# CS 3891 Resilient Distributed Computing Project

# Bao Nguyen and Evan Wang

**Work Division**

We started the project by drawing up a design document, which contained the architecture of the distributed file system as well as features that we wanted the file system to contain. Bao set up the skeleton code for the project and implemented an “empty” master command to spawn a master node. After that, he wrote the command line parser for the client to send up the user commands. Evan used some of the parser code to create a client script that accepted user command-line arguments and sent them, as a compressed object, using ZMQ request-reply sockets to the main.py process. He also set up the AWS instance so that the local client could communicate with the “Master” node through ZMQ sockets. Evan then worked on the upload command, which is invoked by the main.py process, and created ZMQ pub-sub sockets, where the publisher (on the client-side) took in a path, and sent chunks of data from that file to the upload command, using “path” as a topic. The subscriber would analyze the size of the chunks to ensure the file was fully uploaded. In the meantime, Bao finished the volume command to spawn additional volume nodes. After that, Bao continued the upload command logic by having the command sends to the master the uploaded file path. He also implemented the volume assignment logic for master and finished up the upload command with the final disk write in volume. Evan also set up mysql on the AWS instance and implemented the database creation in the master\_server class for the metadata. He also allowed for configurable creation of new tables, for metadata of different file systems, using the Master command. He then worked on uploading relevant metadata to the database every time a file is uploaded to the volumes. Evan also created an LS command, with a path argument, that connected with the metadata.

**Code Structure**

For this project, we spread out our code into server-side and client-side directories. On the client-side, we had a script and a parser to send user commands to the server, as well as a module that helped upload our files. On the server-side, we split the directories according to their corresponding components: command, proto, server, and helper scripts. Further information on the structure is available in our README.

**Architecture Decisions**

We decided to use ZMQ for communication between the client and the server due to its lightweight and simplicity. It provides a lot of networking patterns, and the client also needs less set up to connect to the server. We used request-reply sockets in for the client and server to communicate because it allowed us to receive a “reply” message for each of the commands from the client. This allowed us to see the status and output of each of our client-side commands. In uploading files, we made the decision to write them to the disk in 1000 byte chunks. We believe this made the system more resilient because, instead of holding entire files in memory, we were able to deal with 1000 bytes of data at a time. We also kept track of the size of the files to ensure that they were uploaded completely and handled upload errors by undoing the file writes.

We used mysql as a database because our file metadata has a very consistent structure, with the same general fields. We attempted to make all connections to the database through the master\_server process, so that multiple sources would not be editing the database simultaneously. Using gRPC calls, the associated commands would send the metadata to the master\_server, which inserted them into the metadata database and returned a response.

On the server side, we had a main process that accepts client commands. With master and volume commands, we spawn a new process using Python’s multiprocessing to run a master/volume node. These masters and volumes host a gRPC server that can be used to communicate to them. With upload command, we first download the file temporarily, then send the file path to master using gRPC. Upon receiving the file path, master will query all of its volume using gRPC to get the least used volume and assign the new file to that volume. It then splits the file into chunks again, makes another gRPC to that volume and send the chunks over for a final disk write. Each volume will have its own folder directory and upon receiving the write file request, it will assemble the chunks and write to its directory. Finally, with the ls command, we simply query our mysql database for metadata on the file system and return it to client.

We have decided to use gRPC for communication to and from master since we want the public interfaces of main, master, and volume to be as decoupled as possible. Furthermore, with gRPC we can make direct method calls between different process, without having to worry about marshalling and unmarshalling data when sending them through sockets. Since using gRPC requires us to recompile every time we update the public interface, in a way it helps to try to design the interface as good as possible in the beginning.

**Failure Modes / Resiliency**

Some possible causes of failure are network failure (especially while writing a file), metadata database failure, and disk failure. We combat network failure with our request reply sockets, allowing us to get confirmation statuses from the server on our commands before we continue to try to execute any more commands. We also upload our files chunk by chunk and will delete the file as soon as we get any sort of error. We try to write the files to the disk, instead of keeping the files in memory, to avoid overflow, and allow us to tackle small pieces of the file at a time. We will also verify the size of the file to make sure that we have transferred the entire file. For the metadata database we used a mysql database, basically allowing us to rewind and revisit the state of the database. We also have the possibility of creating new tables for different filesystems, meaning that multiple clients can spread our files across multiple tables.

In the case of disk failures, we will lose data, however, if we have moved our volume processes into new AWS instances, we would manage to shard our data so that only pieces of it are lost. If we had additional time, we could create more replicas in order to make the system even more resilient.

**Possible Improvements**

Our current implementation does not tackle a few problems. Firstly, all of nodes are running together in the same AWS instance, which could be problematic if that instance fails. Moreover, currently we have no way to recover in the case of single master or volume failure. The following section suggests some ideas to tackle these problems:

1. Separation of master and volume nodes

Ideally, we want the master and volume nodes to be truly distributed (i.e. they run on completely different AWS instances). However, for the sake of time and simplicity, we chose to run everything on the same instance to test out the functionalities first.

To achieve this separation, we can use container applications like Docker. For example, we can create Docker images for main, master, and volume. Then instead of using multiprocessing to spawn a node, we can spawn a Docker container with the built images on any AWS instances.

Furthermore, with Docker containers, we can utilize container orchestrator technology like Docker Swarm or Kubernetes to manage multiple containers. If we pre-allocate a few AWS instances and add them to be worker nodes, Swarm/Kubernetes can dynamically assign which container is running on which worker node. This approach is also very scalable since Swarm/Kubernetes can also provide replicas for the nodes and restart them automatically in case they fail

1. Master cluster

Instead of having only one master, which can be a single point of failure, we can have a cluster of master nodes and utilize consensus algorithms such as Two-Phase Commit or Raft to achieve consistency among them. Again the cluster can be deployed using Docker containers for better separation and distribution

1. Volume recovery

If we are using containers and a volume container is restarted, it should know where to look for its directory and look for master. To look for directory, we can add configuration files into our docker images so that on start-up a volume knows where is its directory. To look for master, one simple approach is to implement a heartbeat monitor for master that periodically sends out heartbeat signal to all of its known volumes. Upon receiving heartbeat, volume nodes can update itself.