Handin 5 - A Distributed Auction System

Author(s): Johannes Jensen
Course: Distributed Systems — Handin 5
Date: 26-11-2025

Repo: Github repository

Table of contents

- Introduction
- Architecture
- Correctness 1
- Correctness 2

1. Introduction

This project aims to implement a distributed auctioning system in Golang using gRPC, and a leader-based replication based on these slides. The system consists of:

- Clients that submit bids and query auction results
- Auction servers that expose bid operations to clients and hold local auction state. Each auction server has a backend component:
    - Backend: (running on the same node, as the auction server) that coordinate leader election, maintain a replicated view of the auction state, and propagate updates among replicas

Service and replica discovery use zeroconf, allowing nodes to dynamically join and leave the system.

The system maintains a auction in which clients repeatedly bid, the auction servers ensures replicated consistency of the current highest bid. Clients can join at anytime and participate in the auction. The auction will end when the Lamport Clock is over 100.

2. Architecture

The project is implemented in two parts 1. Clients and 2. Auction servers. The Clients and the Auction server communicate via the gRPC service "*Auction*", while the Auction servers communicate with each other via the gRPC service "*Backend*" this separation ensures that clients only access the functionality intended for them, while internal coordination (election etc.) and replication logic remain encapsulated server-side.

Clients

Clients discover frontend servers using mDNS (via zeroconf) and connect via gRPC. When the client has successfully connected to just 1 auction server it will continuously try to outbid the current highest bid until the auction is over. If the connection to the connected server drops or stops functioning it will connect to the next auction server that it can find via mDNS.

Auction Server

Each Auction Server implements Bid and Result via gRPC, it maintains local auction variables like currentBid, isOver, and a Lamport clock that is shared with the backend server running on the same node. If the node is not the leader it will forward any bids that it receives to the leader through the forwardBid() method. The bid is then executed on the leader and any replicas (that it has discovered trough mDNS) will then be synchronously updated.

Auction server (frontend) is responsible for:

- Receiving bids from clients via Bid()
- Returning results via Result()
- Updating the other replicas with the new bid
- If not the leader, then forward bid to the leader

gRPC methods related to auction service:

```
service Auction {
   rpc Bid (Amount) returns (Ack);
   rpc Result (google.protobuf.Empty) returns (AuctionResult);
}
```

Backend Replication Layer

The backend server layer, is mainly responsible for discovering nodes/replicas via mDNS and keeping track of who is alive (using pings and timeouts). Elections via the Bully algorithm is also done via the backend.

Every backend server is responsible for:

- Leader election with using the Bully algorithm
- Replicating updates using gRPC (TryToUpdateBid)
- Heartbeat monitoring (Ping)
- Failure detection and re-election

gRPC methods related to backend service:

```
service Backend {
    rpc TryToUpdateBid (Amount) returns (Ack);
    rpc Forward        (Amount) returns (BackendAck);
    rpc Election       (Message) returns (Answer);
    rpc Victory        (Message) returns (Ack);
    rpc Ping           (google.protobuf.Empty) returns (Answer);
}
```

Bully

The Bully algorithm is a leader election algorithm for distributed systems where crashes are allowed, and can be detected by timeouts. There are 3 types of messages:

- Election
- Answer
- Coordinator (Victory)

The algorithm

```
On pi receive ‘election’ do
    if i == max process id        // I am the bully
        send ‘coordinator(i)’
    end if

    for all (j > i)
        send ‘election’ to pj, timeout c
    end for

    on timeout:                   // all higher-ID processes are dead
        for all (j < i)
            send ‘coordinator(i)’
        end for
    end on
end on
```

3. Correctness 1

A system is sequentially consistent if:

The result of any execution is the same as if the operations of all nodes were executed in some sequential order, and the operations of each individual process occur in this sequence in the order specified by its program. (taken from slides)

Which means that there should exists a legal interleaving of all operations creating behavior consistent with a single correct copy.

All bids made to the *service* is passed through a single leader, (Bid directly or via forwardBid). The leader imposes a total order on updates. Because of this the leader becomes responsible for deciding whether a bid is accepted or not, and for determining the order in which bids are applied. When the leader has accepted the update it is then send through to the replicas.

Replicas apply updates in the order received from the leader, and the leader never issues two updates concurrently, updates all happens synchronously to maintain order.

Reads observe the currentBid value consistent with the leader's chosen order. Clients call Result on frontends, which return the locally replicated value matching the leader's chosen update ordering.

4. Correctness 2

In fault-free executions exactly one leader exists. Nodes start with a configured leader (discovers a leader via mDNS), this makes it possible for all bids to go through the leader, providing a total order of updates. The leader updates replicas after each accepted bid, ensuring eventual consistency of all nodes.

Clients always receive an up-to-date and legal auction state, as reads come from frontends synchronized with their backends.

The system tolerates crash-stop failures of either a:

- Follower node

or a

- Leader node

The minimum number of nodes, needed to keep the system is alive is 1.

Follower Failure

If a follower node fails, the leader will continue to process bids. Timeout detection (10 seconds) will eventually remove the follower, from the list of known replicas. Most importantly replication continues to remaining followers.

If any clients was connected to the follower node, they will reconnect to the next available node (they discover via mDNS).

Leader Failure

If a leader node crashes, follower nodes will notice (when the leader stops responding to pings), and eventually call for an election of a new leader using the Bully algorithm

No loss of auction data occurs because state is replicated on the backend nodes. If any clients was connected, they will (like with the follower node) reconnect to the next available node.

Safety Under Failures

Only one elected leader accepts updates at a time. Before an follower tries to forward a bid to the leader, it will check if the leader is alive, if no, an election will be triggered. There is no scenario where two different bids are committed concurrently, even across failures. If the follower node is the only node remaining in the network, it will promote itself to leader, this can lead to issues in the event of network partitions.

Liveness Under Failures

If at least one node remains alive: A leader will eventually be elected. Clients will eventually reconnect to some auction server. Bids continues to be processed when a new leader has been elected. Thus the system preserves safety always and liveness when some node remains available.