Let's GO
Sophie Gabriela Laksø
GITHUB: https://github.com/tomat-suppe/disys-handin5/tree/main

Introduction:

This program simulates a Leader crash, in which case a back-up server/Follower takes over as the new Leader. The server-change is invisible to the client, and only servers are able to see whether they have achieved Leader status or have crashed.

The program is implemented with two servers as was minimum requirement. Bidder clients reside within the clientside.go class.

The servers log Highest Bid in real time to a log placed at /tmp/logs.txt. To see this log after having run the program: **In the terminal, write "cat /tmp/logs.txt".** It also logs time of server crash to /tmp/logstime.txt. The logs are instrumental on an alternative branch, LogReliantSystem, but less so on main, where they're just a failsafe, in case a Server crashes in-between heartbeats or in case data has to be retrieved and both servers have crashed.

Otherwise, observe the outputs in the client-side and the server terminals, or simply refer to attached picture in this report (last).

Architecture:

Serverside.go is the original Leader. It handles gRPC requests from the Clients, as well as sends heartbeats to the Follower with current Highest Bid and TimeSinceStart every 2 seconds (also using gRPC).

Backupserver.go is very similar to Serverside.go, but it listens for a crash on Serverside.go. When it gets such, it switches to a Leader state. As such, Backupserver.go is running from the start of the program and does handle gRPC requests from Clients, but only asks the Client to re-route their request, until it assumes Leader state itself. It has received updates on Highest Bid and TimeAuctionHasRun every two seconds from the leader, but in case Leader crashed in-between heartbeats, backupserver.go has the logs to rely on (comparing if the latest entry in the log is bigger than the value it already knows).

Clients can bid on the auction and query the result. In the gRPC, the main message is the Bidder item, as Client always simply sends a Bidder, regardless of whether Bid() or Result() is called. Bidder has an id and a BidAmount, which is used for keeping track of which Bidder has bid how much, and to ensure that a Bidder will never bid a lower number than their own previous bids. They can however bid lower than *other* Bidders, in which case their bid is rejected!

A call to Result() will let the Bidder know whether the auction if over, and what the current/end highest bid was. Auction is over if 30 seconds have passed.

Let's GO
Sophie Gabriela Laksø
GITHUB: https://github.com/tomat-suppe/disys-handin5/tree/main

Serverside.go is hardcoded to crash after 10 seconds, for the purpose of this assignment, where we have to show the server crash. Both servers output to the terminal when serverside.go crashes, and backupserver.go acquires Leader state.
Server-to-client (and reverse) only communicate using gRPC, server-to-server communicate with the logs and gRPC.

Correctness 1:

A linearizable system would ensure that as soon as a client has read a value, all following clients requesting that value will get the same answer: until it gets updated, of course.
Sequential consistency, as far as I understand from page 351 in the course book, is when the update to data between server nodes is not necessarily instant, and so a server might read stale data.
My backup server does not broadcast any actual values to the clients until it becomes Leader. The Leader does log Highest Bid in real-time to the log, but only sends the data to the Follower every 2nd second. However, when the Follower becomes the Leader, it uses the log to check (and potentially update) its data about Highest Bid and TimeSinceAuctionStart. As such, to the client, the system appears linearizable, because they will not be able to detect a server change, as the values from previous Leader are immediately carried over. I would not strictly categorize it as either, because a client querying a Follower gets a re-route message.

Correctness 2:

Assuming no failure at all:
My main server can run entirely independently of the back-up server. The Follower would never achieve a state to manipulate the auction values, but the program would look as expected to a Client.
In the presence of failure: My protocol only tolerates one failure, as was the requirement. As I have hardcoded two servers, and they can only do 'total failures', in the case of which both of them fail, there would be no protocol to start up a third server. It would probably be pretty easy to extend the protocol such that, upon failure of the back-up server, a new server would start and take over, but this was not a part of the requirements. In the case of only one failure (failure of the main server/first Leader), the back-up server assumes Leader status and starts manipulating the data for the auction, as well as giving the client the expected answers. The system looks the same to the client whether there is one failure or none.

Let's GO

Sophie Gabriela Laksø

GITHUB: https://github.com/tomat-suppe/disys-handin5/tree/main