Multi-Paxos Elixir Implementation Report

6009 Distributed Algorithms Coursework
Nicolas D'Cotta (nd3018) and William Profit (wtp18)

Design and Implementation

We closely followed the design outlined in the paper "Paxos Made Moderately Complex" by Robbert Van Renesse and Deniz Altinbuken with a few modifications.

Most notably we make use of a collapsed architecture where a server process is colocated with one replica, one acceptor, one leader and one database. This allows for a simpler design while still ensuring correctness of the algorithm. The server and client processes initially get spawned by the over arching `multipaxos` process which deals with bootstrapping the system.

We used type annotations wherever we could in order to enhance readability and have `mix` provide us with additional information to catch bugs earlier on.

Ballots

In order to represent ballots that can be ordered but still hold their leader's PID, we initially set out to order ballots by hashing the string representation of their PIDs. We later found that two different PIDs may have the same string representation, effetively making leaders *not* totally ordered. Instead, we adopt an approach where a ballot is a tuple of three (rather than two) elements: $< b_{number}, \texttt{server\_num}, process_{id} >$. We then only use the ballot number b_{number} and the node id $\texttt{server\_num}$ to lexicographically order ballots.

Data Structures

To store proposals, we use `Map` s for efficient $O(1)$ membership check operations. For other states, we use `MapSet` s.

For more complex modules like `replica` , we encapsulate all the member variables into a `state` dictionary that gets updated and passed around.

Liveness

To implement liveness, we adpoted an approach where a Leader has a t_{wait} time period as part of their state. When they get preempted, they wait t_{wait} before trying to increase their ballot number.

Choosing t_{wait}

We implemented TCP-like Additive Increase, Multiplicative Decrease. When a leader succesfully has one of its ballots adopted, it increases t_{wait} slightly by Δ. When it gets preempted, it multiplies it by a factor γ close to but smaller than 1.

For a more aggressive setup (ie, more susceptible to live locking, but more efficient) we can decrease Δ and increase γ, and viceversa for a more conservative setup that may be less performant but less likely to livelock.

t_{wait} allows a leader λ that succesfully gets ballots out often to 'leave room' for another leader λ' to preempt it. Once λ' has several succesful adoptions, it will start backing off and leaving room for λ again.

Detecting Leader Failure

While a leader λ waits for λ' (due to the backoff described above). It doesn't just sit idle - it pings λ' to make sure it didn't fail. We implement this failure detection to quickly preempt faulty leaders even if the t_{wait} of λ is large.

Debugging and Testing Methodology

We extended `Monitor` by adding an *active leaders* field. Particularly useful in testing liveliness, and determining when a slow system is 'stuck'.

We use a `Util.log()` helper function extensively, which takes a 'debug level' atom which serves as logging instrumentation for the application. In order to easily debug to what processes the messages get passed around, we always print the `node_num` and the type of process in question (Scout, Acceptor, etc) before every message.

We also wrote a testing suite that unit tests for certain modules of our implementation. This has allowed us to be more confident in making changes as we ensured we were not breaking things elsewhere. Unit testing every module was however not possible due to the distributed atchitecture of the application

Correctness of the System, Outputs and Findings

We have run our program under several scenarios, of which the outputs can be found under the `outputs` directory.

> We call 'livelock prevention' the implementation described above, where a leader backs off if no other leader is preempting it.

We ran several experiments:

- With 2 servers, 5 clients, on low load
  - With livelock prevention
  - Without livelock prevention
- With 5 servers, 5 clients
  - With livelock prevention
    - Low load
    - High Load
    - High load, where a server artifically crashes mid execution
    - Very high load
  - Without livelock prevention

- Low load
- High Load
- Very high load

First off we notice that using 2 servers generally gives better performance than using 5 servers. This is because the majority required to achieve consensus is smaller, so there is less overhead.

Without livelock prevention we can almost manage low loads but then very little work gets done under high and very high load. This is due to the fact that under higher workloads, leaders are competing more to get their ballots adopted so livelock occurs with a higher probability, and thus more often.

On the other hand, adding livelock prevention significantly improves performance and reliability. We are able to manage high loads and even very high loads although for the latter we encounter performance issues, where we are making progress but not fast enough given the rate of incoming requests. This is seen as the update rate lagging behind the request rate. However the system is still correct and would eventually catch up if given the time to or if ran with better performance.

We are sure our system has *liveness* (ie, *'if a client broadcasts a new command to all replicas, that it eventually receives at least one response'*) once livelock prevention is introduced. This is because of how t_{wait} is chosen when a leader gets preempted (described in the Implementation section): when a leader λ fails to get its ballot adopted by getting preempted (by say, λ'), two scenarios are possible:

- λ has time to get its ballot adopted without getting preempted by λ'
  - This appens because the t'_{wait} of λ' is large enough, or by chance
  - This scenario does not lead to a livelock, as described in the paper
  - When λ' does preempt λ, its t'_{wait} will be smaller than t_{wait}, so it will have time to get its ballot approved. If it does not manage and gets preempted back, it will 'try harder' with an even smaller t'_{wait}
- λ preempts λ' back (might lead to a livelock)
  - t'_{wait} of λ' was increased in the previous round (because, as the preemptor, it received `:adopted`), while t_{wait} of λ decreased
  - t_{wait} it liekly decreased more than t'_{wait} increased, due to TCP-like AIMD
  - Thus, for every likevelock-like round, it becomes increasingly likely for one of the competing leaders to decrease their t_{wait} so much that they will be quick enough to preempt the other replica, which is waiting for longer for trying again
  - This guarantees that *eventually*, one of the leaders will win out. When they do, their t_{wait} will become large again as it gets more succesful adoptions, preparing for the next 'livelock like' scenario that we just described.

When crashing a server, we notice that the behaviour of the system is not affected, other servers continue serving clients as expected.

We can conclude that although our system us correct, it is still slow enough that it does not manage to handle the high load of the `:default` configuration (provided with the code skeleton) of 5 clients, 5 servers, 5000 requests/client, under 60s. Unfortunately, with our current implementation the system gets slower as we progress through more client requests. Given we do still make progress, we have reson to think that if we did implement an efficient variation of paxos (e.g. change acceptor to use integers not sets, delete old slots) it would perform very well.

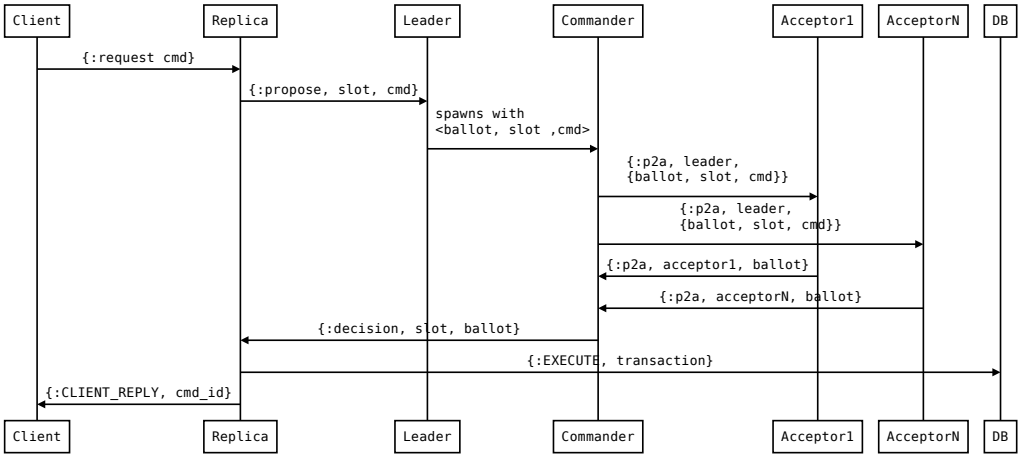
Environment

The program was run uder MacOS on a 2019 16" MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 CPU and 16 GB 2667 MHz DDR4 RAM.

Diagrams and Requests Flow

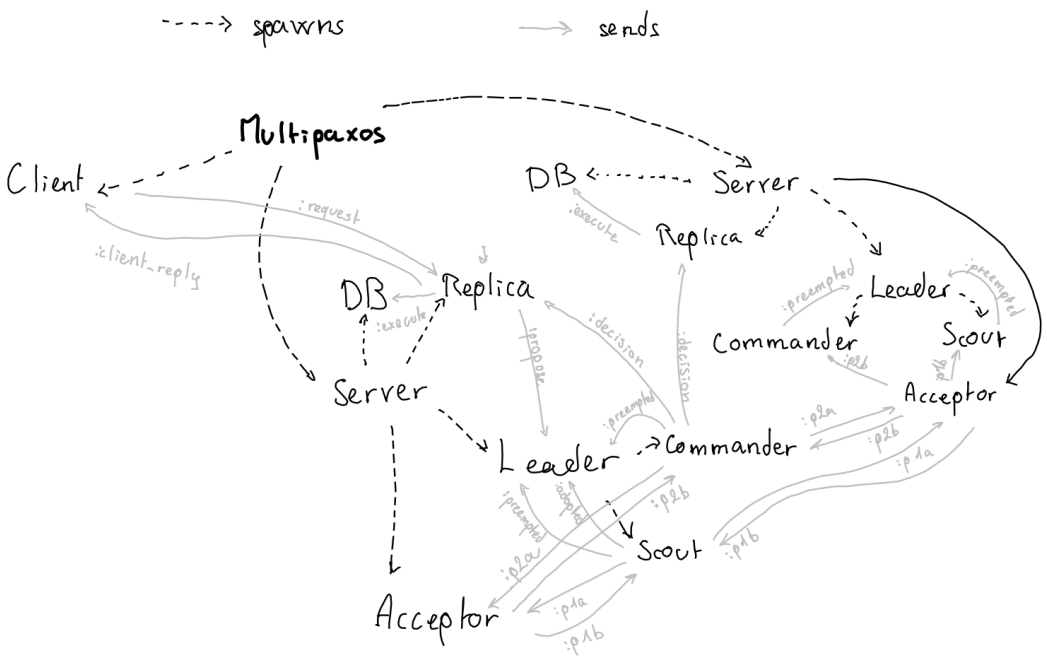
Lifecycle of a Client's `command`

We show the UML diagram of the lifecycle of a Client's `command`, when one of the leaders is active. Note how we can skip phase one (`:p1*` messages) altoghether.



Overview of Flow

We present a flow diagram, where we show the commands that a new command form a client may (or may not) trigger. We do not show every single command that exists in our implementation. Rather, we show for example how a second replica might get `:preempted`, with that command originating from a `:p2b` that an Acceptor may with to a Commander. The content of that `:p2b` originates from the ballots that same Acceptor may have been receiving from a different Commander.



In particular, for readability we do not show:

- `:ping` and `:pong` messages between Leaders for failure detection
- Debugging messages for the Monitor (nor the Monitor itself)