Team 19 Identity Server Whitepaper

- Tyler Bevan
- Phil Gore

Node Creation

When a new server process starts up, it will send out a multicast message asking the running server to broadcast their process rank. The new process will set its process rank to the lowest available id starting with 1. This allows the cluster to grow continually. Also, the process ranks will not outgrow the number of servers running. Periodically the nodes will send out a heartbeat signal to tell the other servers that the node is still alive and available. This is triggered by the coordinator sending a "GETRANK". The nodes that are still alive will respond "RANK" followed by its rank and the latest event clock it has seen. Node syncs are full duplications of the running Redis database. We use the Redis database SAVE functionality for replicating the database. We are reusing the file copy code from project 1 to send the file over the network. For clients that want to connect, they send a "GETMASTER" request for the current master. The master server responds with its rank and address. In the next stage, the client will send a "GETRANK" and bind to the server that responds the fastest, and remember the others as failovers in the order of response.

Electing a coordinator

Our system elects a new coordinator whenever the acting coordinator fails to respond to an rpc request from one of its peers. The server that decides to run the election sends a multicast "GETRANK", and chooses the lowest rank server with the highest event clock as the master. It then multicasts a "NEWMASTER" message to tell the cluster to communicate with a new server. The new coordinator will multicast a "MASTER" message announcing that it is in charge. If a machine that is the coordinator gets a "NEWMASTER" from another server, then it will do another election to decide who is the real master. There can be only one. For this stage, the coordinator creates all new events and handle all client connections. In the next stage it will handle new event distribution, but will not be the only connection handler.

Event Ordering

Add, Modify, and Delete functions are always inter-dependent. The Lookup and ReverseLookup functions are dependent on those but nothing depends on them. Because of this, only Add, Modify, and Delete will increment the event counter. We will order events by a Lamport timestamp. Each event will increment the clock by one, and events must happen in order. This way, if a node gets two events it will know in what order to apply them. If there is a conflict between two events, the event from the node with a lower rank will take effect first. This allows us to predictably order events, with lower ranked servers being considered more "authoritative". For this stage of the project, only the coordinator will create new events. We can implement this functionality, but it will mostly be important during the final stage of the project.

Consistency

We are using sequential consistency as our model. To ensure that there is consistency in the system, the acting coordinator will not report a successful operation to the client until the data has been replicated to at least one other server. If there is only one server in the cluster then it will return anyway. Because of this, the client will be able to detect an operation timeout and retry the operation. The inconsistency window for this system is the time between when the coordinator stores the operation in its own database and successfully replicates it to another server; and when it replicates to a third machine. That time should be no more than 50ms as long as the servers are connected by a fast local network with less than 10ms of latency. If the network latency is higher, then the window will have to be larger to account for that. Our testing showed that our software takes 18-25ms to process a request depending on the type. For this stage, all updates will come from the coordinator. In the next stage, each server will push its updates to the coordinator who will then distribute them out to all of the nodes.