Akshey Sigdel
Kaden Davis

CS455/555 White Paper

Design Choices for a Reliable RMI Identity Server with Replication

Client-Server Model: We stick with a one-shot client with separate servers. Further, the client will only interact directly with one server at a time.

Consistency Model: We choose to implement our servers such that sequential consistency will be maintained. The reason we chose this over a less strict model was partly due to how we have implemented our server and client in the previous project but we also feel that this is a fitting model to use as a less strict model such as eventual consistency may lead to critical errors in the future.

Inconsistency Window: Due to how we have chosen to implement our consistency protocol, our window of inconsistency will be the time between the coordinator acknowledging a modification request and the modification being performed and replicated.

Assumptions:

1. If the coordinator fails before replicating the data modification log to the other servers, the log information is assumed to be lost.

2. No server exhibits malicious behavior.

Implementation Details:

1. Consistency Protocol: Similar to the non-blocking primary backup protocol, any server may handle read operations on data but only the coordinator server will have the right to perform write operations. In other words, we will allow the coordinator to be the primary server for all data. If a client is connected to a non-coordinator server, that server will forward the write request to the coordinator which will acknowledge the request and perform the write.

After performing a write operation, the coordinator will replicate each log event (including the client identifier and updated data) to all other servers. Then, it will wait for an acknowledgment from at least one other server to be assured of successful replication. If an acknowledgment is not received within a timeout period, it will retry the replication.

The coordinator can make use of Lamport timestamps or increasing numerical identifiers for each successive log event to achieve sequential consistency. The entire server state will be periodically checkpointed on each backup server when prompted by the coordinator. To reduce overhead, it updates a log containing all modifications since the last checkpoint.

2. Coordinator Election: We intend to employ a variation of the bully algorithm. The server starting the election will send an “ELECTION” message to all servers. Each server then broadcasts a message containing the highest value (timestamps or identifier) from their log events, as well as the server ID. Servers with a higher log timestamp (higher server ID in the case of equal timestamps) will respond with an "OKAY" message. The server that receives no “OKAY” messages will be the one with the most up-to-date data and will become the coordinator. The new coordinator will then send a “COORDINATOR” message to all other servers in order to notify them of the election results. In the possible case that 2 or more servers report the same value, the server with the higher ID will win the election.

3. If the coordinator server goes down: This will be detected by the backup servers after receiving no response to the heartbeat messages that are sent to the coordinator after a predefined timeout window. This will trigger an election. Due to the way we are implementing our election, the server with the most up-to-date data will become the coordinator. This will help to minimize the amount of data loss.

4. Recovery: If a server fails/restarts, it will retrieve the latest checkpoint from one of the backup servers. After restoring the checkpoint, it will synchronize its state with the coordinator server by applying modifications from the update log. The timestamps in the log will allow the server to apply the updates beginning from the last checkpoint.

5. Server Discovery: Clients will maintain a static list of all potential server addresses over which it will iterate to connect to discover an available server. I.e., if the client fails to connect to a server, it moves on to the next one in the list. We may also provide the client with a command for requesting an updated list of servers.

When a new server comes online, it will notify the coordinator which will then notify all other servers. The new server will be added to each server list maintained by all other servers. The coordinator will also send the updated list to the new server.

6. Other scenarios:

If a server believes it is the coordinator but has not received any heartbeat messages, it should begin an election.

If only 1 server is running, it will be the coordinator but should not expect heartbeat messages. All other functions should work correctly. This can be detected during an election. Additionally, the server should not wait for acknowledgement messages from other servers.

If a backup server goes down, all other servers should continue to function because we will make no assumptions that any server is up or down.

Each time a new server is started, it should bring itself up to date with the other servers (see no. 4) before accepting requests from clients.
Couple of design choices that significantly improve the system performance:

1. Reads are accessible by any server, but only a designated coordinator server can modify the data (perform writes).
2. Instead of sending updated data/state, our servers only share the event log (based on timestamp) which they then apply locally to synchronize. Only if a server is unable to successfully synchronize using the logs will it request a copy of the latest checkpoint from the coordinator server.

Additional Note:
Unlike RAFT/Paxos, our approach doesn't implement distributed commits for the event replication which we realize might introduce some divergence. However, just for the sake of simplicity, the replication is assumed to be successful with acknowledgement from at least half of the other servers.