries responses.
    commitment *commitment

    // stopCh is notified/closed when this leader steps down or the follower is
```

```
    // removed from the cluster. In the follower removed case, it carries a log
    // index; replication should be attempted with a best effort up through
that
    // index, before exiting.
    stopCh chan uint64

    // triggerCh is notified every time new entries are appended to the log.
    triggerCh chan struct{}

    // triggerDeferErrorCh is used to provide a backchannel. By sending a
    // deferErr, the sender can be notifed when the replication is done.
    triggerDeferErrorCh chan *deferError

    // lastContact is updated to the current time whenever any response is
    // received from the follower (successful or not). This is used to check
    // whether the leader should step down (Raft.checkLeaderLease()).
    lastContact time.Time
    // lastContactLock protects 'lastContact'.
    lastContactLock sync.RWMutex

    // failures counts the number of failed RPCs since the last success, which
is
    // used to apply backoff.
    failures uint64

    // notifyCh is notified to send out a heartbeat, which is used to check
that
    // this server is still leader.
    notifyCh chan struct{}
    // notify is a map of futures to be resolved upon receipt of an
    // acknowledgement, then cleared from this map.
    notify map[*verifyFuture]struct{}
    // notifyLock protects 'notify'.
    notifyLock sync.Mutex

    // stepDown is used to indicate to the leader that we
    // should step down based on information from a follower.
    stepDown chan struct{}

    // allowPipeline is used to determine when to pipeline the AppendEntries
RPCs.
    // It is private to this replication goroutine.
    allowPipeline bool
}

// notifyAll is used to notify all the waiting verify futures
// if the follower believes we are still the leader.
func (s *followerReplication) notifyAll(leader bool) {
    // Clear the waiting notifies minimizing lock time
    s.notifyLock.Lock()
    n := s.notify
    s.notify = make(map[*verifyFuture]struct{})
    s.notifyLock.Unlock()
```

```
    // Submit our votes
    for v := range n {
        v.vote(leader)
    }
}

// cleanNotify is used to delete notify, .
func (s *followerReplication) cleanNotify(v *verifyFuture) {
    s.notifyLock.Lock()
    delete(s.notify, v)
    s.notifyLock.Unlock()
}

// LastContact returns the time of last contact.
func (s *followerReplication) LastContact() time.Time {
    s.lastContactLock.RLock()
    last := s.lastContact
    s.lastContactLock.RUnlock()
    return last
}

// setLastContact sets the last contact to the current time.
func (s *followerReplication) setLastContact() {
    s.lastContactLock.Lock()
    s.lastContact = time.Now()
    s.lastContactLock.Unlock()
}

// replicate is a long running routine that replicates log entries to a single
// follower.
func (r *Raft) replicate(s *followerReplication) {
    // Start an async heartbeating routing
    stopHeartbeat := make(chan struct{})
    defer close(stopHeartbeat)
    r.goFunc(func() { r.heartbeat(s, stopHeartbeat) })

RPC:
    shouldStop := false
    for !shouldStop {
        select {
        case maxIndex := <-s.stopCh:
            // Make a best effort to replicate up to this index
            if maxIndex > 0 {
                r.replicateTo(s, maxIndex)
            }
            return
        case deferErr := <-s.triggerDeferErrorCh:
            lastLogIdx, _ := r.getLastLog()
            shouldStop = r.replicateTo(s, lastLogIdx)
            if !shouldStop {
                deferErr.respond(nil)
            } else {
```

```
                deferErr.respond(fmt.Errorf("replication failed"))
            }
        case <-s.triggerCh:
            lastLogIdx, _ := r.getLastLog()
            shouldStop = r.replicateTo(s, lastLogIdx)
        // This is _not_ our heartbeat mechanism but is to ensure
        // followers quickly learn the leader's commit index when
        // raft commits stop flowing naturally. The actual heartbeats
        // can't do this to keep them unblocked by disk IO on the
        // follower. See https://github.com/hashicorp/raft/issues/282.
        case <-randomTimeout(r.conf.CommitTimeout):
            lastLogIdx, _ := r.getLastLog()
            shouldStop = r.replicateTo(s, lastLogIdx)
        }

        // If things looks healthy, switch to pipeline mode
        if !shouldStop && s.allowPipeline {
            goto PIPELINE
        }
    }
    return

PIPELINE:
    // Disable until re-enabled
    s.allowPipeline = false

    // Replicates using a pipeline for high performance. This method
    // is not able to gracefully recover from errors, and so we fall back
    // to standard mode on failure.
    if err := r.pipelineReplicate(s); err != nil {
        if err != ErrPipelineReplicationNotSupported {
            r.logger.Error("failed to start pipeline replication to", "peer",
s.peer, "error", err)
        }
    }
    goto RPC
}

// replicateTo is a helper to replicate(), used to replicate the logs up to a
// given last index.
// If the follower log is behind, we take care to bring them up to date.
func (r *Raft) replicateTo(s *followerReplication, lastIndex uint64)
(shouldStop bool) {
    // Create the base request
    var req AppendEntriesRequest
    var resp AppendEntriesResponse
    var start time.Time
START:
    // Prevent an excessive retry rate on errors
    if s.failures > 0 {
        select {
        case <-time.After(backoff(failureWait, s.failures, maxFailureScale)):
        case <-r.shutdownCh:
```

```
        }
    }

    // Setup the request
    if err := r.setupAppendEntries(s, &req, atomic.LoadUint64(&s.nextIndex),
lastIndex); err == ErrLogNotFound {
        goto SEND_SNAP
    } else if err != nil {
        return
    }

    // Make the RPC call
    start = time.Now()
    if err := r.trans.AppendEntries(s.peer.ID, s.peer.Address, &req, &resp);
err != nil {
        r.logger.Error("failed to appendEntries to", "peer", s.peer, "error",
err)
        s.failures++
        return
    }
    appendStats(string(s.peer.ID), start, float32(len(req.Entries)))

    // Check for a newer term, stop running
    if resp.Term > req.Term {
        r.handleStaleTerm(s)
        return true
    }

    // Update the last contact
    s.setLastContact()

    // Update s based on success
    if resp.Success {
        // Update our replication state
        updateLastAppended(s, &req)

        // Clear any failures, allow pipelining
        s.failures = 0
        s.allowPipeline = true
    } else {
        atomic.StoreUint64(&s.nextIndex, max(min(s.nextIndex-1,
resp.LastLog+1), 1))
        if resp.NoRetryBackoff {
            s.failures = 0
        } else {
            s.failures++
        }
        r.logger.Warn("appendEntries rejected, sending older logs", "peer",
s.peer, "next", atomic.LoadUint64(&s.nextIndex))
    }

CHECK_MORE:
    // Poll the stop channel here in case we are looping and have been asked
```

```
    // to stop, or have stepped down as leader. Even for the best effort case
    // where we are asked to replicate to a given index and then shutdown,
    // it's better to not loop in here to send lots of entries to a straggler
    // that's leaving the cluster anyways.
    select {
    case <-s.stopCh:
        return true
    default:
    }

    // Check if there are more logs to replicate
    if atomic.LoadUint64(&s.nextIndex) <= lastIndex {
        goto START
    }
    return

    // SEND_SNAP is used when we fail to get a log, usually because the
follower
    // is too far behind, and we must ship a snapshot down instead
SEND_SNAP:
    if stop, err := r.sendLatestSnapshot(s); stop {
        return true
    } else if err != nil {
        r.logger.Error("failed to send snapshot to", "peer", s.peer, "error",
err)
        return
    }

    // Check if there is more to replicate
    goto CHECK_MORE
}

// sendLatestSnapshot is used to send the latest snapshot we have
// down to our follower.
func (r *Raft) sendLatestSnapshot(s *followerReplication) (bool, error) {
    // Get the snapshots
    snapshots, err := r.snapshots.List()
    if err != nil {
        r.logger.Error("failed to list snapshots", "error", err)
        return false, err
    }

    // Check we have at least a single snapshot
    if len(snapshots) == 0 {
        return false, fmt.Errorf("no snapshots found")
    }

    // Open the most recent snapshot
    snapID := snapshots[0].ID
    meta, snapshot, err := r.snapshots.Open(snapID)
    if err != nil {
        r.logger.Error("failed to open snapshot", "id", snapID, "error", err)
        return false, err
```

```
    }
    defer snapshot.Close()

    // Setup the request
    req := InstallSnapshotRequest{
        RPCHeader:          r.getRPCHeader(),
        SnapshotVersion:    meta.Version,
        Term:               s.currentTerm,
        Leader:             r.trans.EncodePeer(r.localID, r.localAddr),
        LastLogIndex:       meta.Index,
        LastLogTerm:        meta.Term,
        Peers:              meta.Peers,
        Size:               meta.Size,
        Configuration:      EncodeConfiguration(meta.Configuration),
        ConfigurationIndex: meta.ConfigurationIndex,
    }

    // Make the call
    start := time.Now()
    var resp InstallSnapshotResponse
    if err := r.trans.InstallSnapshot(s.peer.ID, s.peer.Address, &req, &resp,
snapshot); err != nil {
        r.logger.Error("failed to install snapshot", "id", snapID, "error",
err)
        s.failures++
        return false, err
    }
    metrics.MeasureSince([]string{"raft", "replication", "installSnapshot",
string(s.peer.ID)}, start)

    // Check for a newer term, stop running
    if resp.Term > req.Term {
        r.handleStaleTerm(s)
        return true, nil
    }

    // Update the last contact
    s.setLastContact()

    // Check for success
    if resp.Success {
        // Update the indexes
        atomic.StoreUint64(&s.nextIndex, meta.Index+1)
        s.commitment.match(s.peer.ID, meta.Index)

        // Clear any failures
        s.failures = 0

        // Notify we are still leader
        s.notifyAll(true)
    } else {
        s.failures++
        r.logger.Warn("installSnapshot rejected to", "peer", s.peer)
```

```
    }
    return false, nil
}

// heartbeat is used to periodically invoke AppendEntries on a peer
// to ensure they don't time out. This is done async of replicate(),
// since that routine could potentially be blocked on disk IO.
func (r *Raft) heartbeat(s *followerReplication, stopCh chan struct{}) {
    var failures uint64
    req := AppendEntriesRequest{
        RPCHeader: r.getRPCHeader(),
        Term:      s.currentTerm,
        Leader:    r.trans.EncodePeer(r.localID, r.localAddr),
    }
    var resp AppendEntriesResponse
    for {
        // Wait for the next heartbeat interval or forced notify
        select {
        case <-s.notifyCh:
        case <-randomTimeout(r.conf.HeartbeatTimeout / 10):
        case <-stopCh:
            return
        }

        start := time.Now()
        if err := r.trans.AppendEntries(s.peer.ID, s.peer.Address, &req,
&resp); err != nil {
            r.logger.Error("failed to heartbeat to", "peer", s.peer.Address,
"error", err)
            failures++
            select {
            case <-time.After(backoff(failureWait, failures, maxFailureScale)):
            case <-stopCh:
            }
        } else {
            s.setLastContact()
            failures = 0
            metrics.MeasureSince([]string{"raft", "replication", "heartbeat",
string(s.peer.ID)}, start)
            s.notifyAll(resp.Success)
        }
    }
}

// pipelineReplicate is used when we have synchronized our state with the
follower,
// and want to switch to a higher performance pipeline mode of replication.
// We only pipeline AppendEntries commands, and if we ever hit an error, we
fall
// back to the standard replication which can handle more complex situations.
func (r *Raft) pipelineReplicate(s *followerReplication) error {
    // Create a new pipeline
    pipeline, err := r.trans.AppendEntriesPipeline(s.peer.ID, s.peer.Address)
```

```
    if err != nil {
        return err
    }
    defer pipeline.Close()

    // Log start and stop of pipeline
    r.logger.Info("pipelining replication", "peer", s.peer)
    defer r.logger.Info("aborting pipeline replication", "peer", s.peer)

    // Create a shutdown and finish channel
    stopCh := make(chan struct{})
    finishCh := make(chan struct{})

    // Start a dedicated decoder
    r.goFunc(func() { r.pipelineDecode(s, pipeline, stopCh, finishCh) })

    // Start pipeline sends at the last good nextIndex
    nextIndex := atomic.LoadUint64(&s.nextIndex)

    shouldStop := false
SEND:
    for !shouldStop {
        select {
        case <-finishCh:
            break SEND
        case maxIndex := <-s.stopCh:
            // Make a best effort to replicate up to this index
            if maxIndex > 0 {
                r.pipelineSend(s, pipeline, &nextIndex, maxIndex)
            }
            break SEND
        case deferErr := <-s.triggerDeferErrorCh:
            lastLogIdx, _ := r.getLastLog()
            shouldStop = r.pipelineSend(s, pipeline, &nextIndex, lastLogIdx)
            if !shouldStop {
                deferErr.respond(nil)
            } else {
                deferErr.respond(fmt.Errorf("replication failed"))
            }
        case <-s.triggerCh:
            lastLogIdx, _ := r.getLastLog()
            shouldStop = r.pipelineSend(s, pipeline, &nextIndex, lastLogIdx)
        case <-randomTimeout(r.conf.CommitTimeout):
            lastLogIdx, _ := r.getLastLog()
            shouldStop = r.pipelineSend(s, pipeline, &nextIndex, lastLogIdx)
        }
    }

    // Stop our decoder, and wait for it to finish
    close(stopCh)
    select {
    case <-finishCh:
    case <-r.shutdownCh:
```

```
    }
    return nil
}

// pipelineSend is used to send data over a pipeline. It is a helper to
// pipelineReplicate.
func (r *Raft) pipelineSend(s *followerReplication, p AppendPipeline, nextIdx
*uint64, lastIndex uint64) (shouldStop bool) {
    // Create a new append request
    req := new(AppendEntriesRequest)
    if err := r.setupAppendEntries(s, req, *nextIdx, lastIndex); err != nil {
        return true
    }

    // Pipeline the append entries
    if _, err := p.AppendEntries(req, new(AppendEntriesResponse)); err != nil {
        r.logger.Error("failed to pipeline appendEntries", "peer", s.peer,
"error", err)
        return true
    }

    // Increase the next send log to avoid re-sending old logs
    if n := len(req.Entries); n > 0 {
        last := req.Entries[n-1]
        atomic.StoreUint64(nextIdx, last.Index+1)
    }
    return false
}

// pipelineDecode is used to decode the responses of pipelined requests.
func (r *Raft) pipelineDecode(s *followerReplication, p AppendPipeline, stopCh,
finishCh chan struct{}) {
    defer close(finishCh)
    respCh := p.Consumer()
    for {
        select {
        case ready := <-respCh:
            req, resp := ready.Request(), ready.Response()
            appendStats(string(s.peer.ID), ready.Start(),
float32(len(req.Entries)))

            // Check for a newer term, stop running
            if resp.Term > req.Term {
                r.handleStaleTerm(s)
                return
            }

            // Update the last contact
            s.setLastContact()

            // Abort pipeline if not successful
            if !resp.Success {
                return
```

```
            }

            // Update our replication state
            updateLastAppended(s, req)
        case <-stopCh:
            return
        }
    }
}

// setupAppendEntries is used to setup an append entries request.
func (r *Raft) setupAppendEntries(s *followerReplication, req
*AppendEntriesRequest, nextIndex, lastIndex uint64) error {
    req.RPCHeader = r.getRPCHeader()
    req.Term = s.currentTerm
    req.Leader = r.trans.EncodePeer(r.localID, r.localAddr)
    req.LeaderCommitIndex = r.getCommitIndex()
    if err := r.setPreviousLog(req, nextIndex); err != nil {
        return err
    }
    if err := r.setNewLogs(req, nextIndex, lastIndex); err != nil {
        return err
    }
    return nil
}

// setPreviousLog is used to setup the PrevLogEntry and PrevLogTerm for an
// AppendEntriesRequest given the next index to replicate.
func (r *Raft) setPreviousLog(req *AppendEntriesRequest, nextIndex uint64)
error {
    // Guard for the first index, since there is no 0 log entry
    // Guard against the previous index being a snapshot as well
    lastSnapIdx, lastSnapTerm := r.getLastSnapshot()
    if nextIndex == 1 {
        req.PrevLogEntry = 0
        req.PrevLogTerm = 0

    } else if (nextIndex - 1) == lastSnapIdx {
        req.PrevLogEntry = lastSnapIdx
        req.PrevLogTerm = lastSnapTerm

    } else {
        var l Log
        if err := r.logs.GetLog(nextIndex-1, &l); err != nil {
            r.logger.Error("failed to get log", "index", nextIndex-1, "error",
err)
            return err
        }

        // Set the previous index and term (0 if nextIndex is 1)
        req.PrevLogEntry = l.Index
        req.PrevLogTerm = l.Term
    }
```

```
    return nil
}

// setNewLogs is used to setup the logs which should be appended for a request.
func (r *Raft) setNewLogs(req *AppendEntriesRequest, nextIndex, lastIndex
uint64) error {
    // Append up to MaxAppendEntries or up to the lastIndex
    req.Entries = make([]*Log, 0, r.conf.MaxAppendEntries)
    maxIndex := min(nextIndex+uint64(r.conf.MaxAppendEntries)-1, lastIndex)
    for i := nextIndex; i <= maxIndex; i++ {
        oldLog := new(Log)
        if err := r.logs.GetLog(i, oldLog); err != nil {
            r.logger.Error("failed to get log", "index", i, "error", err)
            return err
        }
        req.Entries = append(req.Entries, oldLog)
    }
    return nil
}

// appendStats is used to emit stats about an AppendEntries invocation.
func appendStats(peer string, start time.Time, logs float32) {
    metrics.MeasureSince([]string{"raft", "replication", "appendEntries",
"rpc", peer}, start)
    metrics.IncrCounter([]string{"raft", "replication", "appendEntries",
"logs", peer}, logs)
}

// handleStaleTerm is used when a follower indicates that we have a stale term.
func (r *Raft) handleStaleTerm(s *followerReplication) {
    r.logger.Error("peer has newer term, stopping replication", "peer", s.peer)
    s.notifyAll(false) // No longer leader
    asyncNotifyCh(s.stepDown)
}

// updateLastAppended is used to update follower replication state after a
// successful AppendEntries RPC.
// TODO: This isn't used during InstallSnapshot, but the code there is similar.
func updateLastAppended(s *followerReplication, req *AppendEntriesRequest) {
    // Mark any inflight logs as committed
    if logs := req.Entries; len(logs) > 0 {
        last := logs[len(logs)-1]
        atomic.StoreUint64(&s.nextIndex, last.Index+1)
        s.commitment.match(s.peer.ID, last.Index)
    }

    // Notify still leader
    s.notifyAll(true)
}
```

../raft/snapshot.go

```
package raft

import (
    "fmt"
    "io"
    "time"

    "github.com/armon/go-metrics"
)

// SnapshotMeta is for metadata of a snapshot.
type SnapshotMeta struct {
    // Version is the version number of the snapshot metadata. This does not
cover
    // the application's data in the snapshot, that should be versioned
    // separately.
    Version SnapshotVersion

    // ID is opaque to the store, and is used for opening.
    ID string

    // Index and Term store when the snapshot was taken.
    Index uint64
    Term  uint64

    // Peers is deprecated and used to support version 0 snapshots, but will
    // be populated in version 1 snapshots as well to help with upgrades.
    Peers []byte

    // Configuration and ConfigurationIndex are present in version 1
    // snapshots and later.
    Configuration      Configuration
    ConfigurationIndex uint64

    // Size is the size of the snapshot in bytes.
    Size int64
}

// SnapshotStore interface is used to allow for flexible implementations
// of snapshot storage and retrieval. For example, a client could implement
// a shared state store such as S3, allowing new nodes to restore snapshots
// without streaming from the leader.
type SnapshotStore interface {
    // Create is used to begin a snapshot at a given index and term, and with
    // the given committed configuration. The version parameter controls
    // which snapshot version to create.
    Create(version SnapshotVersion, index, term uint64, configuration
Configuration,
        configurationIndex uint64, trans Transport) (SnapshotSink, error)

    // List is used to list the available snapshots in the store.
    // It should return then in descending order, with the highest index first.
```

```
    List() ([]*SnapshotMeta, error)

    // Open takes a snapshot ID and provides a ReadCloser. Once close is
    // called it is assumed the snapshot is no longer needed.
    Open(id string) (*SnapshotMeta, io.ReadCloser, error)
}

// SnapshotSink is returned by StartSnapshot. The FSM will Write state
// to the sink and call Close on completion. On error, Cancel will be invoked.
type SnapshotSink interface {
    io.WriteCloser
    ID() string
    Cancel() error
}

// runSnapshots is a long running goroutine used to manage taking
// new snapshots of the FSM. It runs in parallel to the FSM and
// main goroutines, so that snapshots do not block normal operation.
func (r *Raft) runSnapshots() {
    for {
        select {
        case <-randomTimeout(r.conf.SnapshotInterval):
            // Check if we should snapshot
            if !r.shouldSnapshot() {
                continue
            }

            // Trigger a snapshot
            if _, err := r.takeSnapshot(); err != nil {
                r.logger.Error("failed to take snapshot", "error", err)
            }

        case future := <-r.userSnapshotCh:
            // User-triggered, run immediately
            id, err := r.takeSnapshot()
            if err != nil {
                r.logger.Error("failed to take snapshot", "error", err)
            } else {
                future.opener = func() (*SnapshotMeta, io.ReadCloser, error) {
                    return r.snapshots.Open(id)
                }
            }
            future.respond(err)

        case <-r.shutdownCh:
            return
        }
    }
}

// shouldSnapshot checks if we meet the conditions to take
// a new snapshot.
func (r *Raft) shouldSnapshot() bool {
```

```
    // Check the last snapshot index
    lastSnap, _ := r.getLastSnapshot()

    // Check the last log index
    lastIdx, err := r.logs.LastIndex()
    if err != nil {
        r.logger.Error("failed to get last log index", "error", err)
        return false
    }

    // Compare the delta to the threshold
    delta := lastIdx - lastSnap
    return delta >= r.conf.SnapshotThreshold
}

// takeSnapshot is used to take a new snapshot. This must only be called from
// the snapshot thread, never the main thread. This returns the ID of the new
// snapshot, along with an error.
func (r *Raft) takeSnapshot() (string, error) {
    defer metrics.MeasureSince([]string{"raft", "snapshot", "takeSnapshot"},
time.Now())

    // Create a request for the FSM to perform a snapshot.
    snapReq := &reqSnapshotFuture{}
    snapReq.init()

    // Wait for dispatch or shutdown.
    select {
    case r.fsmSnapshotCh <- snapReq:
    case <-r.shutdownCh:
        return "", ErrRaftShutdown
    }

    // Wait until we get a response
    if err := snapReq.Error(); err != nil {
        if err != ErrNothingNewToSnapshot {
            err = fmt.Errorf("failed to start snapshot: %v", err)
        }
        return "", err
    }
    defer snapReq.snapshot.Release()

    // Make a request for the configurations and extract the committed info.
    // We have to use the future here to safely get this information since
    // it is owned by the main thread.
    configReq := &configurationsFuture{}
    configReq.init()
    select {
    case r.configurationsCh <- configReq:
    case <-r.shutdownCh:
        return "", ErrRaftShutdown
    }
    if err := configReq.Error(); err != nil {
```

```
        return "", err
    }
    committed := configReq.configurations.committed
    committedIndex := configReq.configurations.committedIndex

    // We don't support snapshots while there's a config change outstanding
    // since the snapshot doesn't have a means to represent this state. This
    // is a little weird because we need the FSM to apply an index that's
    // past the configuration change, even though the FSM itself doesn't see
    // the configuration changes. It should be ok in practice with normal
    // application traffic flowing through the FSM. If there's none of that
    // then it's not crucial that we snapshot, since there's not much going
    // on Raft-wise.
    if snapReq.index < committedIndex {
        return "", fmt.Errorf("cannot take snapshot now, wait until the
configuration entry at %v has been applied (have applied %v)",
            committedIndex, snapReq.index)
    }

    // Create a new snapshot.
    r.logger.Info("starting snapshot up to", "index", snapReq.index)
    start := time.Now()
    version := getSnapshotVersion(r.protocolVersion)
    sink, err := r.snapshots.Create(version, snapReq.index, snapReq.term,
committed, committedIndex, r.trans)
    if err != nil {
        return "", fmt.Errorf("failed to create snapshot: %v", err)
    }
    metrics.MeasureSince([]string{"raft", "snapshot", "create"}, start)

    // Try to persist the snapshot.
    start = time.Now()
    if err := snapReq.snapshot.Persist(sink); err != nil {
        sink.Cancel()
        return "", fmt.Errorf("failed to persist snapshot: %v", err)
    }
    metrics.MeasureSince([]string{"raft", "snapshot", "persist"}, start)

    // Close and check for error.
    if err := sink.Close(); err != nil {
        return "", fmt.Errorf("failed to close snapshot: %v", err)
    }

    // Update the last stable snapshot info.
    r.setLastSnapshot(snapReq.index, snapReq.term)

    // Compact the logs.
    if err := r.compactLogs(snapReq.index); err != nil {
        return "", err
    }

    r.logger.Info("snapshot complete up to", "index", snapReq.index)
    return sink.ID(), nil
```

```
}

// compactLogs takes the last inclusive index of a snapshot
// and trims the logs that are no longer needed.
func (r *Raft) compactLogs(snapIdx uint64) error {
    defer metrics.MeasureSince([]string{"raft", "compactLogs"}, time.Now())
    // Determine log ranges to compact
    minLog, err := r.logs.FirstIndex()
    if err != nil {
        return fmt.Errorf("failed to get first log index: %v", err)
    }

    // Check if we have enough logs to truncate
    lastLogIdx, _ := r.getLastLog()
    if lastLogIdx <= r.conf.TrailingLogs {
        return nil
    }

    // Truncate up to the end of the snapshot, or `TrailingLogs`
    // back from the head, which ever is further back. This ensures
    // at least `TrailingLogs` entries, but does not allow logs
    // after the snapshot to be removed.
    maxLog := min(snapIdx, lastLogIdx-r.conf.TrailingLogs)

    if minLog > maxLog {
        r.logger.Info("no logs to truncate")
        return nil
    }

    r.logger.Info("compacting logs", "from", minLog, "to", maxLog)

    // Compact the logs
    if err := r.logs.DeleteRange(minLog, maxLog); err != nil {
        return fmt.Errorf("log compaction failed: %v", err)
    }
    return nil
}
```

../raft/stable.go

```
package raft

// StableStore is used to provide stable storage
// of key configurations to ensure safety.
type StableStore interface {
    Set(key []byte, val []byte) error

    // Get returns the value for key, or an empty byte slice if key was not
found.
    Get(key []byte) ([]byte, error)

    SetUint64(key []byte, val uint64) error

    // GetUint64 returns the uint64 value for key, or 0 if key was not found.
    GetUint64(key []byte) (uint64, error)
}
```

../raft/state.go

```
package raft

import (
    "sync"
    "sync/atomic"
)

// RaftState captures the state of a Raft node: Follower, Candidate, Leader,
// or Shutdown.
type RaftState uint32

const (
    // Follower is the initial state of a Raft node.
    Follower RaftState = iota

    // Candidate is one of the valid states of a Raft node.
    Candidate

    // Leader is one of the valid states of a Raft node.
    Leader

    // Shutdown is the terminal state of a Raft node.
    Shutdown
)

func (s RaftState) String() string {
    switch s {
    case Follower:
        return "Follower"
    case Candidate:
        return "Candidate"
    case Leader:
        return "Leader"
    case Shutdown:
        return "Shutdown"
    default:
        return "Unknown"
    }
}

// raftState is used to maintain various state variables
// and provides an interface to set/get the variables in a
// thread safe manner.
type raftState struct {
    // currentTerm commitIndex, lastApplied,  must be kept at the top of
    // the struct so they're 64 bit aligned which is a requirement for
    // atomic ops on 32 bit platforms.

    // The current term, cache of StableStore
    currentTerm uint64

    // Highest committed log entry
```

```
    commitIndex uint64

    // Last applied log to the FSM
    lastApplied uint64

    // protects 4 next fields
    lastLock sync.Mutex

    // Cache the latest snapshot index/term
    lastSnapshotIndex uint64
    lastSnapshotTerm  uint64

    // Cache the latest log from LogStore
    lastLogIndex uint64
    lastLogTerm  uint64

    // Tracks running goroutines
    routinesGroup sync.WaitGroup

    // The current state
    state RaftState
}

func (r *raftState) getState() RaftState {
    stateAddr := (*uint32)(&r.state)
    return RaftState(atomic.LoadUint32(stateAddr))
}

func (r *raftState) setState(s RaftState) {
    stateAddr := (*uint32)(&r.state)
    atomic.StoreUint32(stateAddr, uint32(s))
}

func (r *raftState) getCurrentTerm() uint64 {
    return atomic.LoadUint64(&r.currentTerm)
}

func (r *raftState) setCurrentTerm(term uint64) {
    atomic.StoreUint64(&r.currentTerm, term)
}

func (r *raftState) getLastLog() (index, term uint64) {
    r.lastLock.Lock()
    index = r.lastLogIndex
    term = r.lastLogTerm
    r.lastLock.Unlock()
    return
}

func (r *raftState) setLastLog(index, term uint64) {
    r.lastLock.Lock()
    r.lastLogIndex = index
    r.lastLogTerm = term
```

```
    r.lastLock.Unlock()
}

func (r *raftState) getLastSnapshot() (index, term uint64) {
    r.lastLock.Lock()
    index = r.lastSnapshotIndex
    term = r.lastSnapshotTerm
    r.lastLock.Unlock()
    return
}

func (r *raftState) setLastSnapshot(index, term uint64) {
    r.lastLock.Lock()
    r.lastSnapshotIndex = index
    r.lastSnapshotTerm = term
    r.lastLock.Unlock()
}

func (r *raftState) getCommitIndex() uint64 {
    return atomic.LoadUint64(&r.commitIndex)
}

func (r *raftState) setCommitIndex(index uint64) {
    atomic.StoreUint64(&r.commitIndex, index)
}

func (r *raftState) getLastApplied() uint64 {
    return atomic.LoadUint64(&r.lastApplied)
}

func (r *raftState) setLastApplied(index uint64) {
    atomic.StoreUint64(&r.lastApplied, index)
}

// Start a goroutine and properly handle the race between a routine
// starting and incrementing, and exiting and decrementing.
func (r *raftState) goFunc(f func()) {
    r.routinesGroup.Add(1)
    go func() {
        defer r.routinesGroup.Done()
        f()
    }()
}

func (r *raftState) waitShutdown() {
    r.routinesGroup.Wait()
}

// getLastIndex returns the last index in stable storage.
// Either from the last log or from the last snapshot.
func (r *raftState) getLastIndex() uint64 {
    r.lastLock.Lock()
    defer r.lastLock.Unlock()
```

```
    return max(r.lastLogIndex, r.lastSnapshotIndex)
}

// getLastEntry returns the last index and term in stable storage.
// Either from the last log or from the last snapshot.
func (r *raftState) getLastEntry() (uint64, uint64) {
    r.lastLock.Lock()
    defer r.lastLock.Unlock()
    if r.lastLogIndex >= r.lastSnapshotIndex {
        return r.lastLogIndex, r.lastLogTerm
    }
    return r.lastSnapshotIndex, r.lastSnapshotTerm
}
```

../raft/tcp_transport.go

```
package raft

import (
    "errors"
    "github.com/hashicorp/go-hclog"
    "io"
    "net"
    "time"
)

var (
    errNotAdvertisable = errors.New("local bind address is not advertisable")
    errNotTCP          = errors.New("local address is not a TCP address")
)

// TCPStreamLayer implements StreamLayer interface for plain TCP.
type TCPStreamLayer struct {
    advertise net.Addr
    listener  *net.TCPListener
}

// NewTCPTransport returns a NetworkTransport that is built on top of
// a TCP streaming transport layer.
func NewTCPTransport(
    bindAddr string,
    advertise net.Addr,
    maxPool int,
    timeout time.Duration,
    logOutput io.Writer,
) (*NetworkTransport, error) {
    return newTCPTransport(bindAddr, advertise, func(stream StreamLayer)
*NetworkTransport {
        return NewNetworkTransport(stream, maxPool, timeout, logOutput)
    })
}

// NewTCPTransportWithLogger returns a NetworkTransport that is built on top of
// a TCP streaming transport layer, with log output going to the supplied
Logger
func NewTCPTransportWithLogger(
    bindAddr string,
    advertise net.Addr,
    maxPool int,
    timeout time.Duration,
    logger hclog.Logger,
) (*NetworkTransport, error) {
    return newTCPTransport(bindAddr, advertise, func(stream StreamLayer)
*NetworkTransport {
        return NewNetworkTransportWithLogger(stream, maxPool, timeout, logger)
    })
}
```

```go
// NewTCPTransportWithConfig returns a NetworkTransport that is built on top of
// a TCP streaming transport layer, using the given config struct.
func NewTCPTransportWithConfig(
    bindAddr string,
    advertise net.Addr,
    config *NetworkTransportConfig,
) (*NetworkTransport, error) {
    return newTCPTransport(bindAddr, advertise, func(stream StreamLayer)
*NetworkTransport {
        config.Stream = stream
        return NewNetworkTransportWithConfig(config)
    })
}

func newTCPTransport(bindAddr string,
    advertise net.Addr,
    transportCreator func(stream StreamLayer) *NetworkTransport)
(*NetworkTransport, error) {
    // Try to bind
    list, err := net.Listen("tcp", bindAddr)
    if err != nil {
        return nil, err
    }

    // Create stream
    stream := &TCPStreamLayer{
        advertise: advertise,
        listener:  list.(*net.TCPListener),
    }

    // Verify that we have a usable advertise address
    addr, ok := stream.Addr().(*net.TCPAddr)
    if !ok {
        list.Close()
        return nil, errNotTCP
    }
    if addr.IP.IsUnspecified() {
        list.Close()
        return nil, errNotAdvertisable
    }

    // Create the network transport
    trans := transportCreator(stream)
    return trans, nil
}

// Dial implements the StreamLayer interface.
func (t *TCPStreamLayer) Dial(address ServerAddress, timeout time.Duration)
(net.Conn, error) {
    return net.DialTimeout("tcp", string(address), timeout)
}

// Accept implements the net.Listener interface.
```

```
func (t *TCPStreamLayer) Accept() (c net.Conn, err error) {
    return t.listener.Accept()
}

// Close implements the net.Listener interface.
func (t *TCPStreamLayer) Close() (err error) {
    return t.listener.Close()
}

// Addr implements the net.Listener interface.
func (t *TCPStreamLayer) Addr() net.Addr {
    // Use an advertise addr if provided
    if t.advertise != nil {
        return t.advertise
    }
    return t.listener.Addr()
}
```

../raft/tcp_transport_test.go

```
package raft

import (
    "net"
    "testing"
)

func TestTCPTransport_BadAddr(t *testing.T) {
    _, err := NewTCPTransportWithLogger("0.0.0.0:0", nil, 1, 0,
newTestLogger(t))
    if err != errNotAdvertisable {
        t.Fatalf("err: %v", err)
    }
}

func TestTCPTransport_WithAdvertise(t *testing.T) {
    addr := &net.TCPAddr{IP: []byte{127, 0, 0, 1}, Port: 12345}
    trans, err := NewTCPTransportWithLogger("0.0.0.0:0", addr, 1, 0,
newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    if trans.LocalAddr() != "127.0.0.1:12345" {
        t.Fatalf("bad: %v", trans.LocalAddr())
    }
}
```

../raft/testing.go

```
package raft

import (
    "bytes"
    "fmt"
    "io"
    "io/ioutil"
    "os"
    "reflect"
    "sync"
    "testing"
    "time"

    "github.com/hashicorp/go-hclog"
    "github.com/hashicorp/go-msgpack/codec"
)

var (
    userSnapshotErrorsOnNoData = true
)

// Return configurations optimized for in-memory
func inmemConfig(t *testing.T) *Config {
    conf := DefaultConfig()
    conf.HeartbeatTimeout = 50 * time.Millisecond
    conf.ElectionTimeout = 50 * time.Millisecond
    conf.LeaderLeaseTimeout = 50 * time.Millisecond
    conf.CommitTimeout = 5 * time.Millisecond
    conf.Logger = newTestLeveledLogger(t)
    return conf
}

// MockFSM is an implementation of the FSM interface, and just stores
// the logs sequentially.
//
// NOTE: This is exposed for middleware testing purposes and is not a stable
API
type MockFSM struct {
    sync.Mutex
    logs           [][]byte
    configurations []Configuration
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
type MockFSMConfigStore struct {
    FSM
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
type WrappingFSM interface {
```

```
    Underlying() FSM
}

func getMockFSM(fsm FSM) *MockFSM {
    switch f := fsm.(type) {
    case *MockFSM:
        return f
    case *MockFSMConfigStore:
        return f.FSM.(*MockFSM)
    case WrappingFSM:
        return getMockFSM(f.Underlying())
    }

    return nil
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
type MockSnapshot struct {
    logs     [][]byte
    maxIndex int
}

var _ ConfigurationStore = (*MockFSMConfigStore)(nil)

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockFSM) Apply(log *Log) interface{} {
    m.Lock()
    defer m.Unlock()
    m.logs = append(m.logs, log.Data)
    return len(m.logs)
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockFSM) Snapshot() (FSMSnapshot, error) {
    m.Lock()
    defer m.Unlock()
    return &MockSnapshot{m.logs, len(m.logs)}, nil
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockFSM) Restore(inp io.ReadCloser) error {
    m.Lock()
    defer m.Unlock()
    defer inp.Close()
    hd := codec.MsgpackHandle{}
    dec := codec.NewDecoder(inp, &hd)

    m.logs = nil
    return dec.Decode(&m.logs)
```

```
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockFSM) Logs() [][]byte {
    m.Lock()
    defer m.Unlock()
    return m.logs
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockFSMConfigStore) StoreConfiguration(index uint64, config
Configuration) {
    mm := m.FSM.(*MockFSM)
    mm.Lock()
    defer mm.Unlock()
    mm.configurations = append(mm.configurations, config)
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockSnapshot) Persist(sink SnapshotSink) error {
    hd := codec.MsgpackHandle{}
    enc := codec.NewEncoder(sink, &hd)
    if err := enc.Encode(m.logs[:m.maxIndex]); err != nil {
        sink.Cancel()
        return err
    }
    sink.Close()
    return nil
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockSnapshot) Release() {
}

// This can be used as the destination for a logger and it'll
// map them into calls to testing.T.Log, so that you only see
// the logging for failed tests.
type testLoggerAdapter struct {
    t      *testing.T
    prefix string
}

func (a *testLoggerAdapter) Write(d []byte) (int, error) {
    if d[len(d)-1] == '\n' {
        d = d[:len(d)-1]
    }
    if a.prefix != "" {
        l := a.prefix + ": " + string(d)
        if testing.Verbose() {
```

```
            fmt.Printf("testLoggerAdapter verbose: %s\n", l)
        }
        a.t.Log(l)
        return len(l), nil
    }

    a.t.Log(string(d))
    return len(d), nil
}

func newTestLogger(t *testing.T) hclog.Logger {
    return hclog.New(&hclog.LoggerOptions{
        Output: &testLoggerAdapter{t: t},
        Level:  hclog.DefaultLevel,
    })
}

func newTestLoggerWithPrefix(t *testing.T, prefix string) hclog.Logger {
    return hclog.New(&hclog.LoggerOptions{
        Output: &testLoggerAdapter{t: t, prefix: prefix},
        Level:  hclog.DefaultLevel,
    })
}

func newTestLeveledLogger(t *testing.T) hclog.Logger {
    return hclog.New(&hclog.LoggerOptions{
        Name:   "",
        Output: &testLoggerAdapter{t: t},
    })
}

func newTestLeveledLoggerWithPrefix(t *testing.T, prefix string) hclog.Logger {
    return hclog.New(&hclog.LoggerOptions{
        Name:   prefix,
        Output: &testLoggerAdapter{t: t, prefix: prefix},
    })
}

type cluster struct {
    dirs             []string
    stores           []*InmemStore
    fsms             []FSM
    snaps            []*FileSnapshotStore
    trans            []LoopbackTransport
    rafts            []*Raft
    t                *testing.T
    observationCh    chan Observation
    conf             *Config
    propagateTimeout time.Duration
    longstopTimeout  time.Duration
    logger           hclog.Logger
    startTime        time.Time
```

```
    failedLock sync.Mutex
    failedCh   chan struct{}
    failed     bool
}

func (c *cluster) Merge(other *cluster) {
    c.dirs = append(c.dirs, other.dirs...)
    c.stores = append(c.stores, other.stores...)
    c.fsms = append(c.fsms, other.fsms...)
    c.snaps = append(c.snaps, other.snaps...)
    c.trans = append(c.trans, other.trans...)
    c.rafts = append(c.rafts, other.rafts...)
}

// notifyFailed will close the failed channel which can signal the goroutine
// running the test that another goroutine has detected a failure in order to
// terminate the test.
func (c *cluster) notifyFailed() {
    c.failedLock.Lock()
    defer c.failedLock.Unlock()
    if !c.failed {
        c.failed = true
        close(c.failedCh)
    }
}

// Failf provides a logging function that fails the tests, prints the output
// with microseconds, and does not mysteriously eat the string. This can be
// safely called from goroutines but won't immediately halt the test. The
// failedCh will be closed to allow blocking functions in the main thread to
// detect the failure and react. Note that you should arrange for the main
// thread to block until all goroutines have completed in order to reliably
// fail tests using this function.
func (c *cluster) Failf(format string, args ...interface{}) {
    c.logger.Error(fmt.Sprintf(format, args...))
    c.t.Fail()
    c.notifyFailed()
}

// FailNowf provides a logging function that fails the tests, prints the output
// with microseconds, and does not mysteriously eat the string. FailNowf must
be
// called from the goroutine running the test or benchmark function, not from
// other goroutines created during the test. Calling FailNowf does not stop
// those other goroutines.
func (c *cluster) FailNowf(format string, args ...interface{}) {
    c.logger.Error(fmt.Sprintf(format, args...))
    c.t.FailNow()
}

// Close shuts down the cluster and cleans up.
func (c *cluster) Close() {
    var futures []Future
```

```
    for _, r := range c.rafts {
        futures = append(futures, r.Shutdown())
    }

    // Wait for shutdown
    limit := time.AfterFunc(c.longstopTimeout, func() {
        // We can't FailNowf here, and c.Failf won't do anything if we
        // hang, so panic.
        panic("timed out waiting for shutdown")
    })
    defer limit.Stop()

    for _, f := range futures {
        if err := f.Error(); err != nil {
            c.FailNowf("shutdown future err: %v", err)
        }
    }

    for _, d := range c.dirs {
        os.RemoveAll(d)
    }
}

// WaitEventChan returns a channel which will signal if an observation is made
// or a timeout occurs. It is possible to set a filter to look for specific
// observations. Setting timeout to 0 means that it will wait forever until a
// non-filtered observation is made.
func (c *cluster) WaitEventChan(filter FilterFn, timeout time.Duration) <-chan
struct{} {
    ch := make(chan struct{})
    go func() {
        defer close(ch)
        var timeoutCh <-chan time.Time
        if timeout > 0 {
            timeoutCh = time.After(timeout)
        }
        for {
            select {
            case <-timeoutCh:
                return

            case o, ok := <-c.observationCh:
                if !ok || filter == nil || filter(&o) {
                    return
                }
            }
        }
    }()
    return ch
}

// WaitEvent waits until an observation is made, a timeout occurs, or a test
// failure is signaled. It is possible to set a filter to look for specific
```

```
// observations. Setting timeout to 0 means that it will wait forever until a
// non-filtered observation is made or a test failure is signaled.
func (c *cluster) WaitEvent(filter FilterFn, timeout time.Duration) {
    select {
    case <-c.failedCh:
        c.t.FailNow()

    case <-c.WaitEventChan(filter, timeout):
    }
}

// WaitForReplication blocks until every FSM in the cluster has the given
// length, or the long sanity check timeout expires.
func (c *cluster) WaitForReplication(fsmLength int) {
    limitCh := time.After(c.longstopTimeout)

CHECK:
    for {
        ch := c.WaitEventChan(nil, c.conf.CommitTimeout)
        select {
        case <-c.failedCh:
            c.t.FailNow()

        case <-limitCh:
            c.FailNowf("timeout waiting for replication")

        case <-ch:
            for _, fsmRaw := range c.fsms {
                fsm := getMockFSM(fsmRaw)
                fsm.Lock()
                num := len(fsm.logs)
                fsm.Unlock()
                if num != fsmLength {
                    continue CHECK
                }
            }
            return
        }
    }
}

// pollState takes a snapshot of the state of the cluster. This might not be
// stable, so use GetInState() to apply some additional checks when waiting
// for the cluster to achieve a particular state.
func (c *cluster) pollState(s RaftState) ([]*Raft, uint64) {
    var highestTerm uint64
    in := make([]*Raft, 0, 1)
    for _, r := range c.rafts {
        if r.State() == s {
            in = append(in, r)
        }
        term := r.getCurrentTerm()
        if term > highestTerm {
```

```
            highestTerm = term
        }
    }
    return in, highestTerm
}

// GetInState polls the state of the cluster and attempts to identify when it
has
// settled into the given state.
func (c *cluster) GetInState(s RaftState) []*Raft {
    c.logger.Info("starting stability test", "raft-state", s)
    limitCh := time.After(c.longstopTimeout)

    // An election should complete after 2 * max(HeartbeatTimeout,
ElectionTimeout)
    // because of the randomised timer expiring in 1 x interval ... 2 x
interval.
    // We add a bit for propagation delay. If the election fails (e.g. because
    // two elections start at once), we will have got something through our
    // observer channel indicating a different state (i.e. one of the nodes
    // will have moved to candidate state) which will reset the timer.
    //
    // Because of an implementation peculiarity, it can actually be 3 x
timeout.
    timeout := c.conf.HeartbeatTimeout
    if timeout < c.conf.ElectionTimeout {
        timeout = c.conf.ElectionTimeout
    }
    timeout = 2*timeout + c.conf.CommitTimeout
    timer := time.NewTimer(timeout)
    defer timer.Stop()

    // Wait until we have a stable instate slice. Each time we see an
    // observation a state has changed, recheck it and if it has changed,
    // restart the timer.
    var pollStartTime = time.Now()
    for {
        inState, highestTerm := c.pollState(s)
        inStateTime := time.Now()

        // Sometimes this routine is called very early on before the
        // rafts have started up. We then timeout even though no one has
        // even started an election. So if the highest term in use is
        // zero, we know there are no raft processes that have yet issued
        // a RequestVote, and we set a long time out. This is fixed when
        // we hear the first RequestVote, at which point we reset the
        // timer.
        if highestTerm == 0 {
            timer.Reset(c.longstopTimeout)
        } else {
            timer.Reset(timeout)
        }
```

```
        // Filter will wake up whenever we observe a RequestVote.
        filter := func(ob *Observation) bool {
            switch ob.Data.(type) {
            case RaftState:
                return true
            case RequestVoteRequest:
                return true
            default:
                return false
            }
        }

        select {
        case <-c.failedCh:
            c.t.FailNow()

        case <-limitCh:
            c.FailNowf("timeout waiting for stable %s state", s)

        case <-c.WaitEventChan(filter, 0):
            c.logger.Debug("resetting stability timeout")

        case t, ok := <-timer.C:
            if !ok {
                c.FailNowf("timer channel errored")
            }

            c.logger.Info(fmt.Sprintf("stable state for %s reached at %s (%d
nodes), %s from start of poll, %s from cluster start. Timeout at %s, %s after
stability",
                s, inStateTime, len(inState), inStateTime.Sub(pollStartTime),
inStateTime.Sub(c.startTime), t, t.Sub(inStateTime)))
            return inState
        }
    }
}

// Leader waits for the cluster to elect a leader and stay in a stable state.
func (c *cluster) Leader() *Raft {
    leaders := c.GetInState(Leader)
    if len(leaders) != 1 {
        c.FailNowf("expected one leader: %v", leaders)
    }
    return leaders[0]
}

// Followers waits for the cluster to have N-1 followers and stay in a stable
// state.
func (c *cluster) Followers() []*Raft {
    expFollowers := len(c.rafts) - 1
    followers := c.GetInState(Follower)
    if len(followers) != expFollowers {
        c.FailNowf("timeout waiting for %d followers (followers are %v)",
```

```
expFollowers, followers)
    }
    return followers
}

// FullyConnect connects all the transports together.
func (c *cluster) FullyConnect() {
    c.logger.Debug("fully connecting")
    for i, t1 := range c.trans {
        for j, t2 := range c.trans {
            if i != j {
                t1.Connect(t2.LocalAddr(), t2)
                t2.Connect(t1.LocalAddr(), t1)
            }
        }
    }
}

// Disconnect disconnects all transports from the given address.
func (c *cluster) Disconnect(a ServerAddress) {
    c.logger.Debug("disconnecting", "address", a)
    for _, t := range c.trans {
        if t.LocalAddr() == a {
            t.DisconnectAll()
        } else {
            t.Disconnect(a)
        }
    }
}

// Partition keeps the given list of addresses connected but isolates them
// from the other members of the cluster.
func (c *cluster) Partition(far []ServerAddress) {
    c.logger.Debug("partitioning", "addresses", far)

    // Gather the set of nodes on the "near" side of the partition (we
    // will call the supplied list of nodes the "far" side).
    near := make(map[ServerAddress]struct{})
OUTER:
    for _, t := range c.trans {
        l := t.LocalAddr()
        for _, a := range far {
            if l == a {
                continue OUTER
            }
        }
        near[l] = struct{}{}
    }

    // Now fixup all the connections. The near side will be separated from
    // the far side, and vice-versa.
    for _, t := range c.trans {
        l := t.LocalAddr()
```

```
        if _, ok := near[l]; ok {
            for _, a := range far {
                t.Disconnect(a)
            }
        } else {
            for a := range near {
                t.Disconnect(a)
            }
        }
    }
}

// IndexOf returns the index of the given raft instance.
func (c *cluster) IndexOf(r *Raft) int {
    for i, n := range c.rafts {
        if n == r {
            return i
        }
    }
    return -1
}

// EnsureLeader checks that ALL the nodes think the leader is the given
expected
// leader.
func (c *cluster) EnsureLeader(t *testing.T, expect ServerAddress) {
    // We assume c.Leader() has been called already; now check all the rafts
    // think the leader is correct
    fail := false
    for _, r := range c.rafts {
        leader := ServerAddress(r.Leader())
        if leader != expect {
            if leader == "" {
                leader = "[none]"
            }
            if expect == "" {
                c.logger.Error("peer sees incorrect leader", "peer", r,
"leader", leader, "expected-leader", "[none]")
            } else {
                c.logger.Error("peer sees incorrect leader", "peer", r,
"leader", leader, "expected-leader", expect)
            }
            fail = true
        }
    }
    if fail {
        c.FailNowf("at least one peer has the wrong notion of leader")
    }
}

// EnsureSame makes sure all the FSMs have the same contents.
func (c *cluster) EnsureSame(t *testing.T) {
    limit := time.Now().Add(c.longstopTimeout)
```

```
    first := getMockFSM(c.fsms[0])

CHECK:
    first.Lock()
    for i, fsmRaw := range c.fsms {
        fsm := getMockFSM(fsmRaw)
        if i == 0 {
            continue
        }
        fsm.Lock()

        if len(first.logs) != len(fsm.logs) {
            fsm.Unlock()
            if time.Now().After(limit) {
                c.FailNowf("FSM log length mismatch: %d %d",
                    len(first.logs), len(fsm.logs))
            } else {
                goto WAIT
            }
        }

        for idx := 0; idx < len(first.logs); idx++ {
            if bytes.Compare(first.logs[idx], fsm.logs[idx]) != 0 {
                fsm.Unlock()
                if time.Now().After(limit) {
                    c.FailNowf("FSM log mismatch at index %d", idx)
                } else {
                    goto WAIT
                }
            }
        }
        if len(first.configurations) != len(fsm.configurations) {
            fsm.Unlock()
            if time.Now().After(limit) {
                c.FailNowf("FSM configuration length mismatch: %d %d",
                    len(first.logs), len(fsm.logs))
            } else {
                goto WAIT
            }
        }

        for idx := 0; idx < len(first.configurations); idx++ {
            if !reflect.DeepEqual(first.configurations[idx],
fsm.configurations[idx]) {
                fsm.Unlock()
                if time.Now().After(limit) {
                    c.FailNowf("FSM configuration mismatch at index %d: %v,
%v", idx, first.configurations[idx], fsm.configurations[idx])
                } else {
                    goto WAIT
                }
            }
        }
```

```
        fsm.Unlock()
    }

    first.Unlock()
    return

WAIT:
    first.Unlock()
    c.WaitEvent(nil, c.conf.CommitTimeout)
    goto CHECK
}

// getConfiguration returns the configuration of the given Raft instance, or
// fails the test if there's an error
func (c *cluster) getConfiguration(r *Raft) Configuration {
    future := r.GetConfiguration()
    if err := future.Error(); err != nil {
        c.FailNowf("failed to get configuration: %v", err)
        return Configuration{}
    }

    return future.Configuration()
}

// EnsureSamePeers makes sure all the rafts have the same set of peers.
func (c *cluster) EnsureSamePeers(t *testing.T) {
    limit := time.Now().Add(c.longstopTimeout)
    peerSet := c.getConfiguration(c.rafts[0])

CHECK:
    for i, raft := range c.rafts {
        if i == 0 {
            continue
        }

        otherSet := c.getConfiguration(raft)
        if !reflect.DeepEqual(peerSet, otherSet) {
            if time.Now().After(limit) {
                c.FailNowf("peer mismatch: %+v %+v", peerSet, otherSet)
            } else {
                goto WAIT
            }
        }
    }
    return

WAIT:
    c.WaitEvent(nil, c.conf.CommitTimeout)
    goto CHECK
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
```

```
type MakeClusterOpts struct {
    Peers           int
    Bootstrap       bool
    Conf            *Config
    ConfigStoreFSM  bool
    MakeFSMFunc     func() FSM
    LongstopTimeout time.Duration
}

// makeCluster will return a cluster with the given config and number of peers.
// If bootstrap is true, the servers will know about each other before
starting,
// otherwise their transports will be wired up but they won't yet have
configured
// each other.
func makeCluster(t *testing.T, opts *MakeClusterOpts) *cluster {
    if opts.Conf == nil {
        opts.Conf = inmemConfig(t)
    }

    c := &cluster{
        observationCh: make(chan Observation, 1024),
        conf:          opts.Conf,
        // Propagation takes a maximum of 2 heartbeat timeouts (time to
        // get a new heartbeat that would cause a commit) plus a bit.
        propagateTimeout: opts.Conf.HeartbeatTimeout*2 +
opts.Conf.CommitTimeout,
        longstopTimeout:  5 * time.Second,
        logger:           newTestLoggerWithPrefix(t, "cluster"),
        failedCh:         make(chan struct{}),
    }
    if opts.LongstopTimeout > 0 {
        c.longstopTimeout = opts.LongstopTimeout
    }

    c.t = t
    var configuration Configuration

    // Setup the stores and transports
    for i := 0; i < opts.Peers; i++ {
        dir, err := ioutil.TempDir("", "raft")
        if err != nil {
            c.FailNowf("err: %v", err)
        }

        store := NewInmemStore()
        c.dirs = append(c.dirs, dir)
        c.stores = append(c.stores, store)
        if opts.ConfigStoreFSM {
            c.fsms = append(c.fsms, &MockFSMConfigStore{
                FSM: &MockFSM{},
            })
        } else {
```

```
            var fsm FSM
            if opts.MakeFSMFunc != nil {
                fsm = opts.MakeFSMFunc()
            } else {
                fsm = &MockFSM{}
            }
            c.fsms = append(c.fsms, fsm)
        }

        dir2, snap := FileSnapTest(t)
        c.dirs = append(c.dirs, dir2)
        c.snaps = append(c.snaps, snap)

        addr, trans := NewInmemTransport("")
        c.trans = append(c.trans, trans)
        localID := ServerID(fmt.Sprintf("server-%s", addr))
        if opts.Conf.ProtocolVersion < 3 {
            localID = ServerID(addr)
        }
        configuration.Servers = append(configuration.Servers, Server{
            Suffrage: Voter,
            ID:       localID,
            Address:  addr,
        })
    }

    // Wire the transports together
    c.FullyConnect()

    // Create all the rafts
    c.startTime = time.Now()
    for i := 0; i < opts.Peers; i++ {
        logs := c.stores[i]
        store := c.stores[i]
        snap := c.snaps[i]
        trans := c.trans[i]

        peerConf := opts.Conf
        peerConf.LocalID = configuration.Servers[i].ID
        peerConf.Logger = newTestLeveledLoggerWithPrefix(t,
string(configuration.Servers[i].ID))

        if opts.Bootstrap {
            err := BootstrapCluster(peerConf, logs, store, snap, trans,
configuration)
            if err != nil {
                c.FailNowf("BootstrapCluster failed: %v", err)
            }
        }

        raft, err := NewRaft(peerConf, c.fsms[i], logs, store, snap, trans)
        if err != nil {
            c.FailNowf("NewRaft failed: %v", err)
```

```
        }

        raft.RegisterObserver(NewObserver(c.observationCh, false, nil))
        if err != nil {
            c.FailNowf("RegisterObserver failed: %v", err)
        }
        c.rafts = append(c.rafts, raft)
    }

    return c
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func MakeCluster(n int, t *testing.T, conf *Config) *cluster {
    return makeCluster(t, &MakeClusterOpts{
        Peers:     n,
        Bootstrap: true,
        Conf:      conf,
    })
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func MakeClusterNoBootstrap(n int, t *testing.T, conf *Config) *cluster {
    return makeCluster(t, &MakeClusterOpts{
        Peers: n,
        Conf:  conf,
    })
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func MakeClusterCustom(t *testing.T, opts *MakeClusterOpts) *cluster {
    return makeCluster(t, opts)
}

// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func FileSnapTest(t *testing.T) (string, *FileSnapshotStore) {
    // Create a test dir
    dir, err := ioutil.TempDir("", "raft")
    if err != nil {
        t.Fatalf("err: %v ", err)
    }

    snap, err := NewFileSnapshotStoreWithLogger(dir, 3, newTestLogger(t))
    if err != nil {
        t.Fatalf("err: %v", err)
    }
    return dir, snap
}
```

../raft/testing_batch.go

```
// +build batchtest

package raft

func init() {
    userSnapshotErrorsOnNoData = false
}

// ApplyBatch enables MockFSM to satisfy the BatchingFSM interface. This
// function is gated by the batchtest build flag.
//
// NOTE: This is exposed for middleware testing purposes and is not a stable
API
func (m *MockFSM) ApplyBatch(logs []*Log) []interface{} {
    m.Lock()
    defer m.Unlock()

    ret := make([]interface{}, len(logs))
    for i, log := range logs {
        switch log.Type {
        case LogCommand:
            m.logs = append(m.logs, log.Data)
            ret[i] = len(m.logs)
        default:
            ret[i] = nil
        }
    }

    return ret
}
```

../raft/transport.go

```
package raft

import (
    "io"
    "time"
)

// RPCResponse captures both a response and a potential error.
type RPCResponse struct {
    Response interface{}
    Error    error
}

// RPC has a command, and provides a response mechanism.
type RPC struct {
    Command  interface{}
    Reader   io.Reader // Set only for InstallSnapshot
    RespChan chan<- RPCResponse
}

// Respond is used to respond with a response, error or both
func (r *RPC) Respond(resp interface{}, err error) {
    r.RespChan <- RPCResponse{resp, err}
}

// Transport provides an interface for network transports
// to allow Raft to communicate with other nodes.
type Transport interface {
    // Consumer returns a channel that can be used to
    // consume and respond to RPC requests.
    Consumer() <-chan RPC

    // LocalAddr is used to return our local address to distinguish from our
peers.
    LocalAddr() ServerAddress

    // AppendEntriesPipeline returns an interface that can be used to pipeline
    // AppendEntries requests.
    AppendEntriesPipeline(id ServerID, target ServerAddress) (AppendPipeline,
error)

    // AppendEntries sends the appropriate RPC to the target node.
    AppendEntries(id ServerID, target ServerAddress, args
*AppendEntriesRequest, resp *AppendEntriesResponse) error

    // RequestVote sends the appropriate RPC to the target node.
    RequestVote(id ServerID, target ServerAddress, args *RequestVoteRequest,
resp *RequestVoteResponse) error

    // InstallSnapshot is used to push a snapshot down to a follower. The data
is read from
    // the ReadCloser and streamed to the client.
```

```
    InstallSnapshot(id ServerID, target ServerAddress, args
*InstallSnapshotRequest, resp *InstallSnapshotResponse, data io.Reader) error

    // EncodePeer is used to serialize a peer's address.
    EncodePeer(id ServerID, addr ServerAddress) []byte

    // DecodePeer is used to deserialize a peer's address.
    DecodePeer([]byte) ServerAddress

    // SetHeartbeatHandler is used to setup a heartbeat handler
    // as a fast-pass. This is to avoid head-of-line blocking from
    // disk IO. If a Transport does not support this, it can simply
    // ignore the call, and push the heartbeat onto the Consumer channel.
    SetHeartbeatHandler(cb func(rpc RPC))

    // TimeoutNow is used to start a leadership transfer to the target node.
    TimeoutNow(id ServerID, target ServerAddress, args *TimeoutNowRequest, resp
*TimeoutNowResponse) error
}

// WithClose is an interface that a transport may provide which
// allows a transport to be shut down cleanly when a Raft instance
// shuts down.
//
// It is defined separately from Transport as unfortunately it wasn't in the
// original interface specification.
type WithClose interface {
    // Close permanently closes a transport, stopping
    // any associated goroutines and freeing other resources.
    Close() error
}

// LoopbackTransport is an interface that provides a loopback transport
suitable for testing
// e.g. InmemTransport. It's there so we don't have to rewrite tests.
type LoopbackTransport interface {
    Transport // Embedded transport reference
    WithPeers // Embedded peer management
    WithClose // with a close routine
}

// WithPeers is an interface that a transport may provide which allows for
connection and
// disconnection. Unless the transport is a loopback transport, the transport
specified to
// "Connect" is likely to be nil.
type WithPeers interface {
    Connect(peer ServerAddress, t Transport) // Connect a peer
    Disconnect(peer ServerAddress)           // Disconnect a given peer
    DisconnectAll()                          // Disconnect all peers, possibly
to reconnect them later
}
```

```
// AppendPipeline is used for pipelining AppendEntries requests. It is used
// to increase the replication throughput by masking latency and better
// utilizing bandwidth.
type AppendPipeline interface {
    // AppendEntries is used to add another request to the pipeline.
    // The send may block which is an effective form of back-pressure.
    AppendEntries(args *AppendEntriesRequest, resp *AppendEntriesResponse)
(AppendFuture, error)

    // Consumer returns a channel that can be used to consume
    // response futures when they are ready.
    Consumer() <-chan AppendFuture

    // Close closes the pipeline and cancels all inflight RPCs
    Close() error
}

// AppendFuture is used to return information about a pipelined AppendEntries
request.
type AppendFuture interface {
    Future

    // Start returns the time that the append request was started.
    // It is always OK to call this method.
    Start() time.Time

    // Request holds the parameters of the AppendEntries call.
    // It is always OK to call this method.
    Request() *AppendEntriesRequest

    // Response holds the results of the AppendEntries call.
    // This method must only be called after the Error
    // method returns, and will only be valid on success.
    Response() *AppendEntriesResponse
}
```

../raft/transport_test.go

```
package raft

import (
    "bytes"
    "reflect"
    "testing"
    "time"
)

const (
    TTInmem = iota

    // NOTE: must be last
    numTestTransports
)

func NewTestTransport(ttype int, addr ServerAddress) (ServerAddress,
LoopbackTransport) {
    switch ttype {
    case TTInmem:
        return NewInmemTransport(addr)
    default:
        panic("Unknown transport type")
    }
}

func TestTransport_StartStop(t *testing.T) {
    for ttype := 0; ttype < numTestTransports; ttype++ {
        _, trans := NewTestTransport(ttype, "")
        if err := trans.Close(); err != nil {
            t.Fatalf("err: %v", err)
        }
    }
}

func TestTransport_AppendEntries(t *testing.T) {
    for ttype := 0; ttype < numTestTransports; ttype++ {
        addr1, trans1 := NewTestTransport(ttype, "")
        defer trans1.Close()
        rpcCh := trans1.Consumer()

        // Make the RPC request
        args := AppendEntriesRequest{
            Term:         10,
            Leader:       []byte("cartman"),
            PrevLogEntry: 100,
            PrevLogTerm:  4,
            Entries: []*Log{
                {
                    Index: 101,
                    Term:  4,
                    Type:  LogNoop,
```

```
                },
            },
            LeaderCommitIndex: 90,
        }
        resp := AppendEntriesResponse{
            Term:    4,
            LastLog: 90,
            Success: true,
        }

        // Listen for a request
        go func() {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*AppendEntriesRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }
                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
            }
        }()

        // Transport 2 makes outbound request
        addr2, trans2 := NewTestTransport(ttype, "")
        defer trans2.Close()

        trans1.Connect(addr2, trans2)
        trans2.Connect(addr1, trans1)

        var out AppendEntriesResponse
        if err := trans2.AppendEntries("id1", trans1.LocalAddr(), &args, &out);
err != nil {
            t.Fatalf("err: %v", err)
        }

        // Verify the response
        if !reflect.DeepEqual(resp, out) {
            t.Fatalf("command mismatch: %#v %#v", resp, out)
        }
    }
}

func TestTransport_AppendEntriesPipeline(t *testing.T) {
    for ttype := 0; ttype < numTestTransports; ttype++ {
        addr1, trans1 := NewTestTransport(ttype, "")
        defer trans1.Close()
        rpcCh := trans1.Consumer()

        // Make the RPC request
```

```
        args := AppendEntriesRequest{
            Term:         10,
            Leader:       []byte("cartman"),
            PrevLogEntry: 100,
            PrevLogTerm:  4,
            Entries: []*Log{
                {
                    Index: 101,
                    Term:  4,
                    Type:  LogNoop,
                },
            },
            LeaderCommitIndex: 90,
        }
        resp := AppendEntriesResponse{
            Term:    4,
            LastLog: 90,
            Success: true,
        }

        // Listen for a request
        go func() {
            for i := 0; i < 10; i++ {
                select {
                case rpc := <-rpcCh:
                    // Verify the command
                    req := rpc.Command.(*AppendEntriesRequest)
                    if !reflect.DeepEqual(req, &args) {
                        t.Fatalf("command mismatch: %#v %#v", *req, args)
                    }
                    rpc.Respond(&resp, nil)

                case <-time.After(200 * time.Millisecond):
                    t.Fatalf("timeout")
                }
            }
        }()

        // Transport 2 makes outbound request
        addr2, trans2 := NewTestTransport(ttype, "")
        defer trans2.Close()

        trans1.Connect(addr2, trans2)
        trans2.Connect(addr1, trans1)

        pipeline, err := trans2.AppendEntriesPipeline("id1",
trans1.LocalAddr())
        if err != nil {
            t.Fatalf("err: %v", err)
        }
        defer pipeline.Close()
        for i := 0; i < 10; i++ {
            out := new(AppendEntriesResponse)
```

```
            if _, err := pipeline.AppendEntries(&args, out); err != nil {
                t.Fatalf("err: %v", err)
            }
        }

        respCh := pipeline.Consumer()
        for i := 0; i < 10; i++ {
            select {
            case ready := <-respCh:
                // Verify the response
                if !reflect.DeepEqual(&resp, ready.Response()) {
                    t.Fatalf("command mismatch: %#v %#v", &resp,
ready.Response())
                }
            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
            }
        }
    }
}

func TestTransport_RequestVote(t *testing.T) {
    for ttype := 0; ttype < numTestTransports; ttype++ {
        addr1, trans1 := NewTestTransport(ttype, "")
        defer trans1.Close()
        rpcCh := trans1.Consumer()

        // Make the RPC request
        args := RequestVoteRequest{
            Term:         20,
            Candidate:    []byte("butters"),
            LastLogIndex: 100,
            LastLogTerm:  19,
        }
        resp := RequestVoteResponse{
            Term:    100,
            Granted: false,
        }

        // Listen for a request
        go func() {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*RequestVoteRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }

                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
```

```
            }
        }()

        // Transport 2 makes outbound request
        addr2, trans2 := NewTestTransport(ttype, "")
        defer trans2.Close()

        trans1.Connect(addr2, trans2)
        trans2.Connect(addr1, trans1)

        var out RequestVoteResponse
        if err := trans2.RequestVote("id1", trans1.LocalAddr(), &args, &out);
err != nil {
            t.Fatalf("err: %v", err)
        }

        // Verify the response
        if !reflect.DeepEqual(resp, out) {
            t.Fatalf("command mismatch: %#v %#v", resp, out)
        }
    }
}

func TestTransport_InstallSnapshot(t *testing.T) {
    for ttype := 0; ttype < numTestTransports; ttype++ {
        addr1, trans1 := NewTestTransport(ttype, "")
        defer trans1.Close()
        rpcCh := trans1.Consumer()

        // Make the RPC request
        args := InstallSnapshotRequest{
            Term:         10,
            Leader:       []byte("kyle"),
            LastLogIndex: 100,
            LastLogTerm:  9,
            Peers:        []byte("blah blah"),
            Size:         10,
        }
        resp := InstallSnapshotResponse{
            Term:    10,
            Success: true,
        }

        // Listen for a request
        go func() {
            select {
            case rpc := <-rpcCh:
                // Verify the command
                req := rpc.Command.(*InstallSnapshotRequest)
                if !reflect.DeepEqual(req, &args) {
                    t.Fatalf("command mismatch: %#v %#v", *req, args)
                }
```

```
                // Try to read the bytes
                buf := make([]byte, 10)
                rpc.Reader.Read(buf)

                // Compare
                if bytes.Compare(buf, []byte("0123456789")) != 0 {
                    t.Fatalf("bad buf %v", buf)
                }

                rpc.Respond(&resp, nil)

            case <-time.After(200 * time.Millisecond):
                t.Fatalf("timeout")
            }
        }()

        // Transport 2 makes outbound request
        addr2, trans2 := NewTestTransport(ttype, "")
        defer trans2.Close()

        trans1.Connect(addr2, trans2)
        trans2.Connect(addr1, trans1)

        // Create a buffer
        buf := bytes.NewBuffer([]byte("0123456789"))

        var out InstallSnapshotResponse
        if err := trans2.InstallSnapshot("id1", trans1.LocalAddr(), &args,
&out, buf); err != nil {
            t.Fatalf("err: %v", err)
        }

        // Verify the response
        if !reflect.DeepEqual(resp, out) {
            t.Fatalf("command mismatch: %#v %#v", resp, out)
        }
    }
}

func TestTransport_EncodeDecode(t *testing.T) {
    for ttype := 0; ttype < numTestTransports; ttype++ {
        _, trans1 := NewTestTransport(ttype, "")
        defer trans1.Close()

        local := trans1.LocalAddr()
        enc := trans1.EncodePeer("aaaa", local)
        dec := trans1.DecodePeer(enc)

        if dec != local {
            t.Fatalf("enc/dec fail: %v %v", dec, local)
        }
    }
```

```
}
```

../raft/util.go

```
package raft

import (
    "bytes"
    crand "crypto/rand"
    "fmt"
    "math"
    "math/big"
    "math/rand"
    "time"

    "github.com/hashicorp/go-msgpack/codec"
)

func init() {
    // Ensure we use a high-entropy seed for the psuedo-random generator
    rand.Seed(newSeed())
}

// returns an int64 from a crypto random source
// can be used to seed a source for a math/rand.
func newSeed() int64 {
    r, err := crand.Int(crand.Reader, big.NewInt(math.MaxInt64))
    if err != nil {
        panic(fmt.Errorf("failed to read random bytes: %v", err))
    }
    return r.Int64()
}

// randomTimeout returns a value that is between the minVal and 2x minVal.
func randomTimeout(minVal time.Duration) <-chan time.Time {
    if minVal == 0 {
        return nil
    }
    extra := (time.Duration(rand.Int63()) % minVal)
    return time.After(minVal + extra)
}

// min returns the minimum.
func min(a, b uint64) uint64 {
    if a <= b {
        return a
    }
    return b
}

// max returns the maximum.
func max(a, b uint64) uint64 {
    if a >= b {
        return a
    }
    return b
```

```
}

// generateUUID is used to generate a random UUID.
func generateUUID() string {
    buf := make([]byte, 16)
    if _, err := crand.Read(buf); err != nil {
        panic(fmt.Errorf("failed to read random bytes: %v", err))
    }

    return fmt.Sprintf("%08x-%04x-%04x-%04x-%12x",
        buf[0:4],
        buf[4:6],
        buf[6:8],
        buf[8:10],
        buf[10:16])
}

// asyncNotifyCh is used to do an async channel send
// to a single channel without blocking.
func asyncNotifyCh(ch chan struct{}) {
    select {
    case ch <- struct{}{}:
    default:
    }
}

// drainNotifyCh empties out a single-item notification channel without
// blocking, and returns whether it received anything.
func drainNotifyCh(ch chan struct{}) bool {
    select {
    case <-ch:
        return true
    default:
        return false
    }
}

// asyncNotifyBool is used to do an async notification
// on a bool channel.
func asyncNotifyBool(ch chan bool, v bool) {
    select {
    case ch <- v:
    default:
    }
}

// Decode reverses the encode operation on a byte slice input.
func decodeMsgPack(buf []byte, out interface{}) error {
    r := bytes.NewBuffer(buf)
    hd := codec.MsgpackHandle{}
    dec := codec.NewDecoder(r, &hd)
    return dec.Decode(out)
}
```

```
// Encode writes an encoded object to a new bytes buffer.
func encodeMsgPack(in interface{}) (*bytes.Buffer, error) {
    buf := bytes.NewBuffer(nil)
    hd := codec.MsgpackHandle{}
    enc := codec.NewEncoder(buf, &hd)
    err := enc.Encode(in)
    return buf, err
}

// backoff is used to compute an exponential backoff
// duration. Base time is scaled by the current round,
// up to some maximum scale factor.
func backoff(base time.Duration, round, limit uint64) time.Duration {
    power := min(round, limit)
    for power > 2 {
        base *= 2
        power--
    }
    return base
}

// Needed for sorting []uint64, used to determine commitment
type uint64Slice []uint64

func (p uint64Slice) Len() int           { return len(p) }
func (p uint64Slice) Less(i, j int) bool { return p[i] < p[j] }
func (p uint64Slice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
```

../raft/util_test.go

```
package raft

import (
    "regexp"
    "testing"
    "time"
)

func TestRandomTimeout(t *testing.T) {
    start := time.Now()
    timeout := randomTimeout(time.Millisecond)

    select {
    case <-timeout:
        diff := time.Now().Sub(start)
        if diff < time.Millisecond {
            t.Fatalf("fired early")
        }
    case <-time.After(3 * time.Millisecond):
        t.Fatalf("timeout")
    }
}

func TestNewSeed(t *testing.T) {
    vals := make(map[int64]bool)
    for i := 0; i < 1000; i++ {
        seed := newSeed()
        if _, exists := vals[seed]; exists {
            t.Fatal("newSeed() return a value it'd previously returned")
        }
        vals[seed] = true
    }
}

func TestRandomTimeout_NoTime(t *testing.T) {
    timeout := randomTimeout(0)
    if timeout != nil {
        t.Fatalf("expected nil channel")
    }
}

func TestMin(t *testing.T) {
    if min(1, 1) != 1 {
        t.Fatalf("bad min")
    }
    if min(2, 1) != 1 {
        t.Fatalf("bad min")
    }
    if min(1, 2) != 1 {
        t.Fatalf("bad min")
    }
}
```

```
func TestMax(t *testing.T) {
    if max(1, 1) != 1 {
        t.Fatalf("bad max")
    }
    if max(2, 1) != 2 {
        t.Fatalf("bad max")
    }
    if max(1, 2) != 2 {
        t.Fatalf("bad max")
    }
}

func TestGenerateUUID(t *testing.T) {
    prev := generateUUID()
    for i := 0; i < 100; i++ {
        id := generateUUID()
        if prev == id {
            t.Fatalf("Should get a new ID!")
        }

        matched, err := regexp.MatchString(
            `[\da-f]{8}-[\da-f]{4}-[\da-f]{4}-[\da-f]{4}-[\da-f]{12}`, id)
        if !matched || err != nil {
            t.Fatalf("expected match %s %v %s", id, matched, err)
        }
    }
}

func TestBackoff(t *testing.T) {
    b := backoff(10*time.Millisecond, 1, 8)
    if b != 10*time.Millisecond {
        t.Fatalf("bad: %v", b)
    }

    b = backoff(20*time.Millisecond, 2, 8)
    if b != 20*time.Millisecond {
        t.Fatalf("bad: %v", b)
    }

    b = backoff(10*time.Millisecond, 8, 8)
    if b != 640*time.Millisecond {
        t.Fatalf("bad: %v", b)
    }

    b = backoff(10*time.Millisecond, 9, 8)
    if b != 640*time.Millisecond {
        t.Fatalf("bad: %v", b)
    }
}
```

../raft/bench/bench.go

```go
package raftbench

// raftbench provides common benchmarking functions which can be used by
// anything which implements the raft.LogStore and raft.StableStore interfaces.
// All functions accept these interfaces and perform benchmarking. This
// makes comparing backend performance easier by sharing the tests.

import (
    "github.com/hashicorp/raft"
    "testing"
)

func FirstIndex(b *testing.B, store raft.LogStore) {
    // Create some fake data
    var logs []*raft.Log
    for i := 1; i < 10; i++ {
        logs = append(logs, &raft.Log{Index: uint64(i), Data: []byte("data")})
    }
    if err := store.StoreLogs(logs); err != nil {
        b.Fatalf("err: %s", err)
    }
    b.ResetTimer()

    // Run FirstIndex a number of times
    for n := 0; n < b.N; n++ {
        store.FirstIndex()
    }
}

func LastIndex(b *testing.B, store raft.LogStore) {
    // Create some fake data
    var logs []*raft.Log
    for i := 1; i < 10; i++ {
        logs = append(logs, &raft.Log{Index: uint64(i), Data: []byte("data")})
    }
    if err := store.StoreLogs(logs); err != nil {
        b.Fatalf("err: %s", err)
    }
    b.ResetTimer()

    // Run LastIndex a number of times
    for n := 0; n < b.N; n++ {
        store.LastIndex()
    }
}

func GetLog(b *testing.B, store raft.LogStore) {
    // Create some fake data
    var logs []*raft.Log
    for i := 1; i < 10; i++ {
        logs = append(logs, &raft.Log{Index: uint64(i), Data: []byte("data")})
    }
```

```
    if err := store.StoreLogs(logs); err != nil {
        b.Fatalf("err: %s", err)
    }
    b.ResetTimer()

    // Run GetLog a number of times
    for n := 0; n < b.N; n++ {
        if err := store.GetLog(5, new(raft.Log)); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
}

func StoreLog(b *testing.B, store raft.LogStore) {
    // Run StoreLog a number of times
    for n := 0; n < b.N; n++ {
        log := &raft.Log{Index: uint64(n), Data: []byte("data")}
        if err := store.StoreLog(log); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
}

func StoreLogs(b *testing.B, store raft.LogStore) {
    // Run StoreLogs a number of times. We want to set multiple logs each
    // run, so we create 3 logs with incrementing indexes for each iteration.
    for n := 0; n < b.N; n++ {
        b.StopTimer()
        offset := 3 * (n + 1)
        logs := []*raft.Log{
            {Index: uint64(offset - 2), Data: []byte("data")},
            {Index: uint64(offset - 1), Data: []byte("data")},
            {Index: uint64(offset), Data: []byte("data")},
        }
        b.StartTimer()

        if err := store.StoreLogs(logs); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
}

func DeleteRange(b *testing.B, store raft.LogStore) {
    // Create some fake data. In this case, we create 3 new log entries for
each
    // test case, and separate them by index in multiples of 10. This allows
    // some room so that we can test deleting ranges with "extra" logs to
    // to ensure we stop going to the database once our max index is hit.
    var logs []*raft.Log
    for n := 0; n < b.N; n++ {
        offset := 10 * n
        for i := offset; i < offset+3; i++ {
            logs = append(logs, &raft.Log{Index: uint64(i), Data:
```

```
[]byte("data")})
        }
    }
    if err := store.StoreLogs(logs); err != nil {
        b.Fatalf("err: %s", err)
    }
    b.ResetTimer()

    // Delete a range of the data
    for n := 0; n < b.N; n++ {
        offset := 10 * n
        if err := store.DeleteRange(uint64(offset), uint64(offset+9)); err !=
nil {
            b.Fatalf("err: %s", err)
        }
    }
}

func Set(b *testing.B, store raft.StableStore) {
    // Run Set a number of times
    for n := 0; n < b.N; n++ {
        if err := store.Set([]byte{byte(n)}, []byte("val")); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
}

func Get(b *testing.B, store raft.StableStore) {
    // Create some fake data
    for i := 1; i < 10; i++ {
        if err := store.Set([]byte{byte(i)}, []byte("val")); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
    b.ResetTimer()

    // Run Get a number of times
    for n := 0; n < b.N; n++ {
        if _, err := store.Get([]byte{0x05}); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
}

func SetUint64(b *testing.B, store raft.StableStore) {
    // Run SetUint64 a number of times
    for n := 0; n < b.N; n++ {
        if err := store.SetUint64([]byte{byte(n)}, uint64(n)); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
}
```

```
func GetUint64(b *testing.B, store raft.StableStore) {
    // Create some fake data
    for i := 0; i < 10; i++ {
        if err := store.SetUint64([]byte{byte(i)}, uint64(i)); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
    b.ResetTimer()

    // Run GetUint64 a number of times
    for n := 0; n < b.N; n++ {
        if _, err := store.Get([]byte{0x05}); err != nil {
            b.Fatalf("err: %s", err)
        }
    }
}
```

../raft/fuzzy/apply_src.go

```
package fuzzy

import (
    "hash/fnv"
    "math/rand"
    "testing"
    "time"
)

type applySource struct {
    rnd  *rand.Rand
    seed int64
}

// newApplySource will create a new source, any source created with the same
seed will generate the same sequence of data.
func newApplySource(seed string) *applySource {
    h := fnv.New32()
    h.Write([]byte(seed))
    s := &applySource{seed: int64(h.Sum32())}
    s.reset()
    return s
}

// reset this source back to its initial state, it'll generate the same
sequence of data it initially did
func (a *applySource) reset() {
    a.rnd = rand.New(rand.NewSource(a.seed))
}

func (a *applySource) nextEntry() []byte {
    const sz = 33
    r := make([]byte, sz)
    for i := 0; i < len(r); i++ {
        r[i] = byte(a.rnd.Int31n(256))
    }
    return r
}

type clusterApplier struct {
    stopCh  chan bool
    applied uint64
    src     *applySource
}

// runs apply in chunks of n to the cluster, use the returned Applier to Stop()
it
func (a *applySource) apply(t *testing.T, c *cluster, n uint) *clusterApplier {
    ap := &clusterApplier{stopCh: make(chan bool), src: a}
    go ap.apply(t, c, n)
    return ap
}
```

```
func (ca *clusterApplier) apply(t *testing.T, c *cluster, n uint) {
    for true {
        select {
        case <-ca.stopCh:
            return
        default:
            ca.applied += c.ApplyN(t, 5*time.Second, ca.src, n)
        }
    }
}

func (ca *clusterApplier) stop() {
    ca.stopCh <- true
    close(ca.stopCh)
}
```

../raft/fuzzy/cluster.go

```
package fuzzy

import (
    "bytes"
    "fmt"
    "io"
    "io/ioutil"
    "os"
    "path/filepath"
    "testing"
    "time"

    "github.com/hashicorp/go-hclog"

    "github.com/hashicorp/raft"
)

type appliedItem struct {
    index uint64
    data  []byte
}

type cluster struct {
    nodes            []*raftNode
    removedNodes     []*raftNode
    lastApplySuccess raft.ApplyFuture
    lastApplyFailure raft.ApplyFuture
    applied          []appliedItem
    log              Logger
    transports       *transports
    hooks            TransportHooks
}

// Logger is abstract type for debug log messages
type Logger interface {
    Log(v ...interface{})
    Logf(s string, v ...interface{})
}

// LoggerAdapter allows a log.Logger to be used with the local Logger interface
type LoggerAdapter struct {
    log hclog.Logger
}

// Log a message to the contained debug log
func (a *LoggerAdapter) Log(v ...interface{}) {
    a.log.Info(fmt.Sprint(v...))
}

// Logf will record a formatted message to the contained debug log
func (a *LoggerAdapter) Logf(s string, v ...interface{}) {
    a.log.Info(fmt.Sprintf(s, v...))
```

```
}

func newRaftCluster(t *testing.T, logWriter io.Writer, namePrefix string, n
uint, transportHooks TransportHooks) *cluster {
    res := make([]*raftNode, 0, n)
    names := make([]string, 0, n)
    for i := uint(0); i < n; i++ {
        names = append(names, nodeName(namePrefix, i))
    }
    l := hclog.New(&hclog.LoggerOptions{
        Output: logWriter,
        Level:  hclog.DefaultLevel,
    })
    transports := newTransports(l)
    for _, i := range names {

        r, err := newRaftNode(hclog.New(&hclog.LoggerOptions{
            Name:   i + ":",
            Output: logWriter,
            Level:  hclog.DefaultLevel,
        }), transports, transportHooks, names, i)
        if err != nil {
            t.Fatalf("Unable to create raftNode:%v : %v", i, err)
        }
        res = append(res, r)
    }
    return &cluster{
        nodes:        res,
        removedNodes: make([]*raftNode, 0, n),
        applied:      make([]appliedItem, 0, 1024),
        log:          &LoggerAdapter{l},
        transports:   transports,
        hooks:        transportHooks,
    }
}

func (c *cluster) CreateAndAddNode(t *testing.T, logWriter io.Writer,
namePrefix string, nodeNum uint) error {
    name := nodeName(namePrefix, nodeNum)
    rn, err := newRaftNode(hclog.New(&hclog.LoggerOptions{
        Name:   name + ":",
        Output: logWriter,
        Level:  hclog.DefaultLevel,
    }), c.transports, c.hooks, nil, name)
    if err != nil {
        t.Fatalf("Unable to create raftNode:%v : %v", name, err)
    }
    c.nodes = append(c.nodes, rn)
    f := c.Leader(time.Minute).raft.AddVoter(raft.ServerID(name),
raft.ServerAddress(name), 0, 0)
    return f.Error()
}
```

```
func nodeName(prefix string, num uint) string {
    return fmt.Sprintf("%v_%d", prefix, num)
}

func (c *cluster) RemoveNode(t *testing.T, name string) *raftNode {
    nc := make([]*raftNode, 0, len(c.nodes))
    var nodeToRemove *raftNode
    for _, rn := range c.nodes {
        if rn.name == name {
            nodeToRemove = rn
        } else {
            nc = append(nc, rn)
        }
    }
    if nodeToRemove == nil {
        t.Fatalf("Unable to find node with name '%v' in cluster", name)
    }
    c.log.Logf("Removing node %v from cluster", name)
    c.Leader(time.Minute).raft.RemovePeer(raft.ServerAddress(name)).Error()
    c.nodes = nc
    c.removedNodes = append(c.removedNodes, nodeToRemove)
    return nodeToRemove
}

// Leader returns the node that is currently the Leader, if there is no
// leader this function blocks until a leader is elected (or a timeout occurs)
func (c *cluster) Leader(timeout time.Duration) *raftNode {
    start := time.Now()
    for true {
        for _, n := range c.nodes {
            if n.raft.State() == raft.Leader {
                return n
            }
        }
        if time.Now().Sub(start) > timeout {
            return nil
        }
        time.Sleep(time.Millisecond)
    }
    return nil
}

// containsNode returns true if the slice 'nodes' contains 'n'
func containsNode(nodes []*raftNode, n *raftNode) bool {
    for _, rn := range nodes {
        if rn == n {
            return true
        }
    }
    return false
}

// LeaderPlus returns the leader + n additional nodes from the cluster
```

```
// the leader is always the first node in the returned slice.
func (c *cluster) LeaderPlus(n int) []*raftNode {
    r := make([]*raftNode, 0, n+1)
    ldr := c.Leader(time.Second)
    if ldr != nil {
        r = append(r, ldr)
    }
    if len(r) >= n {
        return r
    }
    for _, node := range c.nodes {
        if !containsNode(r, node) {
            r = append(r, node)
            if len(r) >= n {
                return r
            }
        }
    }
    return r
}

func (c *cluster) Stop(t *testing.T, maxWait time.Duration) {
    c.WaitTilUptoDate(t, maxWait)
    for _, n := range c.nodes {
        n.raft.Shutdown()
    }
}

// WaitTilUptoDate blocks until all nodes in the cluster have gotten their
// commitedIndex upto the Index from the last successful call to Apply
func (c *cluster) WaitTilUptoDate(t *testing.T, maxWait time.Duration) {
    idx := c.lastApplySuccess.Index()
    start := time.Now()
    for true {
        allAtIdx := true
        for i := 0; i < len(c.nodes); i++ {
            nodeAppliedIdx := c.nodes[i].raft.AppliedIndex()
            if nodeAppliedIdx < idx {
                allAtIdx = false
                break
            } else if nodeAppliedIdx > idx {
                allAtIdx = false
                idx = nodeAppliedIdx
                break
            }
        }
        if allAtIdx {
            t.Logf("All nodes have appliedIndex=%d", idx)
            return
        }
        if time.Now().Sub(start) > maxWait {
            t.Fatalf("Gave up waiting for all nodes to reach raft Index %d,
[currently at %v]", idx, c.appliedIndexes())
```

```
        }
        time.Sleep(time.Millisecond * 10)
    }
}

func (c *cluster) appliedIndexes() map[string]uint64 {
    r := make(map[string]uint64, len(c.nodes))
    for _, n := range c.nodes {
        r[n.name] = n.raft.AppliedIndex()
    }
    return r
}

func (c *cluster) generateNApplies(s *applySource, n uint) [][]byte {
    data := make([][]byte, n)
    for i := uint(0); i < n; i++ {
        data[i] = s.nextEntry()
    }
    return data
}

func (c *cluster) leadershipTransfer(leaderTimeout time.Duration) raft.Future {
    ldr := c.Leader(leaderTimeout)
    return ldr.raft.LeadershipTransfer()
}

type applyFutureWithData struct {
    future raft.ApplyFuture
    data   []byte
}

func (c *cluster) sendNApplies(leaderTimeout time.Duration, data [][]byte)
[]applyFutureWithData {
    f := []applyFutureWithData{}

    ldr := c.Leader(leaderTimeout)
    if ldr != nil {
        for _, d := range data {
            f = append(f, applyFutureWithData{future: ldr.raft.Apply(d,
time.Second), data: d})
        }
    }
    return f
}

func (c *cluster) checkApplyFutures(futures []applyFutureWithData) uint64 {
    success := uint64(0)
    for _, a := range futures {
        if err := a.future.Error(); err == nil {
            success++
            c.lastApplySuccess = a.future
            c.applied = append(c.applied, appliedItem{a.future.Index(),
a.data})
```

```
        } else {
            c.lastApplyFailure = a.future
        }
    }
    return success
}

func (c *cluster) ApplyN(t *testing.T, leaderTimeout time.Duration, s
*applySource, n uint) uint64 {
    data := c.generateNApplies(s, n)
    futures := c.sendNApplies(leaderTimeout, data)
    return c.checkApplyFutures(futures)
}

func (c *cluster) VerifyFSM(t *testing.T) {
    exp := c.nodes[0].fsm
    expName := c.nodes[0].name
    for i, n := range c.nodes {
        if i > 0 {
            if exp.lastIndex != n.fsm.lastIndex {
                t.Errorf("Node %v FSM lastIndex is %d, but Node %v FSM
lastIndex is %d", n.name, n.fsm.lastIndex, expName, exp.lastIndex)
            }
            if exp.lastTerm != n.fsm.lastTerm {
                t.Errorf("Node %v FSM lastTerm is %d, but Node %v FSM lastTerm
is %d", n.name, n.fsm.lastTerm, expName, exp.lastTerm)
            }
            if !bytes.Equal(exp.lastHash, n.fsm.lastHash) {
                t.Errorf("Node %v FSM lastHash is %v, but Node %v FSM lastHash
is %v", n.name, n.fsm.lastHash, expName, exp.lastHash)
            }
        }
        t.Logf("node %v final FSM hash is %v", n.name, n.fsm.lastHash)
    }
    if t.Failed() {
        c.RecordState(t)
    }
}

func (c *cluster) RecordState(t *testing.T) {
    td, _ := ioutil.TempDir(os.Getenv("TEST_FAIL_DIR"), "failure")
    sd, _ := resolveDirectory("data", false)
    copyDir(td, sd)
    dump := func(n *raftNode) {
        nt := filepath.Join(td, n.name)
        os.Mkdir(nt, 0777)
        n.fsm.WriteTo(filepath.Join(nt, "fsm.txt"))
        n.transport.DumpLog(nt)
    }
    for _, n := range c.nodes {
        dump(n)
    }
    for _, n := range c.removedNodes {
```

```
            dump(n)
        }
        fmt.Printf("State of failing cluster captured in %v", td)
    }

    func copyDir(target, src string) {
        filepath.Walk(src, func(path string, info os.FileInfo, err error) error {
            relPath := path[len(src):]
            if info.IsDir() {
                return os.MkdirAll(filepath.Join(target, relPath), 0777)
            }
            return copyFile(filepath.Join(target, relPath), path)
        })
    }

    func copyFile(target, src string) error {
        r, err := os.Open(src)
        if err != nil {
            return err
        }
        defer r.Close()
        w, err := os.Create(target)
        if err != nil {
            return err
        }
        defer w.Close()
        _, err = io.Copy(w, r)
        return err
    }

    func (c *cluster) VerifyLog(t *testing.T, applyCount uint64) {
        fi, _ := c.nodes[0].store.FirstIndex()
        li, _ := c.nodes[0].store.LastIndex()
        name := c.nodes[0].name
        for _, n := range c.nodes {
            nfi, err := n.store.FirstIndex()
            if err != nil {
                t.Errorf("Failed to get FirstIndex of log for node %v: %v", n.name,
    err)
                continue
            }
            if nfi != fi {
                t.Errorf("Node %v has FirstIndex of %d but node %v has %d", n.name,
    nfi, name, fi)
            }
            nli, err := n.store.LastIndex()
            if err != nil {
                t.Errorf("Failed to get LastIndex of log for node %v: %v", n.name,
    err)
                continue
            }
            if nli != li {
                t.Errorf("Node %v has LastIndex of %d, but node %v has %d", n.name,
```

```
nli, name, li)
        }
        if nli-nfi < applyCount {
            t.Errorf("Node %v Log contains %d entries, but should contain at
least %d", n.name, nli-nfi, applyCount)
            continue
        }
        var term uint64
        for i := fi; i <= li; i++ {
            var nEntry raft.Log
            var n0Entry raft.Log
            if err := c.nodes[0].store.GetLog(i, &n0Entry); err != nil {
                t.Errorf("Failed to log entry %d on node %v: %v", i, name, err)
                continue
            }
            if err := n.store.GetLog(i, &nEntry); err != nil {
                t.Errorf("Failed to log entry at log Index %d on node %v: %v",
i, n.name, err)
                continue
            }
            if i != nEntry.Index {
                t.Errorf("Asked for Log Index %d from Store on node %v, but got
index %d instead", i, n.name, nEntry.Index)
            }
            if i == fi {
                term = nEntry.Term
            } else {
                if nEntry.Term < term {
                    t.Errorf("Node %v, Prior Log Entry was for term %d, but
this log entry is for term %d, terms shouldn't go backwards", n.name, term,
nEntry.Term)
                }
            }
            term = nEntry.Term
            assertLogEntryEqual(t, n.name, &n0Entry, &nEntry)
        }
        // the above checks the logs between the nodes, also check that the log
        // contains the items that Apply returned success for.
        var entry raft.Log
        for _, ai := range c.applied {
            err := n.store.GetLog(ai.index, &entry)
            if err != nil {
                t.Errorf("Failed to fetch logIndex %d on node %v: %v",
ai.index, n.name, err)
            }
            if !bytes.Equal(ai.data, entry.Data) {
                t.Errorf("Client applied %v at index %d, but log for node %v
contains %d", ai.data, ai.index, n.name, entry.Data)
            }
        }
    }
}
```

```
// assertLogEntryEqual compares the 2 raft Log entries and reports any
differences to the supplied testing.T instance
// it return true if the 2 entries are equal, false otherwise.
func assertLogEntryEqual(t *testing.T, node string, exp *raft.Log, act
*raft.Log) bool {
    res := true
    if exp.Term != act.Term {
        t.Errorf("Log Entry at Index %d for node %v has mismatched terms
%d/%d", exp.Index, node, exp.Term, act.Term)
        res = false
    }
    if exp.Index != act.Index {
        t.Errorf("Node %v, Log Entry should be Index %d,but is %d", node,
exp.Index, act.Index)
        res = false
    }
    if exp.Type != act.Type {
        t.Errorf("Node %v, Log Entry at Index %d should have type %v but is
%v", node, exp.Index, exp.Type, act.Type)
        res = false
    }
    if !bytes.Equal(exp.Data, act.Data) {
        t.Errorf("Node %v, Log Entry at Index %d should have data %v, but has
%v", node, exp.Index, exp.Data, act.Data)
        res = false
    }
    return res
}
```

../raft/fuzzy/fsm.go

```
package fuzzy

import (
    "bufio"
    "encoding/binary"
    "fmt"
    "hash/adler32"
    "io"
    "os"

    "github.com/hashicorp/raft"
)

type logHash struct {
    lastHash []byte
}

func (l *logHash) Add(d []byte) {
    hasher := adler32.New()
    hasher.Write(l.lastHash)
    hasher.Write(d)
    l.lastHash = hasher.Sum(nil)
}

type applyItem struct {
    index uint64
    term  uint64
    data  []byte
}

func (a *applyItem) set(l *raft.Log) {
    a.index = l.Index
    a.term = l.Term
    a.data = make([]byte, len(l.Data))
    copy(a.data, l.Data)
}

type fuzzyFSM struct {
    logHash
    lastTerm  uint64
    lastIndex uint64
    applied   []applyItem
}

func (f *fuzzyFSM) Apply(l *raft.Log) interface{} {
    if l.Index <= f.lastIndex {
        panic(fmt.Errorf("fsm.Apply received log entry with invalid Index %v
(lastIndex we saw was %d)", l, f.lastIndex))
    }
    if l.Term < f.lastTerm {
        panic(fmt.Errorf("fsm.Apply received log entry with invalid Term %v
(lastTerm we saw was %d)", l, f.lastTerm))
```

```
    }
    f.lastIndex = l.Index
    f.lastTerm = l.Term
    f.Add(l.Data)
    f.applied = append(f.applied, applyItem{})
    f.applied[len(f.applied)-1].set(l)
    return nil
}

func (f *fuzzyFSM) WriteTo(fn string) error {
    fw, err := os.Create(fn)
    if err != nil {
        return err
    }
    defer fw.Close()
    w := bufio.NewWriter(fw)
    defer w.Flush()
    for _, i := range f.applied {
        fmt.Fprintf(w, "%d.%8d: %X\n", i.term, i.index, i.data)
    }
    return nil
}

func (f *fuzzyFSM) Snapshot() (raft.FSMSnapshot, error) {
    s := *f
    return &s, nil
}

func (f *fuzzyFSM) Restore(r io.ReadCloser) error {
    err := binary.Read(r, binary.LittleEndian, &f.lastTerm)
    if err == nil {
        err = binary.Read(r, binary.LittleEndian, &f.lastIndex)
    }
    if err == nil {
        f.lastHash = make([]byte, adler32.Size)
        _, err = r.Read(f.lastHash)
    }
    return err
}

func (f *fuzzyFSM) Persist(sink raft.SnapshotSink) error {
    err := binary.Write(sink, binary.LittleEndian, f.lastTerm)
    if err == nil {
        err = binary.Write(sink, binary.LittleEndian, f.lastIndex)
    }
    if err == nil {
        _, err = sink.Write(f.lastHash)
    }
    return err
}

func (f *fuzzyFSM) Release() {
```

```
}
```

../raft/fuzzy/fsm_batch.go

```
// +build batchtest

package fuzzy

import "github.com/hashicorp/raft"

// ApplyBatch enables fuzzyFSM to satisfy the BatchingFSM interface. This
// function is gated by the batchtest build flag.
func (f *fuzzyFSM) ApplyBatch(logs []*raft.Log) []interface{} {
    ret := make([]interface{}, len(logs))

    for _, l := range logs {
        f.Apply(l)
    }

    return ret
}
```

../raft/fuzzy/leadershiptransfer_test.go

```
package fuzzy

import (
    "math/rand"
    "testing"
    "time"

    "github.com/hashicorp/raft"
)

// 5 node cluster
func TestRaft_FuzzyLeadershipTransfer(t *testing.T) {
    cluster := newRaftCluster(t, testLogWriter, "lt", 5, nil)
    r := rand.New(rand.NewSource(time.Now().UnixNano()))

    s := newApplySource("LeadershipTransfer")
    data := cluster.generateNApplies(s, uint(r.Intn(10000)))
    futures := cluster.sendNApplies(time.Minute, data)
    cluster.leadershipTransfer(time.Minute)

    data = cluster.generateNApplies(s, uint(r.Intn(10000)))
    futures = append(futures, cluster.sendNApplies(time.Minute, data)...)
    cluster.leadershipTransfer(time.Minute)

    data = cluster.generateNApplies(s, uint(r.Intn(10000)))
    futures = append(futures, cluster.sendNApplies(time.Minute, data)...)
    cluster.leadershipTransfer(time.Minute)

    data = cluster.generateNApplies(s, uint(r.Intn(10000)))
    futures = append(futures, cluster.sendNApplies(time.Minute, data)...)

    ac := cluster.checkApplyFutures(futures)

    cluster.Stop(t, time.Minute)
    cluster.VerifyLog(t, ac)
    cluster.VerifyFSM(t)
}

type LeadershipTransferMode int

type LeadershipTransfer struct {
    verifier  appendEntriesVerifier
    slowNodes map[string]bool
    delayMin  time.Duration
    delayMax  time.Duration
    mode      LeadershipTransferMode
}

func (lt *LeadershipTransfer) Report(t *testing.T) {
    lt.verifier.Report(t)
}
```

```
func (lt *LeadershipTransfer) PreRPC(s, t string, r *raft.RPC) error {
    return nil
}

func (lt *LeadershipTransfer) nap() {
    d := lt.delayMin + time.Duration(rand.Int63n((lt.delayMax -
lt.delayMin).Nanoseconds()))
    time.Sleep(d)
}

func (lt *LeadershipTransfer) PostRPC(src, target string, r *raft.RPC, res
*raft.RPCResponse) error {
    return nil
}

func (lt *LeadershipTransfer) PreRequestVote(src, target string, v
*raft.RequestVoteRequest) (*raft.RequestVoteResponse, error) {
    return nil, nil
}

func (lt *LeadershipTransfer) PreAppendEntries(src, target string, v
*raft.AppendEntriesRequest) (*raft.AppendEntriesResponse, error) {
    lt.verifier.PreAppendEntries(src, target, v)
    return nil, nil
}
```

../raft/fuzzy/membership_test.go

```
package fuzzy

import (
    "io"
    "log"
    "os"
    "path/filepath"
    "testing"
    "time"
)

var testLogWriter io.Writer

func init() {
    testLogWriter = os.Stdout
    logDir := os.Getenv("TEST_LOG_DIR")
    if logDir != "" {
        f, err := os.Create(filepath.Join(logDir, "debug.log"))
        if err != nil {
            log.Fatalf("TEST_LOG_DIR Env set, but unable to create log file:
%v\n", err)
        }
        testLogWriter = f
    }
}

// this runs a 3 node cluster then expands it to a 5 node cluster and checks
all 5 nodes agree at the end
func TestRaft_AddMembership(t *testing.T) {
    v := appendEntriesVerifier{}
    v.Init()
    cluster := newRaftCluster(t, testLogWriter, "m", 3, &v)
    s := newApplySource("AddMembership")
    initApplied := cluster.ApplyN(t, time.Minute, s, 100)
    a := s.apply(t, cluster, 1000)
    if err := cluster.CreateAndAddNode(t, testLogWriter, "m", 3); err != nil {
        t.Fatalf("Failed to add node m3: %v", err)
    }
    if err := cluster.CreateAndAddNode(t, testLogWriter, "m", 4); err != nil {
        t.Fatalf("Failed to add node m4: %v", err)
    }
    time.Sleep(time.Second * 5)
    a.stop()
    cluster.Stop(t, time.Minute)
    v.Report(t)
    cluster.VerifyLog(t, uint64(a.applied+initApplied))
    cluster.VerifyFSM(t)
}

// starts with 3 nodes, goes to 5, then goes back to 3, but never removes the
leader.
func TestRaft_AddRemoveNodesNotLeader(t *testing.T) {
```

```
    v := appendEntriesVerifier{}
    v.Init()
    cluster := newRaftCluster(t, testLogWriter, "ar", 3, &v)
    s := newApplySource("AddRemoveNodesNotLeader")
    initApplied := cluster.ApplyN(t, time.Minute, s, 100)
    a := s.apply(t, cluster, 1000)
    cluster.CreateAndAddNode(t, testLogWriter, "ar", 3)
    cluster.CreateAndAddNode(t, testLogWriter, "ar", 4)
    ldr := cluster.Leader(time.Minute)
    removed := 0
    for _, rn := range cluster.nodes {
        if rn.name != ldr.name {
            cluster.RemoveNode(t, rn.name)
            removed++
            if removed >= 2 {
                break
            }
        }
    }
    a.stop()
    cluster.Stop(t, time.Minute)
    v.Report(t)
    cluster.VerifyLog(t, uint64(a.applied+initApplied))
    cluster.VerifyFSM(t)
}

// starts with a 5 node cluster then removes the leader.
func TestRaft_RemoveLeader(t *testing.T) {
    v := appendEntriesVerifier{}
    v.Init()
    cluster := newRaftCluster(t, testLogWriter, "rl", 5, &v)
    s := newApplySource("RemoveLeader")
    initApplied := cluster.ApplyN(t, time.Minute, s, 100)
    a := s.apply(t, cluster, 100)
    time.Sleep(time.Second)
    ldr := cluster.Leader(time.Minute)
    cluster.RemoveNode(t, ldr.name)
    time.Sleep(5 * time.Second)
    a.stop()
    cluster.Stop(t, time.Minute)
    v.Report(t)
    cluster.VerifyLog(t, uint64(a.applied+initApplied))
    cluster.VerifyFSM(t)
    ldr.raft.Shutdown()
}

// starts with a 5 node cluster, partitions off one node, and then removes it
from the cluster on the other partition
func TestRaft_RemovePartitionedNode(t *testing.T) {
    hooks := NewPartitioner()
    cluster := newRaftCluster(t, testLogWriter, "rmp", 5, hooks)
    s := newApplySource("RemovePartitionedNode")
    initApplied := cluster.ApplyN(t, time.Minute, s, 101)
```

```
    a := s.apply(t, cluster, 100)
    nodes := cluster.LeaderPlus(3)
    victim := nodes[len(nodes)-1]
    hooks.PartitionOff(cluster.log, []*raftNode{victim})
    time.Sleep(3 * time.Second)
    removed := cluster.RemoveNode(t, victim.name)
    time.Sleep(3 * time.Second)
    hooks.HealAll(cluster.log)
    time.Sleep(10 * time.Second)
    a.stop()
    cluster.Stop(t, time.Minute)
    hooks.Report(t)
    cluster.VerifyLog(t, uint64(a.applied+initApplied))
    cluster.VerifyFSM(t)

    // we should verify that the partitioned node see that it was removed &
shutdown
    // but it never gets notified of that, so we can't verify that currently.
    removed.raft.Shutdown()
}
```

../raft/fuzzy/node.go

```
package fuzzy

import (
    "fmt"
    "path/filepath"
    "time"

    "github.com/hashicorp/go-hclog"

    "github.com/hashicorp/raft"
    rdb "github.com/hashicorp/raft-boltdb"
)

type raftNode struct {
    transport *transport
    store     *rdb.BoltStore
    raft      *raft.Raft
    log       hclog.Logger
    fsm       *fuzzyFSM
    name      string
    dir       string
}

func newRaftNode(logger *log.Logger, tc *transports, h TransportHooks, nodes
[]string, name string) (*raftNode, error) {
    var err error
    var datadir string
    datadir, err = resolveDirectory(fmt.Sprintf("data/%v", name), true)
    if err != nil {
        return nil, err
    }
    logger.Printf("[INFO] Creating new raft Node with data in dir %v", datadir)
    var ss *raft.FileSnapshotStore
    ss, err = raft.NewFileSnapshotStoreWithLogger(datadir, 5, logger)

    if err != nil {
        return nil, fmt.Errorf("unable to initialize snapshots %v",
err.Error())
    }
    transport := tc.AddNode(name, h)

    config := raft.DefaultConfig()
    config.SnapshotThreshold = 1409600
    config.SnapshotInterval = time.Hour
    config.Logger = logger
    config.ShutdownOnRemove = false
    config.LocalID = raft.ServerID(name)

    var store *rdb.BoltStore
    store, err = rdb.NewBoltStore(filepath.Join(datadir, "store.bolt"))
    if err != nil {
        return nil, fmt.Errorf("unable to initialize log %v", err.Error())
```

```
    }

    if len(nodes) > 0 {
        c := make([]raft.Server, 0, len(nodes))
        for _, n := range nodes {
            c = append(c, raft.Server{Suffrage: raft.Voter, ID:
raft.ServerID(n), Address: raft.ServerAddress(n)})
        }
        configuration := raft.Configuration{Servers: c}

        if err = raft.BootstrapCluster(config, store, store, ss, transport,
configuration); err != nil {
            return nil, err
        }
    }
    fsm := &fuzzyFSM{}
    var r *raft.Raft
    r, err = raft.NewRaft(config, fsm, store, store, ss, transport)
    if err != nil {
        return nil, err
    }
    n := raftNode{
        transport: transport,
        store:     store,
        raft:      r,
        fsm:       fsm,
        log:       logger,
        name:      name,
        dir:       datadir,
    }
    return &n, nil
}
```

../raft/fuzzy/partition_test.go

```
package fuzzy

import (
    "bytes"
    "fmt"
    "math/rand"
    "sync"
    "testing"
    "time"

    "github.com/hashicorp/raft"
)

// 5 node cluster where the leader and another node get regularly partitioned
off
// eventually all partitions heal.
func TestRaft_LeaderPartitions(t *testing.T) {
    hooks := NewPartitioner()
    cluster := newRaftCluster(t, testLogWriter, "lp", 5, hooks)
    cluster.Leader(time.Second * 10)
    s := newApplySource("LeaderPartitions")
    applier := s.apply(t, cluster, 5)
    for i := 0; i < 10; i++ {
        pg := hooks.PartitionOff(cluster.log, cluster.LeaderPlus(rand.Intn(4)))
        time.Sleep(time.Second * 4)
        r := rand.Intn(10)
        if r < 1 {
            cluster.log.Logf("Healing no partitions!")
        } else if r < 4 {
            hooks.HealAll(cluster.log)
        } else {
            hooks.Heal(cluster.log, pg)
        }
        time.Sleep(time.Second * 5)
    }
    hooks.HealAll(cluster.log)
    cluster.Leader(time.Hour)
    applier.stop()
    cluster.Stop(t, time.Minute*10)
    hooks.Report(t)
    cluster.VerifyLog(t, applier.applied)
    cluster.VerifyFSM(t)
}

type Partitioner struct {
    verifier appendEntriesVerifier
    lock     sync.RWMutex // protects partitioned / nextGroup
    // this is a map of node -> partition group, only nodes in the same
partition group can communicate with each other
    partitioned map[string]int
    nextGroup   int
}
```

```
func NewPartitioner() *Partitioner {
    p := &Partitioner{
        partitioned: make(map[string]int),
        nextGroup:   1,
    }
    p.verifier.Init()
    return p
}

// PartitionOff creates a partition where the supplied nodes can only
communicate with each other
// returns the partition group, which can be used later with Heal to heal this
specific partition
func (p *Partitioner) PartitionOff(l Logger, nodes []*raftNode) int {
    nn := make([]string, 0, len(nodes))
    p.lock.Lock()
    defer p.lock.Unlock()
    pGroup := p.nextGroup
    p.nextGroup++
    for _, n := range nodes {
        p.partitioned[n.name] = pGroup
        nn = append(nn, n.name)
    }
    l.Logf("Created partition %d with nodes %v, partitions now are %v", pGroup,
nn, p)
    return pGroup
}

func (p *Partitioner) Heal(l Logger, pGroup int) {
    p.lock.Lock()
    defer p.lock.Unlock()
    for k, v := range p.partitioned {
        if v == pGroup {
            p.partitioned[k] = 0
        }
    }
    l.Logf("Healing partition group %d, now partitions are %v", pGroup, p)
}

func (p *Partitioner) String() string {
    pl := make([][]string, 0, 10)
    for n, pv := range p.partitioned {
        if pv > 0 {
            for pv >= len(pl) {
                pl = append(pl, nil)
            }
            pl[pv] = append(pl[pv], n)
        }
    }
    b := bytes.Buffer{}
    for i, n := range pl {
        if len(n) > 0 {
```

```
            if b.Len() > 0 {
                b.WriteString(", ")
            }
            fmt.Fprintf(&b, "%d = %v", i, n)
        }
    }
    if b.Len() == 0 {
        return "[None]"
    }
    return b.String()
}

func (p *Partitioner) HealAll(l Logger) {
    p.lock.Lock()
    defer p.lock.Unlock()
    p.partitioned = make(map[string]int)
    l.Logf("Healing all partitions, partitions now %v", p)
}

func (p *Partitioner) Report(t *testing.T) {
    p.verifier.Report(t)
}

func (p *Partitioner) PreRPC(s, t string, r *raft.RPC) error {
    p.lock.RLock()
    sp := p.partitioned[s]
    st := p.partitioned[t]
    p.lock.RUnlock()
    if sp == st {
        return nil
    }
    return fmt.Errorf("Unable to connect to %v, from %v", t, s)
}

func (p *Partitioner) PostRPC(s, t string, req *raft.RPC, res
*raft.RPCResponse) error {
    return nil
}

func (p *Partitioner) PreRequestVote(src, target string, v
*raft.RequestVoteRequest) (*raft.RequestVoteResponse, error) {
    return nil, nil
}

func (p *Partitioner) PreAppendEntries(src, target string, v
*raft.AppendEntriesRequest) (*raft.AppendEntriesResponse, error) {
    return nil, nil
}
```

../raft/fuzzy/readme.md

Fuzzy Raft

Inspired by http://colin-scott.github.io/blog/2015/10/07/fuzzing-raft-for-fun-and-profit/ this package
is a framework and set of test scenarios for testing the behavior and correctness of the raft library
under various conditions.

Framework

The framework allows you to construct multiple node raft clusters, connected by an instrumented transport
that allows a test to inject various transport level behaviors to simulate various scenarios (e.g. you
can have your hook fail all transport calls to a particular node to simulate it being partitioned off
the network). There are helper classes to create and Apply well know sequences of test data, and to
examine the final state of the cluster, the nodes FSMs and the raft log.

Running

The tests run with the standard go test framework, run with go test . [from this dir] or use make fuzz from
the parent directory. As these tests are looking for timing and other edge cases, a pass from a single run
isn't enough, the tests needs running repeatedly to build up confidence.

Test Scenarios

The follow test scenario's are currently implemented. Each test concludes with a standard set of validations

- Each node raft log contains the same set of entries (term/index/data).
- The raft log contains data matching the client request for each call to raft.Apply() that reported success.
- Each node's FSM saw the same sequence of Apply(*raft.Log) calls.
- A verifier at the transport level verifies a number of transport level invariants.

Most tests run with a background workload that is constantly apply()ing new entries to the log. [when there's a leader]

TestRaft_LeaderPartitions

This creates a 5 node cluster and then repeated partitions multiple nodes off (including the current leader),
then heals the partition and repeats. At the end all partitions are removed. [clearly inspired by Jepson]

TestRaft_NoIssueSanity

Is a basic 5 node cluster test, it starts a 5 node cluster applies some data, then does the verifications

TestRaft_SlowSendVote

Tests what happens when RequestVote requests are delaying being sent to other nodes

TestRaft_SlowRecvVote

Tests what happens when RequestVote responses are delaying being received by the sender.

TestRaft_AddMembership

Starts a 3 node cluster, and then adds 2 new members to the cluster.

TestRaft_AddRemoveNodesNotLeader

Starts a 5 node cluster, and then then removes 2 follower nodes from the cluster.

TestRaft_RemoveLeader

Starts a 5 node cluster, and then removes the node that is the leader.

TestRaft_RemovePartitionedNode

Starts a 5 node cluster, partitions one of the follower nodes off the network, and then tells the leader to remove that node, then heals the partition.

../raft/fuzzy/resolve.go

```go
package fuzzy

import (
    "os"
    "path/filepath"
)

// resolveDirectory returns a full directory path based on the supplied dir
path
// if the supplied dir path is absolute (i.e. it starts with / ) then it is
// returned as is, if it's a relative path, then it is assumed to be relative
// to the executable, and that is computed and returned.
//
// if create is true, then the directory path will be created if it doesn't
// already exist
//
// if create is false, then it's upto the caller to ensure it exists and/or
// create it as needed [this won't verify that it exists]
func resolveDirectory(dir string, create bool) (string, error) {
    var resolved string
    if filepath.IsAbs(dir) {
        resolved = dir
    } else {
        execdir, err := filepath.Abs(filepath.Dir(os.Args[0]))
        if err != nil {
            return "", err
        }
        resolved = filepath.Join(execdir, dir)
    }
    if create {
        if _, err := os.Stat(resolved); os.IsNotExist(err) {
            if err := os.MkdirAll(resolved, 0744); err != nil {
                return "", err
            }
        }
    }
    return resolved, nil
}
```

../raft/fuzzy/simple_test.go

```go
package fuzzy

import (
    "testing"
    "time"
)

// this runs a 5 node cluster with verifications turned on, but no failures or
issues injected.
func TestRaft_NoIssueSanity(t *testing.T) {
    v := appendEntriesVerifier{}
    v.Init()
    cluster := newRaftCluster(t, testLogWriter, "node", 5, &v)
    s := newApplySource("NoIssueSanity")
    applyCount := cluster.ApplyN(t, time.Minute, s, 10000)
    cluster.Stop(t, time.Minute)
    v.Report(t)
    cluster.VerifyLog(t, applyCount)
    cluster.VerifyFSM(t)
}
```

../raft/fuzzy/slowvoter_test.go

```
package fuzzy

import (
    "math/rand"
    "testing"
    "time"

    "github.com/hashicorp/raft"
)

// 5 node cluster where 2 nodes always see a delay in getting a request vote
msg.
func TestRaft_SlowSendVote(t *testing.T) {
    hooks := NewSlowVoter("sv_0", "sv_1")
    cluster := newRaftCluster(t, testLogWriter, "sv", 5, hooks)
    s := newApplySource("SlowSendVote")
    ac := cluster.ApplyN(t, time.Minute, s, 10000)
    cluster.Stop(t, time.Minute)
    hooks.Report(t)
    cluster.VerifyLog(t, ac)
    cluster.VerifyFSM(t)
}

// 5 node cluster where vote results from 3 nodes are slow to turn up.
// [they see the vote request normally, but their response is slow]
func TestRaft_SlowRecvVote(t *testing.T) {
    hooks := NewSlowVoter("svr_1", "svr_4", "svr_3")
    hooks.mode = SlowRecv
    cluster := newRaftCluster(t, testLogWriter, "svr", 5, hooks)
    s := newApplySource("SlowRecvVote")
    ac := cluster.ApplyN(t, time.Minute, s, 10000)
    cluster.Stop(t, time.Minute)
    hooks.Report(t)
    cluster.VerifyLog(t, ac)
    cluster.VerifyFSM(t)
}

type SlowVoterMode int

const (
    SlowSend SlowVoterMode = iota
    SlowRecv
)

type SlowVoter struct {
    verifier  appendEntriesVerifier
    slowNodes map[string]bool
    delayMin  time.Duration
    delayMax  time.Duration
    mode      SlowVoterMode
}
```

```
func NewSlowVoter(slowNodes ...string) *SlowVoter {
    sv := SlowVoter{
        slowNodes: make(map[string]bool, len(slowNodes)),
        delayMin:  time.Second,
        delayMax:  time.Second * 2,
        mode:      SlowSend,
    }
    for _, n := range slowNodes {
        sv.slowNodes[n] = true
    }
    sv.verifier.Init()
    return &sv
}

func (sv *SlowVoter) Report(t *testing.T) {
    sv.verifier.Report(t)
}

func (sv *SlowVoter) PreRPC(s, t string, r *raft.RPC) error {
    return nil
}

func (sv *SlowVoter) nap() {
    d := sv.delayMin + time.Duration(rand.Int63n((sv.delayMax -
sv.delayMin).Nanoseconds()))
    time.Sleep(d)
}

func (sv *SlowVoter) PostRPC(src, target string, r *raft.RPC, res
*raft.RPCResponse) error {
    if sv.mode == SlowRecv && sv.slowNodes[target] {
        _, ok := r.Command.(*raft.RequestVoteRequest)
        if ok {
            sv.nap()
        }
    }
    return nil
}

func (sv *SlowVoter) PreRequestVote(src, target string, v
*raft.RequestVoteRequest) (*raft.RequestVoteResponse, error) {
    if sv.mode == SlowSend && sv.slowNodes[target] {
        sv.nap()
    }
    return nil, nil
}

func (sv *SlowVoter) PreAppendEntries(src, target string, v
*raft.AppendEntriesRequest) (*raft.AppendEntriesResponse, error) {
    sv.verifier.PreAppendEntries(src, target, v)
    return nil, nil
}
```

../raft/fuzzy/transport.go

```
package fuzzy

import (
    "bufio"
    "bytes"
    "errors"
    "fmt"
    "github.com/hashicorp/go-hclog"
    "io"
    "os"
    "path/filepath"
    "sync"
    "time"

    "github.com/hashicorp/go-msgpack/codec"
    "github.com/hashicorp/raft"
)

var (
    codecHandle codec.MsgpackHandle
)

type appendEntries struct {
    source      string
    target      raft.ServerAddress
    term        uint64
    firstIndex  uint64
    lastIndex   uint64
    commitIndex uint64
}

type transports struct {
    sync.RWMutex
    nodes map[string]*transport
    log   hclog.Logger
}

func newTransports(log hclog.Logger) *transports {
    return &transports{
        nodes: make(map[string]*transport),
        log:   log,
    }
}

func (tc *transports) AddNode(n string, hooks TransportHooks) *transport {
    t := newTransport(n, tc, hooks)
    t.log = tc.log
    tc.Lock()
    defer tc.Unlock()
    tc.nodes[n] = t
    return t
}
```

```
// TransportHooks allow a test to customize the behavior of the transport.
// [if you return an error from a PreXXX call, then the error is returned to
the caller, and the RPC never made]
type TransportHooks interface {
    // PreRPC is called before every single RPC call from the transport
    PreRPC(src, target string, r *raft.RPC) error
    // PostRPC is called after the RPC call has been processed by the target,
but before the source see's the response
    PostRPC(src, target string, r *raft.RPC, result *raft.RPCResponse) error
    // PreREquestVote is called before sending a RequestVote RPC request.
    PreRequestVote(src, target string, r *raft.RequestVoteRequest)
(*raft.RequestVoteResponse, error)
    // PreAppendEntries is called before sending an AppendEntries RPC request.
    PreAppendEntries(src, target string, r *raft.AppendEntriesRequest)
(*raft.AppendEntriesResponse, error)
}

type transport struct {
    log        hclog.Logger
    transports *transports
    node       string
    ae         []appendEntries

    consumer chan raft.RPC
    hooks    TransportHooks
}

func newTransport(node string, tc *transports, hooks TransportHooks) *transport
{
    return &transport{
        node:       node,
        transports: tc,
        hooks:      hooks,
        consumer:   make(chan raft.RPC),
        ae:         make([]appendEntries, 0, 50000),
    }
}

// Consumer returns a channel that can be used to
// consume and respond to RPC requests.
func (t *transport) Consumer() <-chan raft.RPC {
    return t.consumer
}

// LocalAddr is used to return our local address to distinguish from our peers.
func (t *transport) LocalAddr() raft.ServerAddress {
    return raft.ServerAddress(t.node)
}

func (t *transport) sendRPC(target string, req interface{}, resp interface{})
error {
    t.transports.RLock()
```

```
    tt := t.transports.nodes[target]
    if tt == nil {
        t.log.Info("sendRPC unknown node", "target", target, "transports",
t.transports.nodes)
        t.transports.RUnlock()
        return fmt.Errorf("Unknown target host %v", target)
    }
    t.transports.RUnlock()
    rc := make(chan raft.RPCResponse, 1)

    buff := bytes.Buffer{}
    if err := codec.NewEncoder(&buff, &codecHandle).Encode(req); err != nil {
        return err
    }
    rpc := raft.RPC{RespChan: rc}
    var reqVote raft.RequestVoteRequest
    var timeoutNow raft.TimeoutNowRequest
    var appEnt raft.AppendEntriesRequest
    dec := codec.NewDecoderBytes(buff.Bytes(), &codecHandle)
    switch req.(type) {
    case *raft.TimeoutNowRequest:
        if err := dec.Decode(&timeoutNow); err != nil {
            return err
        }
        rpc.Command = &timeoutNow
    case *raft.RequestVoteRequest:
        if err := dec.Decode(&reqVote); err != nil {
            return err
        }
        rpc.Command = &reqVote
    case *raft.AppendEntriesRequest:
        if err := dec.Decode(&appEnt); err != nil {
            return err
        }
        rpc.Command = &appEnt
    default:
        t.log.Warn("unexpected request type", "type", hclog.Fmt("%T", req),
"request", req)
    }
    var result *raft.RPCResponse
    if t.hooks != nil {
        if err := t.hooks.PreRPC(t.node, target, &rpc); err != nil {
            return err
        }
        switch req.(type) {
        case *raft.RequestVoteRequest:
            hr, err := t.hooks.PreRequestVote(t.node, target, &reqVote)
            if hr != nil || err != nil {
                result = &raft.RPCResponse{Response: hr, Error: err}
            }
        case *raft.AppendEntriesRequest:
            hr, err := t.hooks.PreAppendEntries(t.node, target, &appEnt)
            if hr != nil || err != nil {
```

```
                result = &raft.RPCResponse{Response: hr, Error: err}
            }
        }
    }
    if result == nil {
        tt.consumer <- rpc
        cr := <-rc
        result = &cr
    }

    if t.hooks != nil {
        err := t.hooks.PostRPC(t.node, target, &rpc, result)
        if err != nil {
            result.Error = err
        }
    }
    buff = bytes.Buffer{}
    codec.NewEncoder(&buff, &codecHandle).Encode(result.Response)
    codec.NewDecoderBytes(buff.Bytes(), &codecHandle).Decode(resp)
    return result.Error
}

// TimeoutNow implements the Transport interface.
func (t *transport) TimeoutNow(id raft.ServerID, target raft.ServerAddress,
args *raft.TimeoutNowRequest, resp *raft.TimeoutNowResponse) error {
    return t.sendRPC(string(target), args, resp)
}

// AppendEntries sends the appropriate RPC to the target node.
func (t *transport) AppendEntries(id raft.ServerID, target raft.ServerAddress,
args *raft.AppendEntriesRequest, resp *raft.AppendEntriesResponse) error {
    ae := appendEntries{
        source:      t.node,
        target:      target,
        firstIndex:  firstIndex(args),
        lastIndex:   lastIndex(args),
        commitIndex: args.LeaderCommitIndex,
    }
    if len(t.ae) < cap(t.ae) {
        t.ae = append(t.ae, ae)
    }
    return t.sendRPC(string(target), args, resp)
}

func (t *transport) DumpLog(dir string) {
    fw, _ := os.Create(filepath.Join(dir, t.node+".transport"))
    w := bufio.NewWriter(fw)
    for i := range t.ae {
        e := &t.ae[i]
        fmt.Fprintf(w, "%v -> %v\t%8d - %8d : %8d\n", e.source, e.target,
e.firstIndex, e.lastIndex, e.commitIndex)
    }
    w.Flush()
```

```
    fw.Close()
}

func firstIndex(a *raft.AppendEntriesRequest) uint64 {
    if len(a.Entries) == 0 {
        return 0
    }
    return a.Entries[0].Index
}

func lastIndex(a *raft.AppendEntriesRequest) uint64 {
    if len(a.Entries) == 0 {
        return 0
    }
    return a.Entries[len(a.Entries)-1].Index
}

// RequestVote sends the appropriate RPC to the target node.
func (t *transport) RequestVote(id raft.ServerID, target raft.ServerAddress,
args *raft.RequestVoteRequest, resp *raft.RequestVoteResponse) error {
    return t.sendRPC(string(target), args, resp)
}

// InstallSnapshot is used to push a snapshot down to a follower. The data is
read from
// the ReadCloser and streamed to the client.
func (t *transport) InstallSnapshot(id raft.ServerID, target
raft.ServerAddress, args *raft.InstallSnapshotRequest, resp
*raft.InstallSnapshotResponse, data io.Reader) error {
    t.log.Debug("INSTALL SNAPSHOT ***************************************")
    return errors.New("huh")
}

// EncodePeer is used to serialize a peer name.
func (t *transport) EncodePeer(id raft.ServerID, p raft.ServerAddress) []byte {
    return []byte(p)
}

// DecodePeer is used to deserialize a peer name.
func (t *transport) DecodePeer(p []byte) raft.ServerAddress {
    return raft.ServerAddress(p)
}

// SetHeartbeatHandler is used to setup a heartbeat handler
// as a fast-pass. This is to avoid head-of-line blocking from
// disk IO. If a Transport does not support this, it can simply
// ignore the call, and push the heartbeat onto the Consumer channel.
func (t *transport) SetHeartbeatHandler(cb func(rpc raft.RPC)) {
}

// AppendEntriesPipeline returns an interface that can be used to pipeline
// AppendEntries requests.
func (t *transport) AppendEntriesPipeline(id raft.ServerID, target
```

```
raft.ServerAddress) (raft.AppendPipeline, error) {
    p := &pipeline{
        t:        t,
        id:       id,
        target:   target,
        work:     make(chan *appendEntry, 100),
        consumer: make(chan raft.AppendFuture, 100),
    }
    go p.run()
    return p, nil
}

type appendEntry struct {
    req      *raft.AppendEntriesRequest
    res      *raft.AppendEntriesResponse
    start    time.Time
    err      error
    ready    chan error
    consumer chan raft.AppendFuture
}

func (e *appendEntry) Request() *raft.AppendEntriesRequest {
    return e.req
}

func (e *appendEntry) Response() *raft.AppendEntriesResponse {
    <-e.ready
    return e.res
}

func (e *appendEntry) Start() time.Time {
    return e.start
}

func (e *appendEntry) Error() error {
    <-e.ready
    return e.err
}

func (e *appendEntry) Respond(err error) {
    e.err = err
    close(e.ready)
    e.consumer <- e
}

type pipeline struct {
    t        *transport
    target   raft.ServerAddress
    id       raft.ServerID
    work     chan *appendEntry
    consumer chan raft.AppendFuture
}
```

```
func (p *pipeline) run() {
    for ap := range p.work {
        err := p.t.AppendEntries(p.id, p.target, ap.req, ap.res)
        ap.Respond(err)
    }
}

// AppendEntries is used to add another request to the pipeline.
// The send may block which is an effective form of back-pressure.
func (p *pipeline) AppendEntries(args *raft.AppendEntriesRequest, resp
*raft.AppendEntriesResponse) (raft.AppendFuture, error) {
    e := &appendEntry{
        req:      args,
        res:      resp,
        start:    time.Now(),
        ready:    make(chan error),
        consumer: p.consumer,
    }
    p.work <- e
    return e, nil
}

func (p *pipeline) Consumer() <-chan raft.AppendFuture {
    return p.consumer
}

// Closes pipeline and cancels all inflight RPCs
func (p *pipeline) Close() error {
    close(p.work)
    return nil
}
```

../raft/fuzzy/verifier.go

```
package fuzzy

import (
    "fmt"
    "sync"
    "testing"

    "github.com/hashicorp/raft"
)

// AppendEntriesVerifier looks at all the AppendEntry RPC request and verifies
that only one node sends AE requests for any given term
// it also verifies that the request only comes from the node indicated as the
leader in the AE message.
type appendEntriesVerifier struct {
    sync.RWMutex
    leaderForTerm map[uint64]string
    errors        []string
}

func (v *appendEntriesVerifier) Report(t *testing.T) {
    v.Lock()
    defer v.Unlock()
    for _, e := range v.errors {
        t.Error(e)
    }
}

func (v *appendEntriesVerifier) Init() {
    v.Lock()
    defer v.Unlock()
    v.leaderForTerm = make(map[uint64]string)
    v.errors = make([]string, 0, 10)
}

func (v *appendEntriesVerifier) PreRPC(src, target string, r *raft.RPC) error {
    return nil
}

func (v *appendEntriesVerifier) PostRPC(src, target string, req *raft.RPC, res
*raft.RPCResponse) error {
    return nil
}

func (v *appendEntriesVerifier) PreRequestVote(src, target string, rv
*raft.RequestVoteRequest) (*raft.RequestVoteResponse, error) {
    return nil, nil
}

func (v *appendEntriesVerifier) PreAppendEntries(src, target string, req
*raft.AppendEntriesRequest) (*raft.AppendEntriesResponse, error) {
    term := req.Term
```

```
    ldr := string(req.Leader)
    if ldr != src {
        v.Lock()
        defer v.Unlock()
        v.errors = append(v.errors, fmt.Sprintf("Node %v sent an appendEnties
request for term %d that said the leader was some other node %v", src, term,
ldr))
    }
    v.RLock()
    tl, exists := v.leaderForTerm[term]
    v.RUnlock()
    if exists && tl != ldr {
        v.Lock()
        defer v.Unlock()
        v.errors = append(v.errors, fmt.Sprintf("Node %v sent an AppendEntries
request for term %d, but node %v had already done some, multiple leaders for
same term!", src, term, tl))
    }
    if !exists {
        v.Lock()
        tl, exists := v.leaderForTerm[term]
        if exists && tl != ldr {
            v.errors = append(v.errors, fmt.Sprintf("Node %v sent an
AppendEntries request for term %d, but node %v had already done some, multiple
leaders for same term!", src, term, tl))
        }
        if !exists {
            v.leaderForTerm[term] = ldr
        }
        v.Unlock()
    }
    return nil, nil
}
```