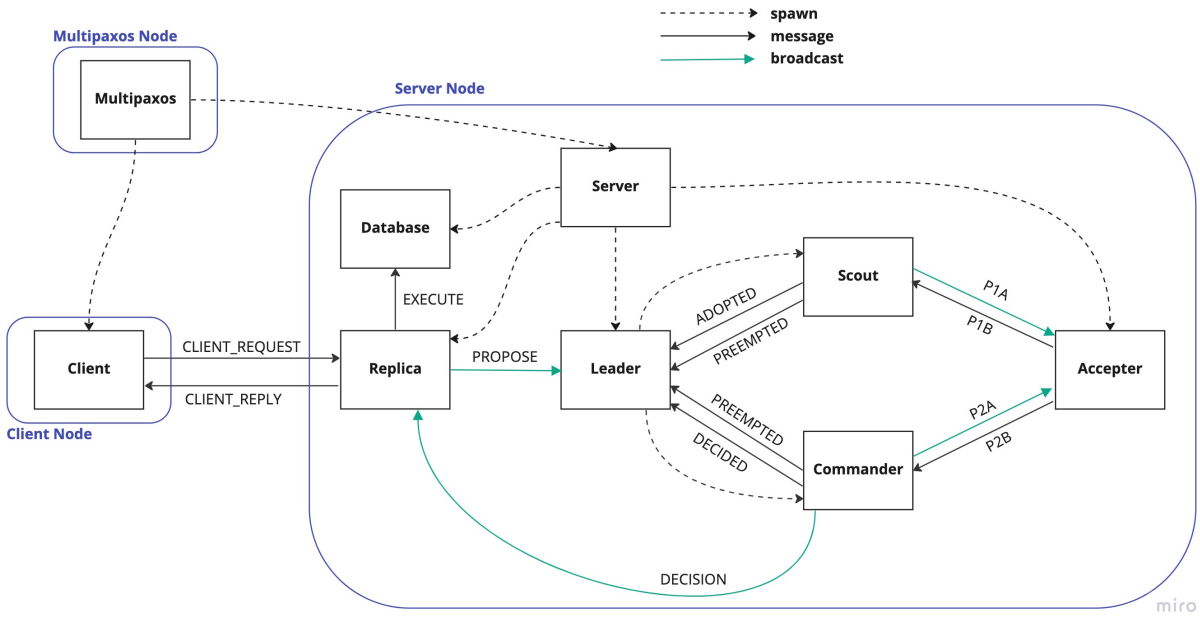
Multipaxos Coursework

Justine Khoo (jnk20)

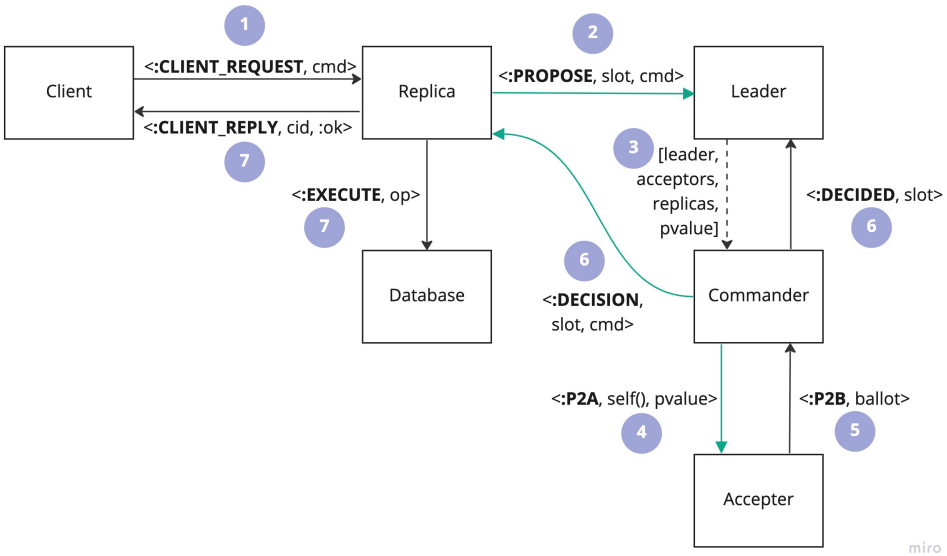
February 27, 2023

1 Architecture



This diagram shows the overall structure of the Multipaxos consensus algorithm. Messages with a solid black arrow are unicast, except for `CLIENT_REQUEST` which depends on the send_policy. Green arrows are broadcast messages, sent to the receiver process in all nodes, though only one server node is pictured. Dotted arrows indicate that a process spawned another. Each replica knows all the leaders and each leader knows all the acceptors and replicas. Each client knows all the replicas.

The flow of messages for a typical client request would be as follows: The client sends a `:CLIENT_REQUEST` message containing a command to a number of replicas. The replicas then broadcast a `:PROPOSE` message with the command and a slot number chosen by the replica, to all leaders. When active, the leader spawns a commander, sending a pvalue of the command, slot, and its current ballot number. The commander broadcasts a `:P2A` message containing the pvalue to all acceptors. Acceptors send back a `:P2B` message with their own ballot number. Once the commander has received `:P2B` messages from a majority of acceptors, with ballot numbers equal to its own, it broadcasts a `:DECISION` message to all replicas, containing the chosen command and slot. It also informs its own leader that the slot has been decided. Finally, the replica sends an `:EXECUTE` message to its database and a `:CLIENT_REPLY` to the client which sent the request.



2 Liveness

2.1 AIMD timeout

The TCP-like approach uses exponential increase of timeouts to prevent repeated over-contention between leaders and allow one leader to be successful. A leader decreases its timeout when one of its commanders notifies it that its proposal has been chosen. When it gets preempted by a higher ballot number, it sleeps for the current timeout duration and then increases its timeout by the multiplier.

```
{ :DECIDED, slot } ->
    self |> decided(slot)
         |> timeout(:decrease)
         |> next()

{ :PREEMPTED, {round, _} = curr_ballot } ->
    if curr_ballot > self.ballot do
      Process.sleep(ceil(self.timeout))
      new_ballot = { round + 1, self.config.node_num }
      spawn(Scout, :start, [self.config, self(), self.acceptors, new_ballot])
      self |> active(false)
           |> timeout(:increase)
           |> ballot(new_ballot)
           |> next()
    ...
```

This helper function implements the AIMD timeout and makes the rest of the code very clean and readable. The initial value, multiplier, and decrease are set to 10ms, 2, and 2ms respectively and can be changed in the configuration.

```
defp timeout(self, mode) do
    value = case mode do
      :increase -> min((self.timeout + 1) * self.config.timeout_mult, self.config.max_timeout)
      :decrease -> max(self.timeout - self.config.timeout_decr, 0)
    end
    Map.put(self, :timeout, value)
end
```

2.2 How timeout values affect performance

I experimented with various values for the timeout multiplier, with 7 servers to better see the effects of contention. As the multiplier increased, the time taken to finish executing all 2500 requests generally decreased, because it was more likely that one server would take over the consensus protocol and spawn nearly all the commanders while the others were in timeout almost all the time, reducing the time spent in livelock. Also, the higher the multiplier, the lower the maximum ballot number reached, showing less contention. However, even with `timeout_mult=2` leaders can get stuck in a livelock but this usually ends after a few seconds. With a lower multiplier, there was a more even distribution of load among the servers. The code was run with `DEBUG=2` which prints whenever a leader is preempted, showing the ballot numbers and current timeout. The run with `timeout_mult=1` is not the same as a timeout-free implementation as the timeout still gets incremented by 1ms with every preempt. These results are the average of 5 runs. **Output files: 00-04**

Timeout multiplier	Time taken / s	Highest ballot number	PARAMS
2	2.2	33.0	
1.5	3.0	93.0	mult15
1.2	3.0	113.2	mult12
1.1	2.8	133.0	mult11
1	4.0	221.2	mult1

When timeouts are turned off, the program livelocks. The log can be seen in `timeout_none.txt`, with parameter `no_timeout`. After 14s, the program had spawned over 250000 commanders and reached ballot number 575 but only managed to execute 2078/2500 requests. If left to run longer, it might continue indefinitely. **Output file: 05**

2.3 Considerations

I did not implement pinging as the AIMD timeout mechanism was sufficient to ensure liveness and I felt it would over-complicate the design. If a process crashed, the other leaders would eventually finish their timeout and become active. The downside is if a process crashes quite late (when other process's timeouts have grown exponentially large) and it is currently the only active process, it would take quite a while for the others to wake up, adding long delays to the system. To mitigate this, I added a maximum timeout which can be adjusted in the configuration. Going by a rough feel, I think a value of 1000ms is a good trade-off between avoiding contention and preventing long delays in the event where the currently active leader crashes. Note that all readings recorded in this report were run without a maximum timeout.

I also think decreasing the timeout by 2ms each time is too small and has negligible effect on a leader which has reached large timeout value. However, making the decrease amount bigger could overcome the multiplicative increase at the start, when timeouts are still low and contention is the greatest, defeating the purpose of the AIMD timeout.

3 Evaluation

The program was tested on a Macbook Pro with a 2GHz Intel i5 processor with 4 cores and 16GB RAM. The machine was not running anything else intensive at the time. All results shown in tables are the average of 5 runs, whose logs can be inspected in the outputs folder. I mainly used the following metrics to measure performance:

- Time taken: The number of seconds taken for all servers to execute requests sent by all clients
- Commanders spawned: A run that spawns less commander processes is preferred as it is less CPU intensive

3.1 Number of requests

I varied the number of requests sent by each client, from 500 to 4000, with 5 clients and 5 servers. The time taken to complete and the number of commanders spawned increased somewhat linearly which is expected. The parameter used is `request_n` where n is the number of requests, for example `PARAMS=request_2500`. **Output files: 10-17**

Number of requests	Time taken / s	Commanders spawned
500	2.0	16848.2
1000	4.0	19052.2
1500	7.8	29256.4
2000	12.9	39393.8
2500	19.6	53998.2
3000	24.7	60129.8
3500	40.6	65747.2
4000	47.0	78625.6

3.2 Number of servers

I tested the program with different numbers of servers from 1 to 10, with 5 clients sending 500 requests each. I expected a more noticeable increase in the time taken to execute all 2500 requests as there should have been more contention. The difference could be more noticeable by increasing the number of requests or printing more often. Unsurprisingly, when there was only 1 server the number of commanders was exactly the number of requests as there was no preemption and a leader spawns one commander per command. The average number of commanders spawned actually decreases after 2 servers. Due to the AIMD timeout, most of the time there are 1 or 2 dominant servers creating most of the proposals while the rest are dormant, therefore the number of commanders does not increase proportionally to the number of servers. **Output files: 20-29**

Number of servers	Time taken / s	Commanders spawned	Avg commanders per server
1	2.0	2500	2500
2	2.0	7973	3986.5
3	2.0	11782.4	3927.5
4	2.0	12953.8	3238.5
5	2.0	12583	2516.6
6	2.0	14684.2	2447.4
7	2.3	14275.2	2039.3
8	2.2	18956.4	2369.6
9	2.5	22061.4	2451.3
10	2.7	21698.4	2169.8

3.3 send_policy and crashed servers

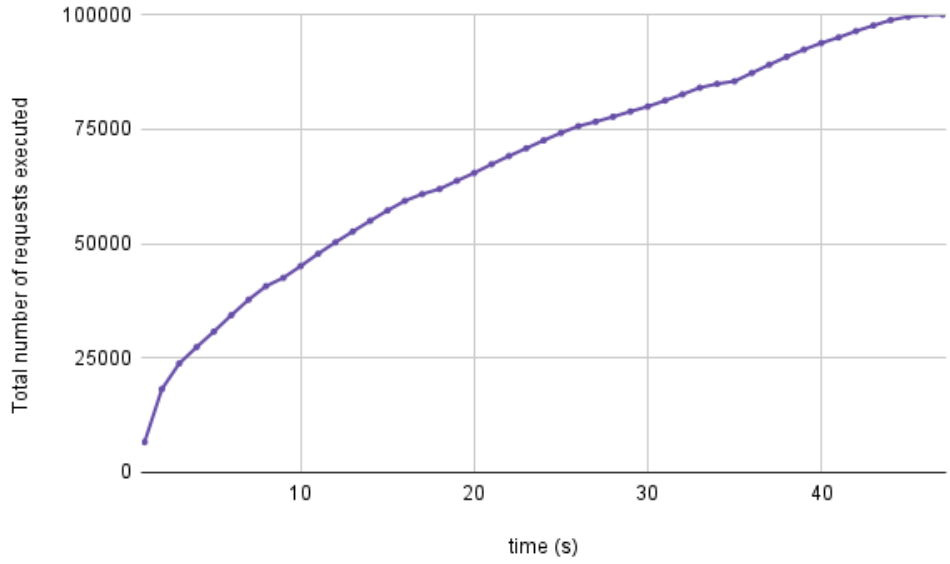
I ran the program with 3/7 faulty servers, which gave expected results. The round robin policy was not sufficient to ensure all 2500 requests were being executed as each request was only sent to one server, which could be faulty. Quorum only sends requests to a majority of replicas while broadcast sends to all replicas. Both ensured all requests were executed, with quorum having better performance than broadcast. Quorum was only sufficient as long as a minority of servers crashed. If at least half of the servers crash, not all requests get delivered. **Output files: 30-33**

Send policy	Time taken / s	Commanders spawned	PARAMS
Quorum	4.3	39574.4	quorum_crash
Broadcast	6.0	66928.2	broadcast_crash
	Requests executed		
Round robin	2157.8	13730.6	crash3

3.4 Observations

In many runs, the number of commanders spawned would spike significantly even after all commands have been executed. This makes sense as all leaders need to complete their proposals even though other leaders have gotten a decision for the same slot and command and notified the replicas. This could have affected the accuracy of my results as it is possible I did not run the program long enough to count all commanders, though I did try to allocate enough time for a buffer of at least 5 seconds after all requests finished executing. Before the spike though, if you took readings just when all servers had finished executing requests, there would be only a couple of servers with a 4-digit commander count while the rest had a 1 or 2-digit count, showing the effects of the exponential timeout.

Also, the rate of requests executed decreases with time. This graph shows the total number of requests executed by all databases against time. I suspect this is because multiple leaders make duplicate proposals and hence decisions, and replicas only forward the first instance of a command to databases. But if we were to plot the total number of decisions made, it should grow somewhat linearly. **Output file: 40**



4 Appendix

Following Section 4.1 of the specification, I modified the acceptors to only store the latest ballot number and command for each slot. This required minimal changes due to the requirement that accceptors only add pvalues to `accepted` if its ballot number equals the highest ballot number it has seen. `accepted` was changed from a set of triples to a map from slot to ballot number and command. This also simplified the computation of the updated proposals in the leader when a new ballot number was adopted.

I also added my own optimisation where the leader keeps track of all slots for which it had a successful proposal. When a commander receives a majority of `:P2B` replies from acceptors with ballot number equal to its own, it also sends a `:DECIDED, slot` message to the leader. When a ballot number is adopted, the

leader only spawns commanders for slots which are not in decided. This reduced the number of commanders spawned by about half. It is not as strong as the garbage collection idea in Section 4.2, where replicas periodically inform leaders and acceptors about their slot_out value, allowing leaders and acceptors to garbage collect all information about lower slots, but it is far simpler to implement. All the tests done so far have been with the optimisation, but it can be turned off with `PARAMS=no_opt`. Without the optimisation, the average number of commanders spawned across 5 runs was about 4 times as much as `request_500` or `servers_5` which have the same parameters otherwise. **Output file: 50**