Multi-Paxos

Kazuya Kai-Olowu (kyk3218)

February 16, 2021

1 Multi-Paxos

This report will cover the implementation of the multi-paxos protocol, closely following the 'Paxos Made Moderately Complex' paper by Robbert van Renesse and Deniz Altinbüken. The implementation used Elixir, taking advantage of its support for building distributed, fault-tolerant systems. This report will address design and implementation choices, approaches to testing and debugging, and experimental results and findings with the final implementation.

1.1 Project Structure Overview

The Multi-Paxos structure builds on the foundation provided already in the skeleton code. It also follows the `Replica`, `Leader`, `Scout`, `Commander` and `Acceptor` modularity of the original paper by Robbert van Renesse and Deniz Altinbüken.

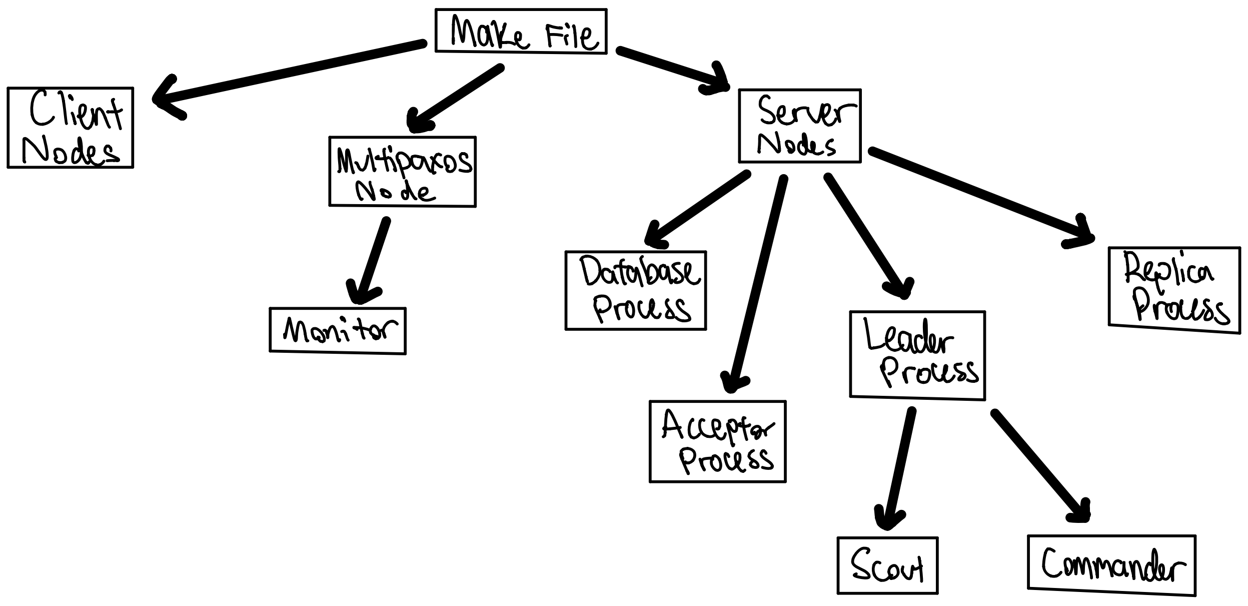

Figure 1: Spawning architecture for nodes and processes.

2 Design and Implementation

The design and implementation of Paxos was based on the pseudo code and ideas provided by Robbert van Renesse and Deniz Altinbüken in the paper "Paxos Made Moderately Complex".

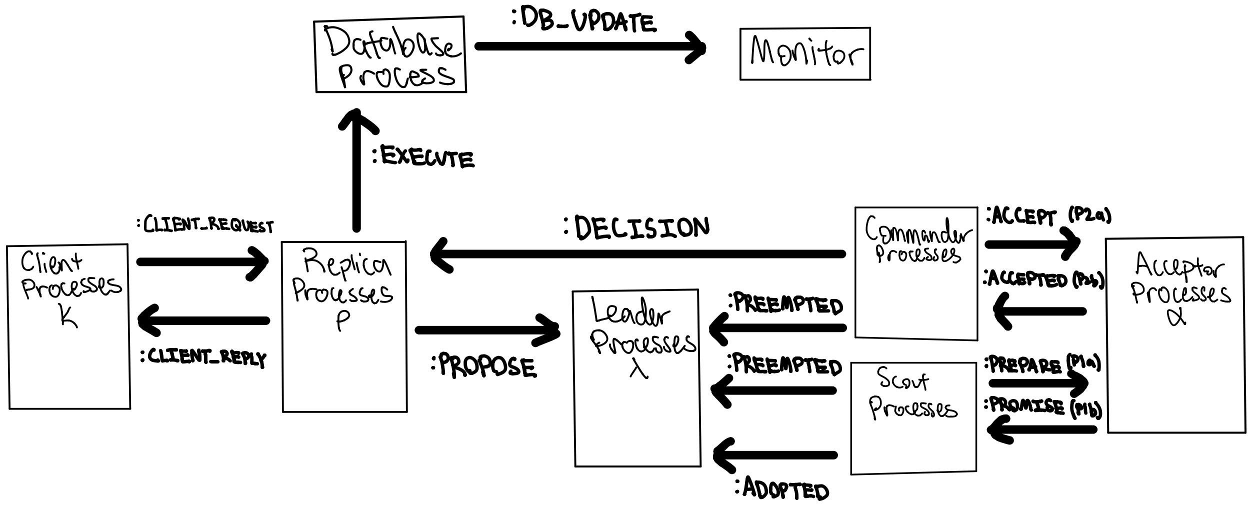

Figure 2: Process communication pathways.

2.1 Elixir

Having to essentially translate the pseudo code to Elixir was difficult due to Elixir's special traits, such as its functional paradigm, its lexical scoping language, and its syntax; but after some research combined with trial and error, the correct functionality was achieved. One example being the recursive call of `defp next` in order to mimic the `for ever` functionality due to the lack of access to a `while` loop. Another exmaple being the use of `defstruct` in `Replica` in order to contain the module's state. This not only allowed maps and list fields required by the protocol such as `proposals` to persist, but also cleaned up the tail recursion of `defp next`, making it more concise and readable.

2.2 Live-Locking

To solve the live-locking, random sleep, exponential back-off, and exponential back-off with the bully algorithm was used. These methods all prevent the scenario where multiple leaders are concurrently fighting for leadership, preventing any leader from being able to acquire leadership.

2.2.1 Leadership Election Strategies

- :none - This is the default configuration, meaning there is no live-locking solutions.
- :randomised_backoff - With this configuration, upon receiving a `:PREEMPTED` message with a greater ballot number than its own, the leader process will proceed to sleep for a random time in a certain range before sending out a new `Scout` with a ballot number one greater than what it received.
- :exponential_backoff - Same as the randomised backoff, the difference being that the time period the leader process would sleep is accumulated, increasing exponentially.
- :exponential_bully_backoff - Same as exponential backoff but with an additional condition regarding the process identifiers of the leader process and the process that caused the `:PREEMPTED` message to be sent in the first place. The bully algorithm solves live-locking using the global ordering of the leaders using their identifiers, where the larger identifier is always given priority in a live-locked scenario, preventing the possibility of live-locking.

3 Debugging and Testing Methodology

Prior to starting the implementation of the required modules, it was important to think about the approach to testing and debugging that would be taken; this is because it would aid in reducing the impact of erroneous code, speeding up the debugging process. The provided `Monitor` and `Debug` modules were studied and understood to know what was required in all the modules to be implemented.

3.1 Debugging Methodology

Rather than scattering `IO.puts "debug message"` directly in the module code to debug, debugging functions from the `Debug` module were called. An advantage of this is the ability to easily make alterations to the debug code in once place rather than scattered all over the code base. The debug functions held a `verbose` level, where if the current `debug_level` set in the `Makefile` is greater or equal to the `verbose` level, the debug message would be printed. Having several granularities of debug messages was essential in the debugging process as at the lowest level, the debug messages would be too numerous to easily find the important debug messages. For example, setting `debug_level` to 2 would only show the messages exchanged between modules; meanwhile setting `debug_level` to 4 would show everything down to the contents of list, sets, maps, etc

3.1.1 Debug levels

- 0: Monitor messages.
- 1: Sending and receiving of client requests.
- 2: Sending and receiving of messages between processes.
- 3: Spawning of different processes.
- 4: Contents of lists, sets, maps and other low level details.

3.2 Monitor

The `Monitor` module keeps a record of the number of `Scout` and `Commander` processes that have spawned and exited by receiving `(module)_SPAWN`, `(module)_FINISHED` messages from the `Leader`, `Scout`, and `Commander`. A personal addition to the Monitor module was the inclusion of two new monitor features, the most recent time slept, and the total time slept by each server; both implemented through a `Commander`. This was quintessential to understanding and debugging the behaviours of the system under various configurations; refer to the experiments 4.3, 4.6, and 4.7

3.3 Testing

The code was developed iteratively, testing and verifying our code in each iteration comparing debug and monitor function outputs to what was expected. Sanity tests were conducted with smaller numbers of servers and clients in order to ensure that the flow of messages between each module was correct; with a smaller number of servers and clients it was easier to see the sending and receiving of messages.

Individual testing was done on functions by putting in test values and ensuring it has the intended behaviour. One example being inside the `:ADOPTED` receive in leader, where the proposals map needed to be merged with the pvalues list after it had been made unique with the slot number as the key, solving duplicates by only keeping those with the highest ballot number.

To test if the Multi-Paxos system live-locked, that hyper parameters of 3 to 5 servers and 1 to 3 clients were experimented with. Refer our live-locking experiments results below.

4 Experiments

The system was experimented with under several different configurations to test whether consensus is ultimately reached. The output logs of the various experiments can be found in the `outputs` folder. By default, the system was ran with the configuration of 5 servers, 5 clients, a broadcast client send strategy, 50,000 maximum requests per clients, run time of 15,000ms, and no live-locking solution strategies; any specific alterations to this are describe in the file name of each output log file.

Experimental Environment:

- OS: Ubuntu 20.04.2 LTS
- Processor/Cores: Intel Core i7-6700K CPU @ 4.00GHz $\times$ 8
- Memory: 62.8 GiB

4.1 default_5_server_5_client

This was the experiment with all the default configurations. The behaviour observed is that after each server is able to complete a small number of database updates, the servers become live-locked, thus unable to proceed further for the rest of the run time. What is expected is that eventually the live-lock would be momentarily broken just by random chance if ran long enough; this means that if ran long enough, all client request would eventually go through.

4.2 default_2_server_5_client

This experiment reduces the number of servers to 2, and what is observed is that the servers are able to break out of live-locked scenarios much more often as there are now only 2 servers contesting each other, so more database updates are possible. The issue with this system configuration is that it cannot tolerate any server failures as Paxos requires a minimum of 2m+1 servers to tolerate m failures.

4.3 server_1_crash_default_2_server_5_client

This experiment confirms the requirement of minimum of 2m+1 servers to tolerate m failures. Here we have 2 servers, but a minimum of 3 is required to tolerate a single failure, and as expected when server 1 crashes, no more database updates are possible.

4.4 randomised_backoff.txt

This experiment tests the impact of randomised backoff in solving the live-locking. What is observed is that all servers are able to make progress without live-locking as seen by the database updates done. As seen in the total sever sleep times, they are relatively uniform; and a inversely proportional relationship between the total sleep time and the number of commanders spawned can be observed.

4.5 random_round_robin.txt and random_quorum.txt

These experiment were conducted to test the impact of using round robin and quorum compared to the default broadcast client send strategy. The round robin client send strategy involves only sending each client request to a single replica, and rotating through all the replicas for different client requests. The quorum client send strategy involves sending each client request to only half of the replicas. As seen in Figure 3, the round robin client send strategy completed a larger number of database updates compared to the other two. A possible explanation of this behaviour is that broadcast would lead to a significant number of messages to need to be ignored, as a single client request is sent to all replicas but only one of them will actually go through all the way, so is not very efficient/optimal.

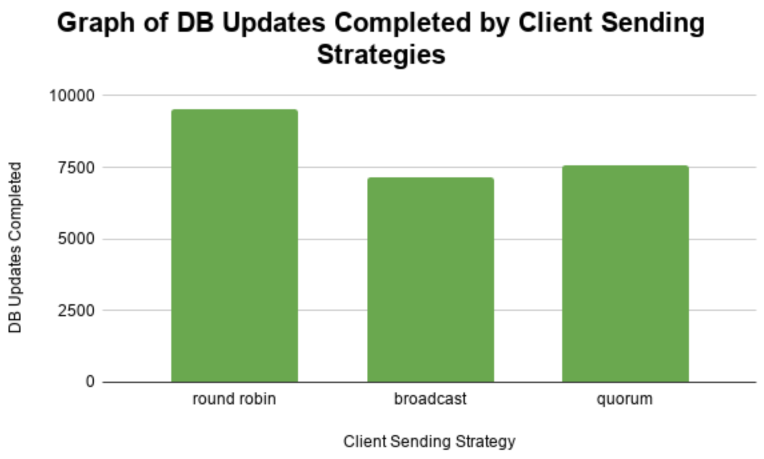

Figure 3: Client Send Strategies.

4.6 server_1_crash_random_backoff

This experiment tests if consensus is still maintained if one server crashes. What is observed is that even after server 1 crashed, consensus is maintained and database updates are continuously made. In theory, the system with 5 servers can maintain consensus with up to 2 failures.

4.7 3_server_crash_random_backoff

This experiment is the same as the previous, but explores what happen if three out of the total five servers crash, as in theory, this the system can only tolerate 2 failures as discussed above; and as expected, once the third server crashes, processes are no longer able to achieve majority, so the system fails and no more database updates can be made.

4.8 exponential_backoff

This experiment tests the impact of using exponential backoff to solve the live-locking. The behaviour observed is that initially the database updates are smoothly processed by all servers; however, as the backoff time exponentially increases, the rate at which the database updates decreases exponentially too. The long sleep times can be seen in the total server timeouts at the end reaching up to 19,221 ms. One possible improvement that I would have implemented given more time would be to reset the sleep time to either some constant, or to divide the sleep time by 2 if it is able to go through; this would stop the sleep times from reaching to ineffective levels.

4.9 exponential_bully_backoff

This experiment tests the impact of using exponential backoff with the bully algorithm to solve live-locking. What was discovered was that as server 5 always has the greatest process identifier, it is never put to sleep as seen by the lack server 5 in the total sleep time field in the `Monitor` logs. This did not have as positive of an effect as expected where it essentially means only server 5 will ever reach majority and complete its request, as seen by its significantly higher commander count. Meanwhile, server 2 only had a commander count of only 5, thus was rarely if ever able to process its own requests.