Distributed Systems - Homework 5

Replication

Group: Jacob, Jacob og Jacob

Matthias Nyman Nielsen - mnni@itu.dk
Jacob Præstegaard Folkmann - jafo@itu.dk
Jacob Hørberg - jacho@itu.dk

November 26th 2025
Link to GitHub: `https://github.com/natthias/replication`

Introduction

We implemented a distributed auction system that remains operational despite the crash of a node. The system consists of several gRPC-based nodes arranged in a leader-centric architecture.

When an auction is started, the leader opens the auction for a fixed time period and all state changes (bids, current winner, and auction status) are replicated to followers. Clients connect to the leader to place bids or query results. The leader periodically sends a hearbeat, and if the leader crashes, the remaining nodes run a simple leader election protocol and continue the auction using the replicated state.

Architecture

The architecture follows a leader-centric replication model, where exactly one node acts as the leader and the remaining nodes act as followers. The leader is responsible for handling client requests, replication, and the auction state.

Node Structure

Each node runs as an independent process and provides two gRPC services: `Auction` for client-facing operations (`Bid` and `Result`) and `AuctionNode` for internal coordination (heartbeats, replication, leader election, and state updates). Nodes communicate exclusively through gRPC.

Internally, a node keeps a Lamport logical timestamp, a map of bidders and their current highest bids, the name of the current winner, and a flag indicating whether an auction is running. Each node also stores RPC clients for the other nodes. All shared state is protected by a single mutex to ensure thread safety.

Leader Election

Initially, the node that successfully binds to the designated leader port (`:6969`) becomes the leader. The leader periodically sends heartbeat messages to followers. If a follower does not receive a heartbeat within a timeout window, it assumes that the leader has crashed and initiates a leader election.

When the Leader election is performed, the nodes compare port numbers to decide who to grant the leader role. This will always be the port with the lowest value. If a node wins the election: it marks itself as leader $\rightarrow$ rebinds the leader port $\rightarrow$ notifies all other nodes $\rightarrow$ begins sending heartbeat messages to confirm its livelieness.

Replication Protocol

The leader ensures consistency across replicas by replicating all state changes to the followers. When a bid is accepted, the leader updates its local state and sends a `Replicate` request to all followers. This message includes the bidder name, bid amount, and current winner. Then when an auction starts, the leader resets the auction state and calls `ReplicateState` on all followers so they also enter the auctioning state. After a fixed interval (30 seconds), the auction is automatically closed.

Logical timestamps are implemented, however, they are not actually used to maintain an ordering of events.

Correctness 1.

We will first argue that our program conditionally provides sequential consistency, as it is a strictly weaker guarantee than linearizability.

Sequential consistency means that all requests appear to have the same order on every node. Our program guarantees this by the use mutex, to ensure only one replication can occur at once, in addition to the leader serializing the bids. In addition to this, the assumption of a perfect networkbrings us closer to having sequential consistency. As this gives us the guarantee that a message will make it to the other nodes. All these things in combination ensure a queue of messages providing a total order.

However, certain situations caused by a crash in the leader can lead to the program no longer satisfying sequential consistency. Consider the following events:

- a client places a bid
- the leader recieves and stores the bid
- the leader sends a replication request to follower1
- the leader crashes
- follower2 is elected as the new leader

There is now an inconsistent state between the two nodes, causing sequential consistency to no longer be true. This could be solved by adding consensus to the implementation

Linearizability is the requirement of a total ordering of events across clients, respecting the real time ordering of when events occured, this is a guarantee, which is strictly stronger than sequential consistency. Normally, leader-centric replication can provide linearizabiliy. However, as discussed earlier, since sequential consistency is only true under certain conditions (no leader crash during replication), the system can never satisfy linearizability, at least under the conditions where sequential consistency does not hold true.

Correctness 2.

In the absence of failures, When no node crashes, our system behaves correctly because all operations are ordered through a single leader (the node with the lowest port). Every bid is validated and then replicated to the followers before being acknowledged. Since all replicas receive the same sequence of updates, all nodes converge to the same auction state. Therefore the system produces a consistent result.

In the presence of a signle crash failure where the follower crashes, the leader continues operating normally and replicates updates the remaining followers. The system remains safe because the leader always holds the up-to-date state. If the leader crashes, the remaining node with the lowest port is elected as the leader. As long as the crashed node stays down and all previously comitted bids were replicated before the crash, the new leader has a correct satate and the system continues to operate correctly.

One known flaw is the previously discussed situation, which leads to loss of sequential consistency and linearizability. A leader crash during a replication event. This can result in the state of two nodes being inconsistent, which would prove problematic for the final auction result, as nodes could disagree on a winner.