Distributed Algorithms 60009 Coursework - Multi-Paxos

Szymon Kubica, CID: 01871147

February 27, 2023

Architecture

At the top level of the system the Multipaxos module spawns a collection of Servers and Clients and the Monitor which keeps track of the state of the whole system. Each Server has its own Database, Replica, Leader and Acceptor. The initial binding of the modules is done by Multipaxos, each replica gets bound to all leaders and each leader gets bound to all acceptors and replicas, each client gets access to all replicas. I omitted the corresponding `:BIND` messages.

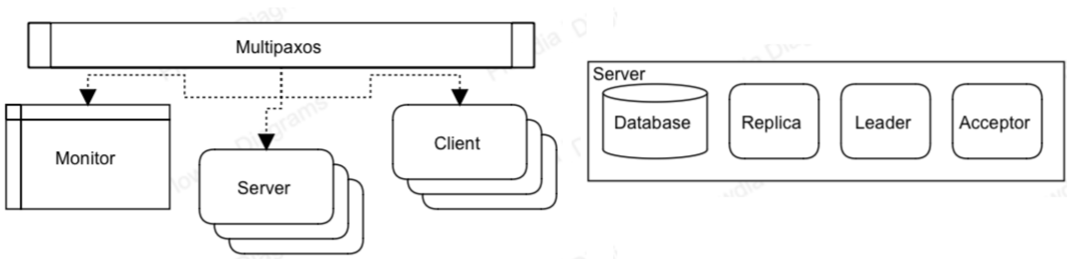

Figure 1: Top-level architecture.

A typical client request flow can be seen in Figure 2 below. The dotted arrows represent spawned child processes, the black solid ones represent directed messages with one recipient, whereas the blue ones represent messages which are broadcast to all modules of a given type. Diagram on the left depicts the phase 1 of the synod protocol, whereas phase 2 can be seen on the right. On the left one of the scouts got preempted and one got adopted. On the right a commander got accepted and broadcast its decision to all replicas which have sent execute requests to the databases.

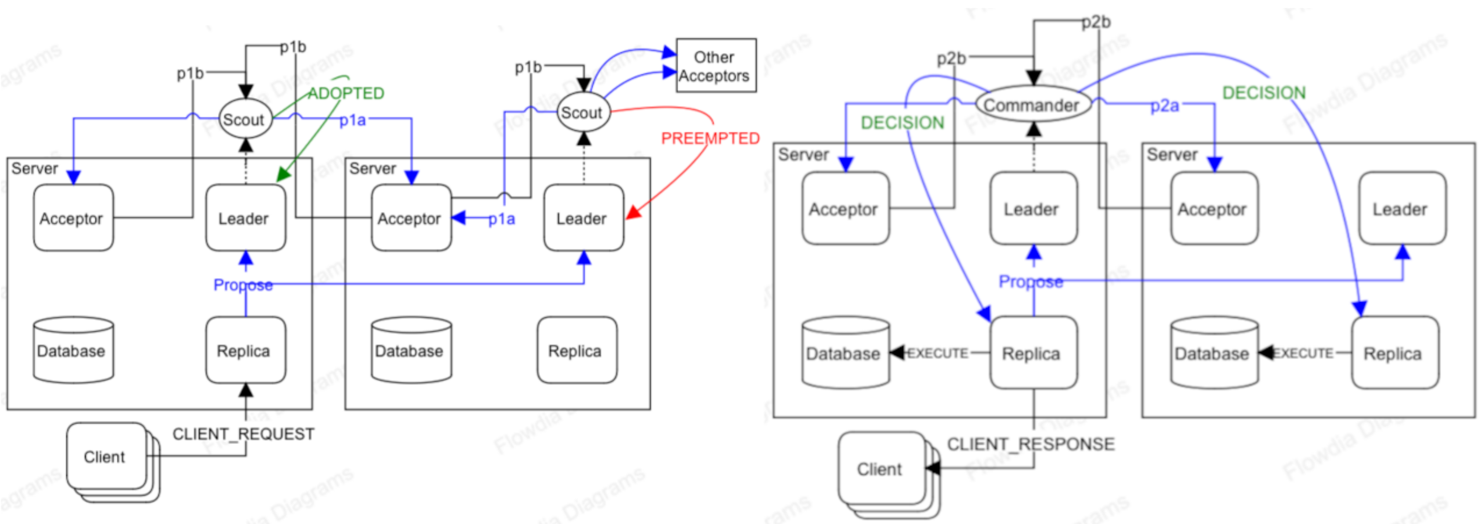

Figure 2: Typical client request flow.

Liveness

My initial implementation of the section 3 followed closely the algorithm described in the paper. Each leader after receiving a `:PREEMPTED` message becomes inactive and starts pinging the leader that has preempted it, instead of picking a higher ballot number and spawning a scout immediately. I implemented that by spawning one failure detector module for each leader. When preempted, a leader notifies its failure detector, which starts pinging the leader associated with the preempting ballot `b`, then the leader deactivates itself and continues .

```
{:PREEMPTED, b} ->
  send(self.failure_detector, {:PING, b})
  self |> deactivate |> next
```

Listing 1: New way of handling `:PREEMPTED` messages.

Each leader associates a timeout with its current ballot number. Every time a leader gets preempted, its next ballot will use a longer timeout (multiplicative increase). However, the timeout of the current ballot decreases linearly with each proposal chosen for that ballot. It is done by sending an appropriate notification from the commander after a `:DECISION` message is sent to the replicas. Whenever an active leader gets pinged by FD of some other leader, it sends its timeout back, and the FD updates it and uses it from now on.

```
{:RESPONSE_REQUESTED, requestor} ->
    cond do
      self.active -> send(requestor, {:STILL_ALIVE, ballot_num, timeout})
      self.preempted_by != nil -> send(requestor, {:STILL_ALIVE, preempted_by, timeout})
```

Listing 2: Leader responding to a ping message.

When tweaking the timeout settings, it was very difficult to pick the appropriate constants. If the timeout decreased too quickly or it didn't grow enough after preemptions, all leaders ended up preempting each other which impacted performance. I noticed that we don't prioritise fairness, it is perfectly fine if one leader is working on its ballot and all of the other ones keep pinging it regularly and only react if they detect a failure. I added a new version of the failure detector which has a static timeout. I realised that it could happen that three leaders λ_1, λ_2, λ_3 behave as follows: λ_2 preempts λ_1 and λ_3 preempts λ_2, in which case λ_2 becomes inactive and stops responding to λ_1's pings, thus λ_1 wakes up and can possibly preempt λ_3. Even though variable timeouts would help to solve this issue and prevent it from live-locking, it introduces an unnecessary congestion. I solved this problem by having each leader record the last preempting ballot number. If a leader was recently preempted and is now inactive and gets pinged by somebody whom he preempted before, it sends the ballot number that preempted it back to the requestor (Listing2 above). That way the leader who pinged us doesn't wake up immediately but starts pinging the leader who is currently working.

Evaluation

Hardware

The evaluation was conducted on my laptop with the following specification:

OS: Linux 6.1.12-arch1-1, processor: 11th Gen Intel i7-1165G7 4.700GHz, cores: 8, ram: 32GB.

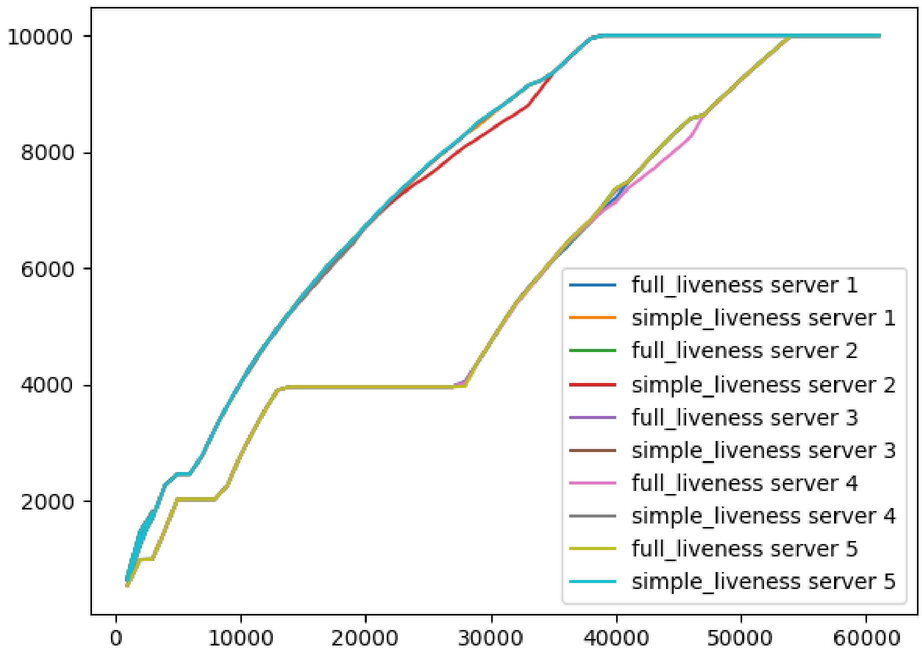

Figure 3: Top-level architecture.

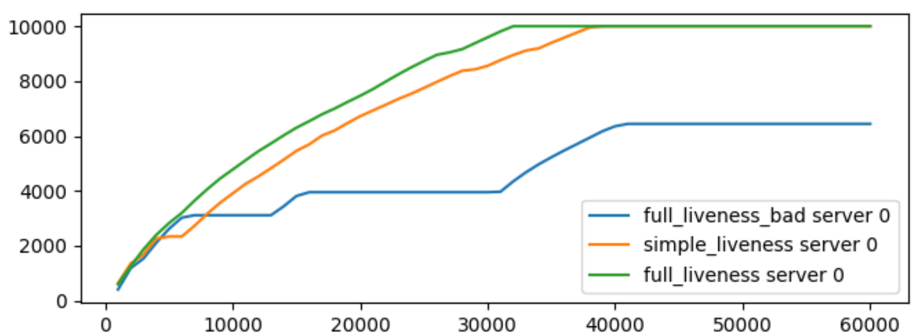

Figure 4: Top-level architecture.

Live-locking Implementation and a Partial Fix

Full Liveness and Simple Liveness

Crash Failure of the Main Leader