Multi-Paxos Report

Daniel Simols (ds1920) and Benson Zhou (bz620)

February 27, 2023

1 Architecture

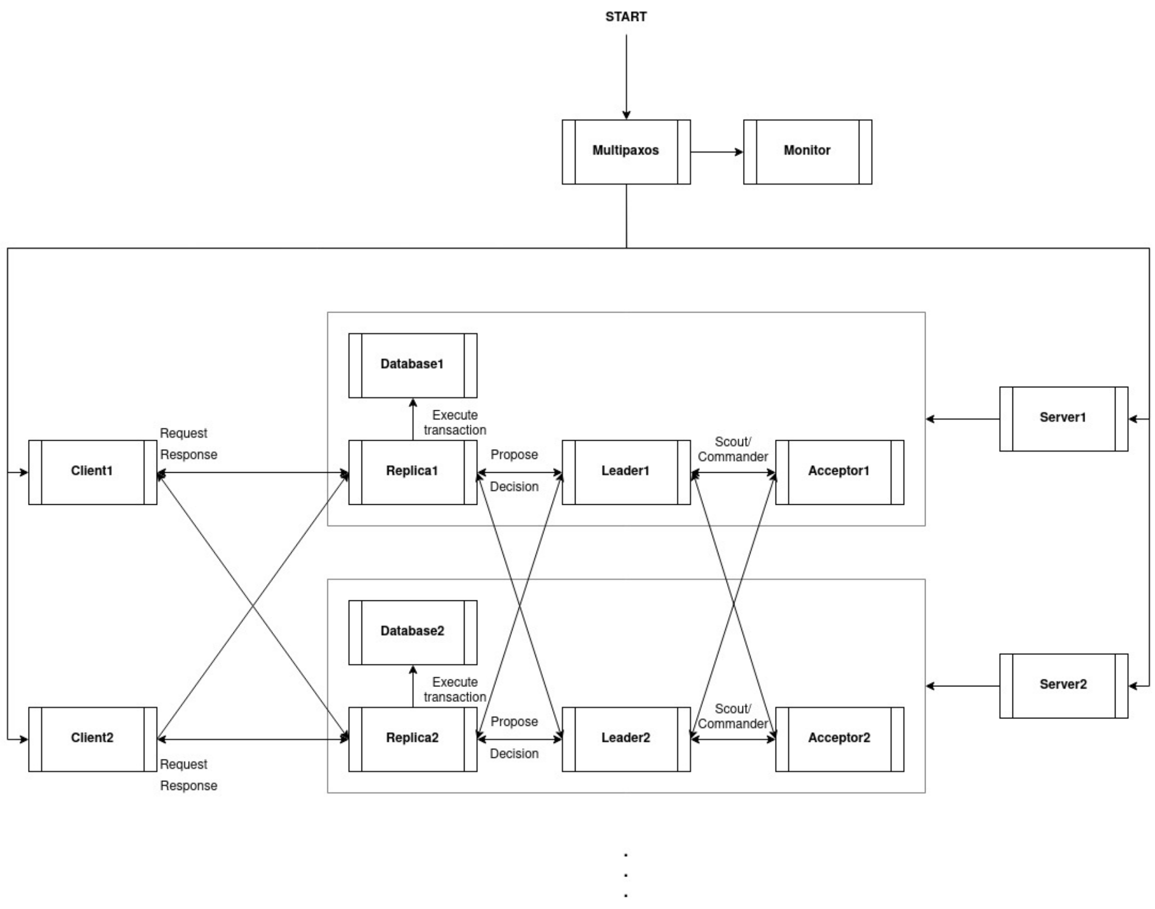
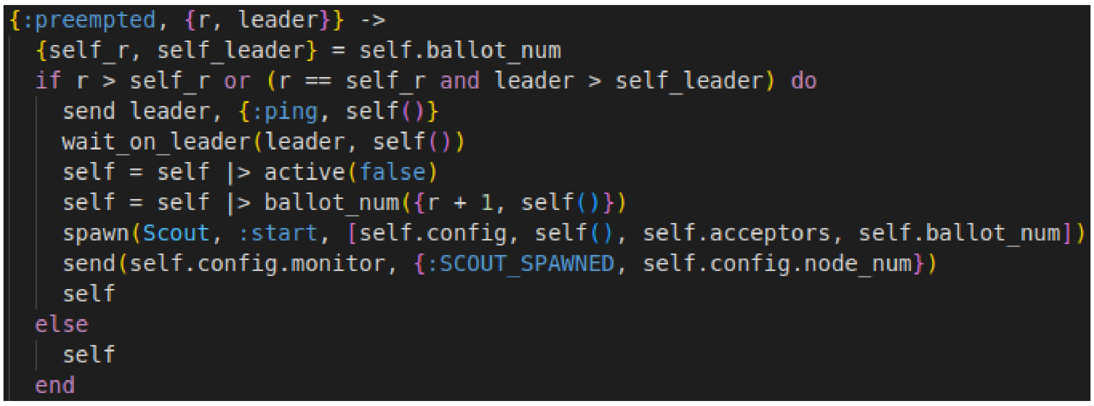


Figure 1: Structural overview of Multi-Paxos.

In the diagram, Multipaxos spawns 'N' Clients and Servers. Each Server spawns a Replica, Database, Leader, and Acceptor. Any one client sends its request to all spawned Replicas, any one Replica sends a proposal to all spawned Leaders, and any one Leader can spawn a Scout or some Commanders to talk to each spawned Acceptor.

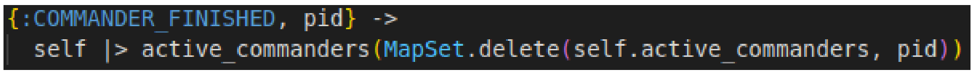
2 Liveness

2.1 Elixir Snippets



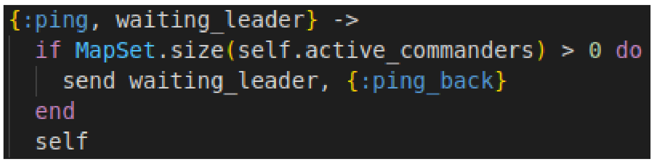
```elixir
{:preempted, {r, leader}} ->
  {self_r, self_leader} = self.ballot_num
  if r > self_r or (r == self_r and leader > self_leader) do
    send leader, {:ping, self()}
    wait_on_leader(leader, self())
    self = self |> active(false)
    self = self |> ballot_num({r + 1, self()})
    spawn(Scout, :start, [self.config, self(), self.acceptors, self.ballot_num])
    send(self.config.monitor, {:SCOUT_SPAWNED, self.config.node_num})
    self
  else
    self
  end
```

Figure 2: Leader is preempted.



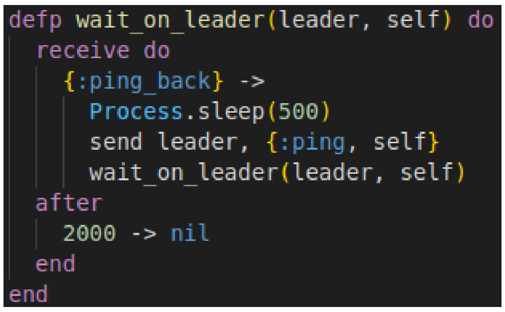
```elixir
{:COMMANDER_FINISHED, pid} ->
  self |> active_commanders(MapSet.delete(self.active_commanders, pid))
```

Figure 3: Leader is informed of commander finishing.



```elixir
{:ping, waiting_leader} ->
  if MapSet.size(self.active_commanders) > 0 do
    send waiting_leader, {:ping_back}
  end
  self
```

Figure 4: Leader is pinged to see if it's executing instructions.



```elixir
defp wait_on_leader(leader, self) do
  receive do
    {:ping_back} ->
      Process.sleep(500)
      send leader, {:ping, self}
      wait_on_leader(leader, self)
  after
    2000 -> nil
  end
end
```

Figure 5: Wait some time and pings the other leader again. Returns if timeout is elapsed.

2.2 Explanation

Each leader stores a list of active commanders it has spawned. Whenever a leader spawns a new commander, it is added to the list. Whenever a commander is about to exit, it notifies its leader, which then removes it from active commanders list.

If a leader L1 receives a preempted message about another leader L2 with a higher ballot number, L1 pings L2 to see if L2 has any active commanders in its list. If L2 does, L1 keeps waiting. Once L2's commander list is empty, L1 sends a scout and attempts to take control and create its own commanders. This implementation means, once L2 receives a ping, it only sends a ping back if L2's active commanders list is non-empty. If the list is empty, L2 simply doesn't send a ping back. After some reasonable timeout, L1 and other leader who pings L2 assume L2 is either finished or has gone faulty. They then spawn a scout with a new, higher ballot number.

3 Evaluation

3.1 Experimental Set-Up

OS: Ubuntu 22.04.1 LTS

Linux Kernel : 5.15.0-56-generic

Processor: Intel Core i7-6700 CPU @ 3.40GHz

Cores: 4

Memory: 16 GiB

3.2 Findings

We found that the timeout and the time spent in between pings affects the performance significantly. A timeout of 2000ms and sleeping 500ms in between pings yielded good performance for the default configuration. We were able to process all 2500 requests within 4-5 seconds. When the timeout and sleep time is reduced too much, the processing speed slows down instead. A timeout of 100ms causes the program to stall after processing around 2000 requests. We theorise the reason is context switching takes up too much of the timeout, meaning leaders do not have enough time to respond. Also, as the set of `decisions` grow, it takes longer to find a given slot in the set, increasing the processing time and the risk of breaching the timeout limit.

3.2.1 High Message Load

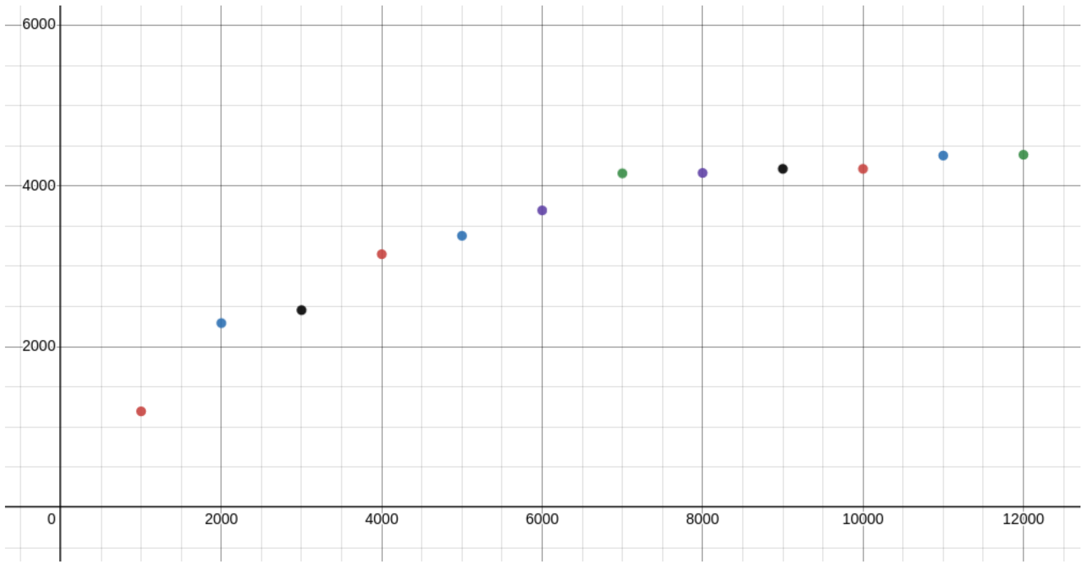


Figure 6: X-axis is time, while Y-axis is number of commands executed by a replica. This is a plot of file 01 in outputs/

We change the `max_requests` value in `configuration.ex` to 5000, meaning 5 clients will now send 5000 requests each, totalling at 25000 requests.

After 7 seconds, we have received 11105 requests, and each replica processed 4157 of them. The speed of the program slows down significantly afterwards, being only able to process 242 more requests in the next 7 seconds. We receive all 25000 requests at 17 seconds, but afterwards, we were only able to process around 10 requests per second, much slower than around 1000 requests per second in the first 3 seconds of the program.

We theorise the performance difference between seconds 1-3 and seconds 17-30, is the result of the set of `decisions` growing in each replica, as time goes on. Each replica used the set `decisions` to keep track of the commands that have been executed in old slots, so that it no longer process a command for a slot that has already been decided. Since old slots cannot be deleted, finding a given slot number in the set becomes more and more time-consuming, due to the size increase of the set. This means the speed at which the replica processes requests is significantly reduced.

The full output for this experiment can be found at `outputs/01-high_message_load.txt`

3.2.2 Different Send Policies

When using `quorum` policy in `configuration.ex`, we process around 2000 requests in the first 7 seconds, then the speed stalls significantly. It seems like live-locking occurs around the 13 seconds mark, as the number of requests processed is stuck 2233.

The full output for this experiment can be found at `outputs/02-quorum.txt`

When using `broadcast` policy in `configuration.ex`, we encounter a similar situation. We process around 2000 requests in the first 6 seconds, then we stall and could only process a few more requests.

The full output for this experiment can be found at `outputs/03-broadcast.txt`

When using `quorum` or `broadcast`, the client sends a request to multiple replicas each time, instead of one at a time in `round robin`. When using 5 servers, `quorum` sends a request to 3 servers at a time, and `broadcast` sends a request to all 5 servers at a time. This duplication in client requests may be slowing the program down and causing a live-lock.

3.2.3 Crashing servers 3 and 5

The full output for this experiment can be found at `outputs/04-crash2.txt`. The experiment starts like the default setup, except that Server 3 is set to crash at time 1500ms, and Server 5 at 2500ms.

The system starts off executing the requests like normal. When server 3 crashes, the monitor shows that the database for server 3 no longer receives updates, while the other databases do, as expected. The same pattern is observed when server 5 crashes. The total client requests seen by replicas is just below what we would expect when no server has crashed. This is also expected as the crashed replicas will no longer receive the messages.

More interestingly, the total number of requests executed was below what we expected. We realised this is because of the client send policy. When a client sends requests in a round robin fashion, the crashed replicas do not receive the request and so are unable to pass it on to the system for the other replicas to execute. Changing the send policy to broadcast fixes this problem, but we run into performance / live-lock problems. Our replicas get slower at executing commands over the course of our program until they stop executing new commands at all. The problem might lie in our liveness solution, as we don't update the list of acceptors that we pass to new scouts and commanders.

3.2.4 Window Size

Since we did not implement reconfiguration commands, changing the window size does not do much. Window size is used in one place in replica, and as long as it is big enough, it doesn't affect performance. If it is really small, like 1, it slightly bottlenecks the rate of commands executed by the replicas.

3.2.5 Increasing the number of clients and servers

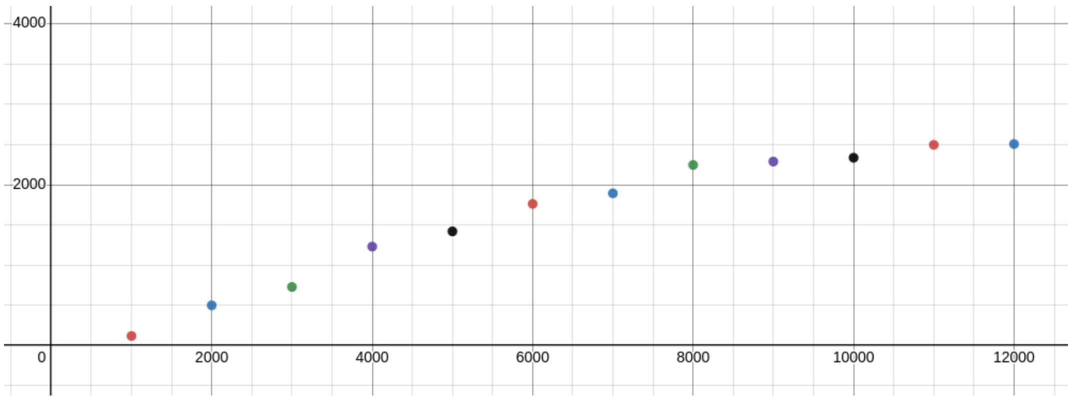


Figure 7: X-axis is time, while Y-axis is number of commands executed by a replica. This is a plot of file 05 in outputs/

We tested the program on 10 clients and 10 servers. Each client still sends 500 requests, resulting in a total number of 5000 requests. We processed 2900 requests after 22 seconds. After that, we experience live-locking and performance stalls. When timeout occurred at 60 seconds, we processed 3031 requests in total. It looked like the increase the in the number of requests once again caused live-locking.

The full output for this experiment can be found at `outputs/05-10_servers_10_clients.txt`.

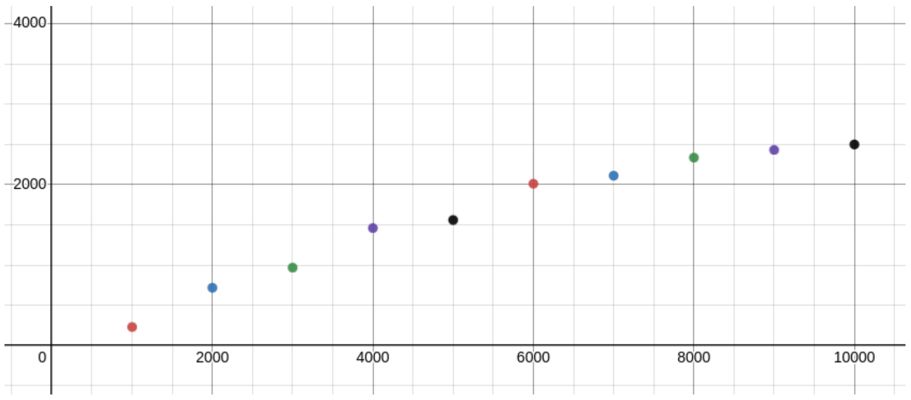


Figure 8: X-axis is time, while Y-axis is number of commands executed by a replica. This is a plot of file 06 in outputs/

We also tested the program on 10 servers and 5 clients. Interestingly, the performance was slower than the default configuration of 5 servers and 5 clients. It took 10 seconds to process all 2500 requests, roughly double the time of what the default configuration would take. We theorise the increase in the number of servers caused high contention for the acceptors, increasing the processing time.

The full output for this experiment can be found at `outputs/06-10_servers_5_clients.txt`.