Imperial College London

Distributed Algorithms Coursework

Imperial College London

Department of Computing

Multi-Paxos

Authors:
Daniel Loney (dl4317)
Ossama Chaib (oc3317)

February 21, 2021

1 Design and Implementation

The primary method for implementation was a line by line recreation of the pseudocode given in the paper adjusted so that it was able to fit with Elixir syntax and features. Processes in the paper such as *Acceptor, Scout, Commander, Leader* and *Replica* were implemented as Elixir modules.

A number of other modules were also used from the given code or implemented. A primary module, *Multipaxos* was used as the top level node to spawn all other processes. *Multipaxos* would spawn a set number of *Servers* and *Clients*. Each server would spawn its own *Database, Replica, Leader* and *Acceptor* processes. *Server* would then pass its own replica and leader back. *Multipaxos* would then pass a list of all leaders to all the replicas and a list of all acceptors and replicas to all leaders so that all the appropriate processes would be available for use when implementing the pseudocode for *Replica* and *Leader*. *Multipaxos* also spawns a *Monitor* process. *Monitor* allows for additional monitoring of transactions and messages sent and received between processes for debugging purposes.

Client would make transaction requests between two accounts of random amounts along a regular interval of time. Client requests would be made to the replicas. The client would then stop sending requests after a set number of maximum requests. *Client* would also keep track of replies from *Replica*. *Replica* state updates, as they are referred in the paper, are reflected in the *Database* module. Commands that are decided to be executed by *Replica* are issued and are updated in the *Database* module. The *Database* module also notifies *Monitor* of state changes.

A full diagram of all process spawning and messages sent can be seen in Figure 2.

When implementing modules with given pseudocode, there were a number of differences between the syntax of the pseudocode and the syntax and features of Elixir. The most notable difference between the pseudocode and Elixir are the functional characteristics of Elixir and the assumption of mutable variables in the pseudocode. The functional characteristics of Elixir required a number of adaptations. Mutable variable updates in the pseudocode had to be reflected by carrying variables around in the parameters of functions and making value updates in the values of function parameters as opposed to variable updates. Looping, notably endless loops, had to be accounted for via use of recursion. One difference was that recursion had to be explicitly considered under every control flow, where such consideration was not as explicit in the pseudocode. Elixir's *send* and *receive* features between processes was used to implement sending and receiving messages between processes in the pseudocode. Syntax for sending and receiving is very similar between the pseudocode and Elixir so it was not particularly challenging to implement.

To account for a livelocking scenario, we implemented random process sleep times so that eventually the system would exit the livelock. Random process sleep times ensures the end of a livelock since the times that adoption happens would have some randomisation and eventually reach a threshold where a proposal is able to be adopted by a majority and decided upon.

Figure 1: Screenshot of replica logs in terminal.

2 Debugging and Testing

Due to the nature of programming, the programmer who writes code is much more familiar with their own code but also is less able to judge the correctness of their own code. Our initial step in debugging thus relied on having each partner thoroughly inspect, read and compare the written Elixir implementation and the pseudocode in the given paper. By taking turns thoroughly analyzing each partner's Elixir implementations, line by line, many obvious bugs could be detected and fixed without having to go through any more complicated debugging process.

Further debugging relied on using the function IO.puts to provide a description on what had been sent or received by a process and monitor the various connections that were being made. At a certain point the volume of messages that were being printed became intensive and as a result the console output was instead saved for later viewing of specific runs. We then opted to use the Monitor process to accurately and succinctly monitor various sub-process spawning/ending and requests and updates made by these processes.

A major issue that we encountered when attempting to run the experiments was that the client would send a client_request message to N=5 replicas with distinctly different process id's of the form; #PID<16780.139.0> however the only replica that would receive this client request would have a process id of the form #PID<0.139.0>. This meant that this was the only replica that would carry out the necessary actions to maintain the multi-paxos system. We found this issue initially by assessing the monitor logs and realising that there were much fewer commanders and scouts being spawned for configurations with larger requests and accounts. We also realised that the database updates were only done in the first 3 seconds of execution and would remain constant afterwards, this is due to only a single request executing their database updates upon receiving a decision. We then used various print functionality in Elixir to deduce that only a single replica had been receiving every client request. Additionally every server would be spawning the same replica with process id; #PID<0.139.0>. When inspecting the multipaxos.ex file and examining the communications between different processes we noticed that the multipaxos process sends a bind with leaders to a list of unique replicas however only a single replica (#PID<0.139.0>) would receive that bind, this replica was not in the list of replicas that we send the bind to. This can be seen in Figure 1. We were unable to solve this issue and as a result our resulting implementation is limited.

3 Experiments and Potential Findings

If the implementation were suitable to run experiments on, there are a number of variables that could have been altered to observe results. The first variables that could have been changed were the number of servers and the number of clients. It is very likely that increasing the number of clients would have resulted in a lower throughput since the same system would need to handle more client requests and it would take a longer amount of time for a majority decision to be reached while handling multiple client requests simultaneously, since multiple requests could preempt one another and require new cycles of decision making. Another possible limit could have been the time it would take to process many proposals.

The second most straightforward variable to change would have been in the number of servers, hence the number of replicas, leaders and acceptors. An increased number of servers caused an earlier livelock situation since more proposers and decision makers would cause a higher occurrence of preemption and acceptance between phases. The increased number of preemption and acceptance would also increase the chances that the system could enter a livelock. With livelock prevention, a possible outcome could have been that the transaction throughput would still decrease due to the requirement of more scouts and commanders needing to adopt or decide to reach a quorum.

Another available variable to change was the quorum number of replicas that issue proposals to leaders. In a broadcast situation where all replicas are notified of a new client request and its proposal, the speed at which the proposal is adopted and decided upon could be slower, since all proposals need to compete and could possibly preempt each other when there are a high number of client requests. When client requests and proposals are sent to the smallest majority of servers, a good prediction is that the adoption and decision rate would increase since there would be fewer competing proposals that could possibly preempt one another. Round-robin would be the fastest way of proposing, where a proposal is passed between one replica at a time until it has reached all. In the condition of preemption, at most nearly all the replicas would have to be notified, which is better than a majority or a broadcast situation. Round-robin proposals however present the risk that if a replica fails while its leader is in between phases, there is no way to recover the client request.

4 Diagrams

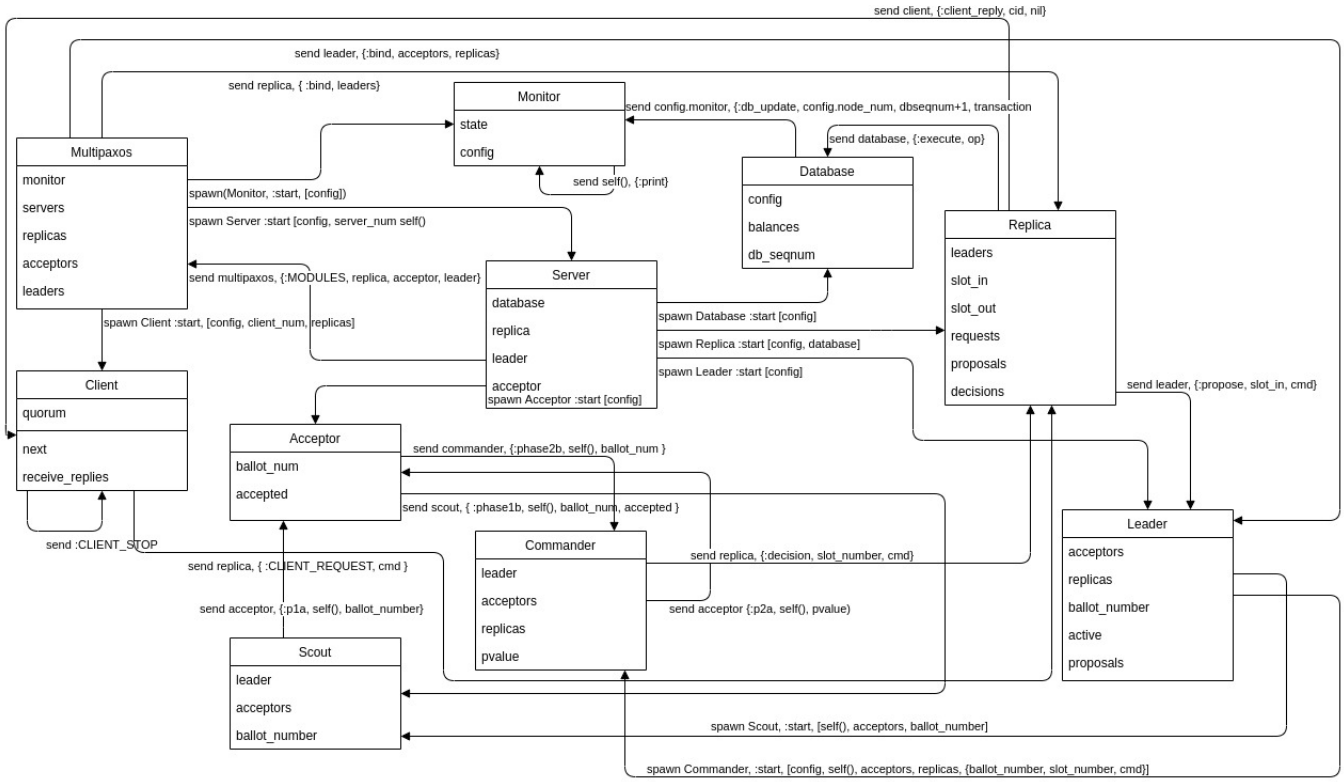

Figure 2: Flow diagram of all spawning and message sending between processes.

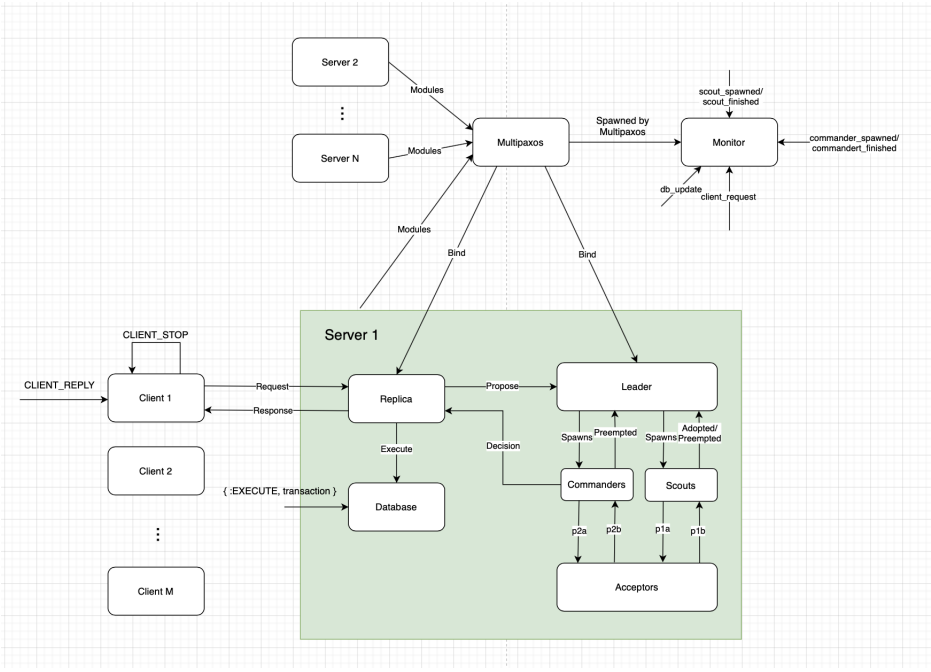

Figure 3: Second diagram of all spawning and message sending between processes.