Mandatory Activity 5 - Distributed Auction System

Introduction

We have developed a distributed auction system with leader-based replication. We have implemented service discovery for the servers, so that they can find each other without manual user configuration.

The system can handle n-1 crashes with n servers, meaning you can crash all but one server and continue operation (assuming the client also knows ports of the remaining servers).

As described in the mandatory activity, our system has implemented the two required API methods bid (to send bids) and result (to get current auction state).

Architecture

Service discovery

We use a simple service discovery algorithm. The first node will always connect at the same port (33345). If the port is not available, we will know that it is an active node, so we can connect to it and exchange ports. Then, every node will be kept up-to-date on which nodes exist, since any new nodes will be shared with all existing nodes.

Replication

The servers use leader-based replication to ensure n-1 fault tolerance. The leader is elected through the bully algorithm where their ID is their port number. Once a leader (coordinator) is elected its followers recieve changes from the leader.

To avoid a crashed leader not being discovered the followers ping the leader every five seconds with a five second timeout. The timeout is five seconds, and not something less like 500 ms, to avoid unnecessary failovers due to slow connection.

To ensure zero data loss we use synchronous replication. This is because we are handling "money" and care more for data loss more than latancy. The leader update it's followers through an rpc call and the followers updates their variables and send back an acknowledgement. A follower crashing doesn't really affect the system. Although synchronous replication is used, the leader immediately gets a connection error when trying to update a crashed follower. This avoids unnecessary latency of waiting for a timeout because the follower won't send back an acknowledgement.

A failover (leader chrashes and new leader gets elected) gets started when either a followers ping gets timedout or a client tries to get data but can't due to crash of leader. The system calls an election and use bully algorithm to choose new leader.

Correctness 1

Linearizability is when a system gives out the illusion of having only one copy of some data, when in reality the data is distributed between multiple nodes in their own data storages. Having one copy of data means that every request to read this data will result in the same data being sent to every requester, given the server

recieved the requesters' requests at the same time. If the requests were recieved independently, this promise would not apply since the data could have been changed in the mean time. However, if the server recieved no writes between 2 reads, those 2 reads should get the same respond.

Our system consists of multiple server nodes, where one of these nodes is the leader. The leader's job is to respond to the clients querys and make updates to the stored data. That means only the leader can make changes and only the leader can respond to clients about the auction. Every time the leader makes changes to the data, the leader sends updates to all the follower nodes about what has been changed to keep them updated. If the leader goes down, a new leader is elected with the new data and the auction may continue. The clients find out who the new leader is by asking one of the other server nodes, and since there is only one leader all clients will get the port for that one leader. We can then safly assume that the system is linearizable.

Actually, I lied. We can not assume that the system is linearizable, since there still is some edge cases that makes the system somewhat not linearizable. One of the cases are as follows: If the leader only sends out the updates to some of the followers before the crash and if the new leader only has the old data, this would not be linearizable either. Here we could have made use of lamport timestamps so that the followers only elect a leader who has the new data. A node which recieved an election notice would first compare the sender's and its own lamport timestamp to see which one had the newest data. If the timestamps are equal, then it would compare the ports.

All in all, safly assuming that a system is linearizable is hard. Perfect linearizability is close to impossible to achive.

Correctness 2

Our protocol works as expected during normal operation (in the absence of failures).

- The client can connect to any node and discover the leader if necessary.
- Bids can be sent to the leader without issue, and the server replicates the bids to any working followers before sending a success reply.
- The state can be queried on any of the nodes, whether they are leaders or followers.

Our protocol can also work in the presence of failures.

- If a follower fails, the system will continue as usual; the leader will simply ignore the failed follower.
- If a leader fails, the followers will eventually discover the failure and start an election. Then a follower will be promoted as the new leader.
- If the client can no longer reach the server, and it knows multiple ports, it will reconnect to a new port and discover the new leader. From there, bidding can continue as usual.

Link to repository

https://github.com/kie2204/Distributed-Auction-System