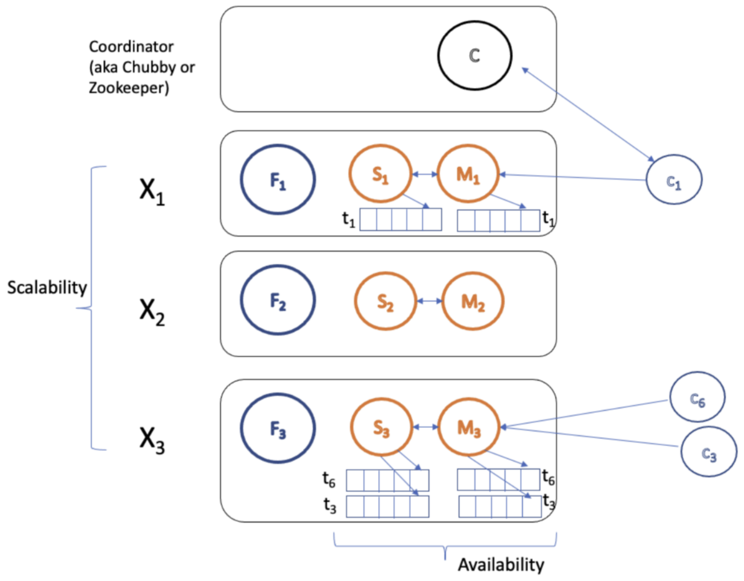
Ethan Cherian
830002546
CSCE 438-500

Machine Problem 3 Design Document

Broadly, the design of the system follows the one outlined in the instructions:



Client

From the perspective of the client, this SNS is nearly identical to the one created in Machine Problem 2. Upon startup, the client connects to the coordinator, who informs the client of their assigned server cluster and provides the requisite IP address and port number with which the client should send all future communications. By default, all client communication goes to the master server within the appropriate server cluster, and the master forwards these requests to the slave within the cluster, though this process is entirely transparent to the user. Should the master fail, the client will reconnect to the slave, who has now been effectively promoted to be the new master.

Coordinator

When a client joins, it asks the coordinator for the IP address and port number of its appropriate server cluster. As there are exactly 3 server clusters, the client is assigned to the one with `clusterId = (userId % 3) + 1`, for the user's particular `userId`. The coordinator maintains a map (in memory) between `clusterId`s and the relevant information for that cluster, so it returns to the client the address and port of the master server for the user's assigned server cluster. After this point, there should be no communication between the client and the coordinator.

The coordinator has a single thread dedicated to monitoring and appropriately handling heartbeat messages from each of the servers (masters and slaves). The coordinator also maintains an (in memory) status for each server, as well as their last recorded heartbeat. Upon receiving a heartbeat from a server (master or slave), it marks their status as ACTIVE and updates the last recorded heartbeat to be the clock's current time. If a server fails to respond to a heartbeat message from the coordinator twice (a minimum of 20s apart), that server is marked as inactive. If the newly inactive server is a master, its corresponding slave is "promoted" and begins to update the master's directory. If the process comes back online, it will become the slave. The synchronizer class also sends a single heartbeat message to the coordinator when it comes online, to inform it of its existence, and this is simply handled by the same heartbeat thread, as the coordinator's in-memory database must be updated.

Beyond that, the coordinator waits for other communications from master, slave, or synchronizer. When a master starts up, it asks the coordinator for the address and port of its corresponding slave server, so all client requests may be forwarded appropriately. When a synchronizer detects changes to a timeline, it determines which users must learn of the new posts and asks the coordinator for the addresses and ports of each user's assigned synchronizer. Both of these tasks are handled by a single gRPC call to the coordinator each, `GetSlave` and `GetFollowSyncsForUsers`.

Server Cluster

Each cluster is composed of exactly three pieces: a master server, a slave server, and a synchronizer. Each of these share a common `clusterId`, which serves to logically group each of the three clusters together. There is next to no difference between the master and the slave, and the slaves solely exist to provide fault tolerance in the event of a master's failure.

The master and slave servers each have their own directory of persistent files, ideologically stored on separate machines, so all information between them is communicated via RPCs. For simplicity, the synchronizer is considered to be on the same machine as the master, so it has direct access to the master's directory and files contained therein. Within each directory is a file containing all users in the entire system, along with a number of files for each user assigned to the cluster: relation information is stored in two separate files (one for following, another for followers), and timeline information is stored in yet another file.

Both master and slave keep an in-memory store of all users on the system, as well as two total maps for each user's following and followers. This data is updated whenever a change is detected in the corresponding files within their directories, though only masters actually perform this file monitoring. These files can be changed by the synchronizer servers, upon receiving a notification from another synchronizer server that their data needs to be updated. Whenever a server is brought up, it checks the contents of its files (should they exist) and updates its in-memory contents with everything that has been persisted.

Master and slave servers send heartbeat messages to the coordinator every 10 seconds and receive replies. If a server fails to send 2 heartbeat messages in a row, it is marked as inactive and considered "dead".

Master

Each master periodically checks the contents of its directory (specifically all stored timeline files) using a thread and reports any modifications to its slave, so it may be duplicated appropriately.

If a master server dies, the coordinator informs the corresponding slave, who is then “elevated” to master

Slave

Slave servers are functionally identical to master servers, with the only real difference being that clients communicate with the master, who relays requests to the slave, for the purposes of fault tolerance. To that end, if the master fails, the client will connect to the slave, who will begin writing to (and reading from) the master’s directory, effectively making it the new master. If the old master comes back online, it becomes the new slave, and the former slave continues to operate as the master.

The slave does not track the contents of its directory, as it relies on the master to relay all necessary information to it.

Follow Synchronizer

The synchronizer monitors the master’s directory of its cluster for changes to files. If a file is changed by the master, the synchronizer determines what kind of file was altered (all users, relation, or timeline), the new content since the last recorded change, and who must be informed of these changes. With knowledge of the users being added/changed, the synchronizer makes a call to the coordinator’s `GetFollowSyncsForUsers` RPC to determine where to send this updated information to: specifically, the synchronizers want to send these updates among themselves. Using one of the synchronizer’s RPCs (`SyncUsers`, `SyncRelations`, `SyncTimeline`), a synchronizer can inform the other synchronizers of any changes that need to be made to files, and the receiving synchronizers can make them within their own cluster.

The synchronizer uses threads to check for changes to its assigned files: one for the users file, one for *all* relation files and timeline files. In order to prevent a race condition where a master and synchronizer write/read from a file concurrently, the system creates a file which functions as a lock. A process may only act on the file if the corresponding lock exists, and it will release this lock when it is finished by destroying that file. An identical process is used in the master/slave server to keep the same guarantee on their end.

Running Instructions

To run the system, run

```
      > make
      > ./startup.sh
```

This will run one coordinator, three synchronizers, three master servers, and three slave servers (10 total processes). The coordinator will be located at `localhost::9000`, and any clients may connect to it there. The synchronizers use ports 9070, 9080, and 9090, while the master and slave servers use ports 10000-10006, so clients shouldn't use these.

Clients should be run as follows:

```
      > GLOG_log_dir=./logs ./tsc --cip localhost --cp 9000 -p
{port_num} --id {user_id}
```

with the appropriate port number and user ID substituted in.

Cleanup Instructions

To kill all processes, run

```
      > ./terminate.sh
```

This will terminate all 10 processes created by `startup.sh`.

To remove all created files, run

```
      > make clean
```

This will remove all compiled files, as well as all log files and persistent data.

To remove only the files for persistence (users, relations, timelines), run

```
      > make dclean
```

This will remove only the persistent files and leave compiled gRPC files, object files, and binaries intact.