Distributed Auction System

Johannes Jørgensen (jgjo),
Kevin Skovgaard Gravesen (kegr),
Joakim Andreasen (joaan)

November 24, 2024

1 Introduction

This small GoLang program is a demo of active replication. A list of ports can be added to a .env file, which the clients and server (nodes) can use to find their respective ports.
The demo assumes that no messages are lost, which would require some alteration to the code if this was the case.

2 Architecture

Our implementation of the distributed auction system applies the active replication principle, which means that the installed servers do not have a primary server. Instead, they all maintain a direct connection to the clients.
To validate the responses given by the servers, we could implement a system that compares all responses from the servers and selects the response that occurs most frequently - as a form of pseudo consensus between the servers. This ensures that no corrupt server can provide an incorrect response to mislead the client. Furthermore, it guarantees that a single server can crash or go offline while the clients can still access the service without any downtime. Communication between clients and servers is facilitated through gRPC and protocol buffers (proto).

This means that our service has two APIs: one for bidding (*bid*) and one for retrieving the auction results (*result*). These APIs use proto objects, which simplify communication between clients and servers by employing serialized structured data. In our implementation, we have four proto objects. The first is Amount, which is used for placing a bid and includes the bid amount and the bidder's username. The second is Outcome, which provides a boolean (isFinished) indicating whether the auction has ended, the highest bid (*price*), and the username of the highest bidder. The third important proto object is Ack, which is a boolean confirming that the bid amount has been acknowledged by the servers. The fourth proto object acts like a return void statement, as it does not return any significant data to the system. This is necessary because gRPC does not support void statements.

3 1st Correctness

If a system is linearizable it means that the systems needs to process operations according to their real-time invocation. This includes the travel time of messages sent in the system. So if a user farther away from a server sends a request, that request has to be processed before a possible closer invocation.

This demo does not satisfy this requirement because of potential race conditions between the servers (nodes) and clients. Ofcourse this demo runs locally on the same machine, which minimizes the risk for out of order handling of the requests.

But if the system were deployed globally, theses race conditions would become every more problematic.

A sequential consistant system does not need the real-time order, but the timeline of operations still needs to be the same on every client.

This is easier to deal with, and if the nodes in the demo locked a central database, then the system could be sqeuential consistant. Altough that would mean that there is a single point of failure on the database that also would need to be accounted for.

4 2nd. Correctness

In the absence of failures, the implementation is designed to ensure the correctness of bidding, protocol resilience, and consistency of results.
We ensure that our protocol remains correct in the presence of failures by using active replication. If a server fails, clients can still interact with other active servers via the multi-port setup specified in the .env file. This implementation thus ensures resilience if a server goes down.

The implementation ensures the correctness of bidding by having the auction house servers handle bids and validate them. However, it does not guarantee safety against race conditions when two users bid at the same time.
Clients can independently query the results of an open bid at any time. The servers handle these requests and reliably return the current auction outcome for that server. However, we do not implement any form of consensus between servers. Therefore, the implementation does not ensure consistency of state across the servers.

4.1 Improvements for robustness

There are mutible changes that could be done in the current implementation, to further ensure the protocols robustness.

1. The implementation of state replication for $highestBid$ and $highestBidUsername$ across all servers would ensure consistency of state between servers, even if a server failed. This could be done with the use of a leader-based design to maintain the state of the servers.
2. The implementation of Lamport Clocks to handle concurrent operations and ensure robustness across the system.

Link to Github repository

https://github.com/ITU-DISYS2024-CENTRALIZEDSYSTEMS/distributed-auction-system