Mandatory Handin 5 - A Distributed Auction System

Nicolaj Heinrich Pedersen (nihp) & Mikkel Bistrup Andersen (mbia)

November 2022

`https://github.com/mbia-ITU/DISYS-HandIn-5`
IT-University of Copenhagen

Contents

1 Introduction

We have created a distributed auction system that supports n number of nodes, where n is greater than zero. We recommend using three nodes with one server pr client for our auction system. Each server comes with it's own replication manager and uses active replication to communicate with other nodes, meaning there are no acknowledgements between nodes with the exception of errors, low bids or auction over.

2 How To Run

1: the program is hard-coded to work with 3 total servers/replicas, but with an unlimited amount of clients
2: we recommend readying 3 servers and 3 clients in 6 total terminals, thereby removing some of the time pressure of starting each individually
3: start the 3 servers with port 5000, 5001 and 5002 (it will ask for port after the initial "go run server.go"
4: start the three (or more) clients, they will auto connect to localhost with ports 5000, 5001 and 5002
5: you can now bid by typing any integer, or query the status of the auction by typing "status"
6: after 180 sec. the auction will finish, giving the result and returning "AUCTION_OVER" to any client that tries to bid
7: to simulate a crash, simply close one of the server terminals, to reconnect a crashed server, simply start it up again with the same port.

3 Architecture

Our system makes use of active replication. This can be seen in our implementation in client.go specifically in our MakeABid() and MakeABidToAllReplications() methods, where each node has its own replication manager and they send out bids to other replication managers, but they never communicate back to each other. This can only be done because we assume that our network has reliable, ordered message transport, where transmissions to non-failed nodes complete within a known time-limit, as stated in the hand in description. Each nodes and its replication manager only cares about itself and it's own values. Bids and values are only updated when a replication manager receives a message to change them from another replication manager,

4 Correctness

4.1 Correctness 1

in terms of consistency, we can quickly rule out linearizabiliy, simply because we are using active replication and our program does not implement synchronised clocks, to make sure the real time ordering of the operations.

When it comes to sequential consistency, this is mainly achieved through the use of locks in each of our active replications. Every time a bid is made to every active replication, it locks the entire method, making sure no other call to bid accesses the variables, then it checks whether the bid is higher than the previous, and updates it if its a new highest bid. This ultimately makes sure that we satisfy the sequential requirement of a client that should always reads the most recent written value, regardless of clients, thereby preserving the sequential ordering of the executions, and making sure no higher value is overwritten by a lower value. The same of course happens in the read function to prevent any client from reading old values. On top of this, when a client reads the bid values from different replications, it will always read the highest value in any of them, since that would be the highest bid, thereby preventing it from accedently reading a value from a replication that had reconnected to the system but lost its values. Although this is also based upon the fact that we assume a network with reliable and ordered messages.

4.2 Correctness 2

Our protocol runs mainly on two rpc calls, MakeABid and GetResult. In the absence of faliures and assuming the network is reliable and transmissions complete within a known time-limit, we can argue that both should work, since the network should be reliable.

in the presence of failures, again assuming the network is reliable and transmissions complete within a known time-limit, we have for all rpc calls setup a context with timeout, where it will cancel the method call after 2 seconds of no-response, returning an error with the message "failed to get result with error: "error message"". Even though one server has crashed, these calls will still return the values from the remaining active replications, we achieve this by using concurrent go functions that each make a call to a unique server, such that even if only one replication return a true value, and the rest return timeout errors, a value will be returned (thereby making our system resilient to n-1 crashes, n being the amount of servers/replications)