Trevor Ledbetter, 326001657
Anthony Nguyen, 826005614
CSCE 438, Section 500

# Project 4 Design

## Compilation Instructions

The system can be compiled and deployed using the `launch.sh` script. If it is not able to be executed, run `chmod +x launch.sh` to get it working. Running the script with `./launch.sh` will show the argument options. Every option will run `make` to compile the needed executables, it will not, however, run a clean build. The script will need to be run on each VM according to what the purpose of that machine will be. A routing server should first be started to handle and register the other master/slave server nodes when they are started. This can be done by obtaining the IP address of the current machine with `ip addr`, and using the broadcast IPv4 address of the machine. Then run `./launch.sh r <IP Address>` to launch the routing server on that machine. On separate VMs, to launch the server nodes, obtain the IP address again, and then run `./launch.sh s <IP Address of Machine> <IP Address of Routing Server>`. Finally, a client can be launched by running `./launch.sh c <IP Address of Routing Server> <Username>`. When a process terminates, it is restarted by the slave/master, but the terminal it resides in will return to bash. The windows should not be closed until testing is done, after which all processes can be terminated at once by closing all terminals.

## Routing Server and System Design

We constructed our fault tolerant system by adapting our P2 code since we were already very familiar with the logic and happy with its performance. The first major change is the use of a routing server that the client contacts in order to obtain the port number of the available server. The routing server implements three new RPCs, to accomplish this.

First, it uses the RegisterServer RPC to maintain a list of master servers that can act as the available server. When a master server starts up it uses RegisterServer to be added to the router's list.

Next, the Connect RPC is used by the clients to obtain the port of the current available server. The clients first make a stub for the routing server so they can call Connect, then the clients make another stub for the available server using the port obtained from Connect. This master stub is used by the client until it detects that its master server has crashed. After a crash the client will rerun connectTo(), to obtain a new stub, then will retry the RPC call that it failed on.

Finally, the routing server also implements the Crash RPC that is used when a master server crashes. Each master server also has a slave server that notices when the master has crashed, and then notifies the routing server by using the Crash RPC. The router will then remove that master server from its list of available servers and if that master was the acting available server it will pick a new one.

The master server (server.cpp) also implements the KeepAlive RPC for communication between the master and it’s slave. The slave will send a KeepAlive message to its master every three seconds, if the grpc_status returned is not “OK” then the master is down and it will notify the router using Crash, and will restart the master using fork() and execvp system calls. On the other hand, the master will keep a variable checkTime that keeps track of the last received message from its slave. If too much time passes between messages the master will restart the slave. We choose to detect slave server failure in this fashion because it’s simpler, and it reduces message overhead.

## Slave Server

The slave “server” acts more like a client because it only really sends messages to its master and the routing server. Its main function stems from the fact that it runs on the same machine as its master so when the master crashes, the slave can restart it. Of course it also detects when its master dies through the use of KeepAlive messages as mentioned above.

## New RPCs

- When a server starts up it will register with the router as an available server.
  - rpc RegisterServer (ServerInfo) returns (KeepAliveReply) {}
- Client sends to routing server, reply is the available server's IP/PORT.
  - rpc Connect (ClientConnect) returns (ServerInfo) {}
- Sent from slave to routing server notifying router that the master server has crashed. Also the master server is unregistered with the router.
  - rpc Crash (ServerInfo) returns (KeepAliveReply) {}
- Sent between master and slave servers.
  - rpc KeepAlive (KeepAliveRequest) returns (KeepAliveReply) {}