

GCP setup

- `gcloud components update`
- `gcloud init --if-needed`
 - pick zone 'us-west1-c' -- arbitrarily (?)
- `gcloud auth application-default login --and-sign-in` w/ account

Python setup

- have `venv` available, install 3.12.0 (or something else recent)
- `pip install -r requirements.txt`
 - `pip install google-cloud-storage`
- open `/Applications/Python\ 3.12/Install\ Certificates.command`

Get hostnames from `swirls/infrastructure` repo

- see `mainnet/ansible/host_vars_mainnet/node*.yaml`
- `grep source_hostname * | sed 's/[.]yaml:source_hostname: //g'`
- use file `mainnet_hostnames-2024-02-04.txt`

To run:

First you use the script to create a file that contains the names of all the stream files in the interval that are up there in Google Cloud Storage.

(And this file contains other data, such as the size of the file, and whether or not it has been downloaded already.)

Then you use that file to download all the data files (which Google Cloud Storage calls "blobs").

Easy. Except due to cloud and network problems the smooth processing of calls to Google Cloud Storage may get interrupted from time to time. So the script allows you to *rerun* both phases until they're finally complete (keeping track, via that file created in the first phase, of stuff that's already been done, so it isn't done again.)

1. First create the list of blobs (record/event files) you want to download from gcp:

```
python3 main.py get_blob_names -root <dir-for-files> \
    -b <bloblist-filename> \
    -s <start-time> -e <end-time> \
    -node <node#>
```

where the start and end of the interval are specified like `2024-02-01T00:00:00` , and the node number is the node you want to pull files for.

This will tell you how many files it found in that interval.

But it may happen the node you picked was down for some time during that interval. So run the script again using the command `reget_blob_names` (instead of `get_blob_names`) and specify a different node number. It will *merge* additional files found into the bloblist you already have. Repeat until it finds no new files.

2. Download the blobs with the command

```
python3 main.py download_blobs -root <dir-for-files> \
    -b <bloblist-filename>
```

It will fetch files in batches - and give you a progress report on how many batches it is doing.

Files can fail to download. Keep repeating this command until you see, by the metrics reported, that all the files are downloaded.

You can "tune" the performance by changing the batch size with the `-batch nnn` argument, and by changing the level of concurrency with the `-concurrency nnn` argument.

More explanation

- What are all these "blobs" in the names of things in the source code?
"Blob" is what Google Cloud Storage calls a "file".
- All nodes put the streams out on GCP, even though they're *identical* (except for the signatures, of course). So you can pick any node number (1..29) to pull from, your choice. But sometimes nodes go down so the data is missing out there (without any indication). So this script lets you *rerun* getting the names of all the files, using a different node number ... you do that until no new files are discovered. Then, presumably, you have a full set of stream file names.
But the node you pull from doesn't matter. And so the default chosen by the script, if you don't specify one, is arbitrary.
- Speaking of nodes, nodes are always referred to by their *number* (1.. 29/30). And that's true whether the script calls it a "number" or an "id" or whatever.
- This script - and I still think of it as a script - has grown a bit. It might be a bit larger than a "script" now ... but there it is. And so, even though it is not good software engineering to use *global variables* this script does use global variables. And since those global variables are referred to

everywhere, even though it is not good software engineering to use *single character variable names* **especially** for global variables this script *does* use single character variable names to refer to global variables. Because I wanted the minimum syntax overhead to specify the context.

Because it's necessary to have the context for readability, but too much of it is just visual cruft.

- `a` is the global holder of command line arguments. All command line arguments are in there, with default values for those that weren't actually on the command line. It is of the class `Args`, and both are declared in `cli_arguments.py` and imported elsewhere.
- `g` is the global holder of global variables. It is of the class `Globals`, and both are declared in `globals.py` and imported elsewhere.

And unfortunately importing a variable doesn't work the way I expected in Python. I'm not sure exactly what happens, but I think a copy is made. Shallow copy probably. Anyway, it hasn't impacted things so far but it's something to be aware of and I'll probably have to fix it if the script has further developments.