CSCE 438 MP3 Design Document

The coordinator is used to handle requests from clients, servers, and synchronizers. It also includes several data structures to hold specific data, including maps for client ID’s and ports, and vectors containing structs representing servers. First, the coordinator starts a server using a given port and creates two directories, one for master and one for slave. These directories are used to store timeline files (more on this later). It is also important to note that a separate .proto file is created for the coordinator, because the coordinator becomes a server and needs its own gRPC services to be communicated with.

When a server wants to start, it first contacts the coordinator to be assigned a cluster. Its data is also stored in a server struct vector for further reference. On top of this, a thread is started that sends a ‘heartbeat’ message to the coordinator every 10 seconds to show that it is alive. This heartbeat message is a custom gRPC data type, and it includes a timestamp field that contains the current timestamp the message was sent at. Whenever the coordinator receives the heartbeat message, it updates a timestamp variable in the server struct vector and that time is subtracted from the current time every second. If the difference becomes greater than 11 (we add 1 second of error to 10 for fault tolerance), then we know that the server is dead, and we can transfer connection to the slave server. In this implementation, the coordinator only detects a SINGLE heartbeat, as we couldn’t detect two in time.

Speaking of slave server, both slave and master servers must be active if a client wants to connect. When a slave is sent, its data is sent to its master to have in the situation where the master dies, and it needs to redirect connected clients to the slave server. The port number of the slave is sent to every client and stored and is used by a separate thread. This thread is started in the connectTo() method after the client contacts the coordinator and its master server, and continuously checks with the coordinator if the master server is active. Because the coordinator declares a server inactive as soon as it doesn’t detect a heartbeat, the client thread sees the status change and exits the infinite loop. Finally, it sets the stub to connect to the slave server’s port.

Lastly, the follower synchronizer is used to communicate follow changes between clusters. In this implementation, clients in separate clusters unfortunately cannot see each other *immediately* as they can still follow each other despite not appearing on each other’s lists. First the synchronizer starts a thread that creates a file to store follow interactions. Whenever a follow is called from the client and is successful, it is documented in these files. The thread continuously detects for changes in this file using getline(), and extracts the client ids. It sends these client ids to the coordinator, who then checks what cluster the ids are from. It then sends the follower synchronizer port of the corresponding cluster. The thread then connects to the other follower synchronizer to write the message another file, which is then read by the server. The server extracts ids from the message in the same way as the follower synchronizer did, and then adds the user to its client database. After that, the server list is updated and the user from the other cluster is visible.

Another important thing to note is that our implementation doesn’t have separate directories for each server and with custom ids, as we weren’t able to figure that out in time.