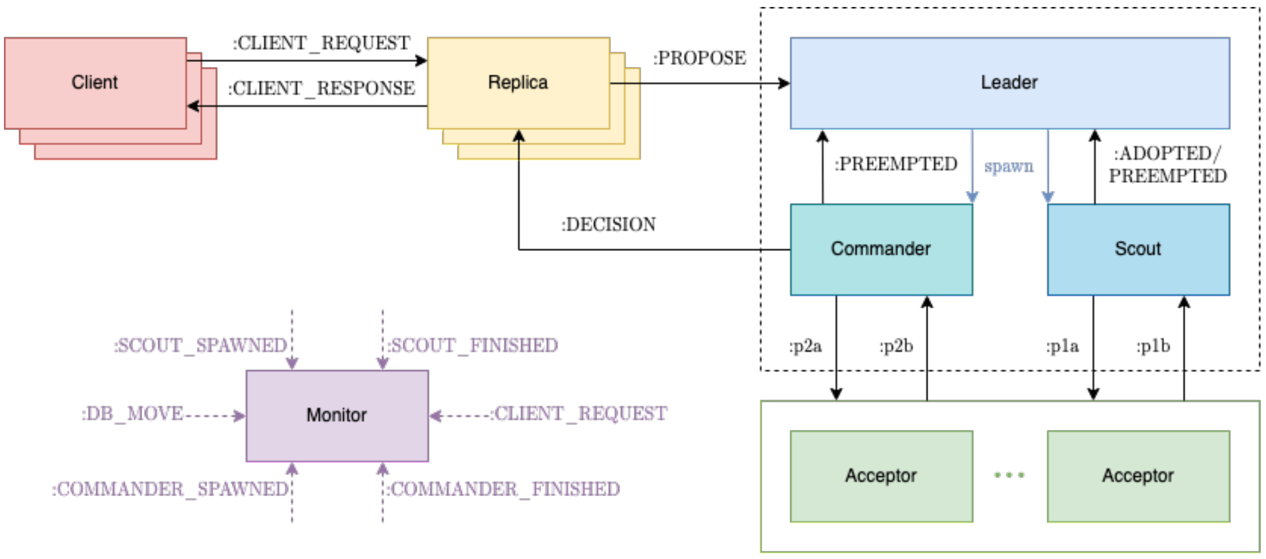
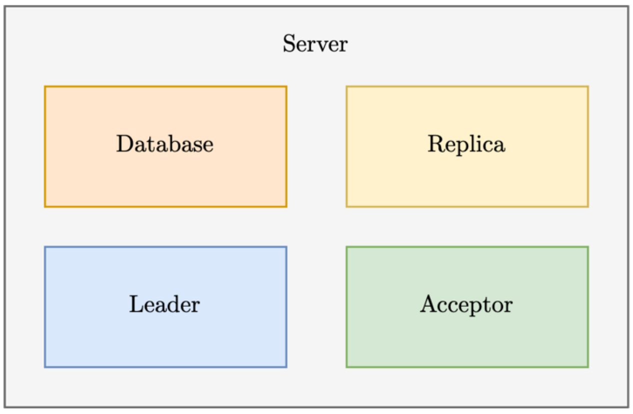
Multi-Paxos Report
-
Adam Alilou (aa1320)

Architecture

The elixir implementation of the Multi-Paxos algorithm uses 6 key modules; Clients, Replicas, Leaders, Scouts, Commanders and Acceptors.
There is also a Monitor module which is used to keep track of the progress during the run of the algorithm, it receives messages to measure spawning and finishing of commanders and clients, client requests and database moves which are sent after a replica sends an :EXECUTE message to the database.



Above is a diagram about how the modules interact with respect to messages sent and received.



Above is a diagram showing the actual structure of how a server is constructed, with my experiments 5 servers were run

The Client module sends the initial request to all the replicas.
The Replica module receives requests and eventually sends a client response to the clients and command to the database.
The Leader module receive proposed commands and decides on a slot where the command will go.

The Commander module works on a single slot that may be one of many the leader is working on.
The Scout module aims to receive p1b messages from a majority of acceptors.
The Acceptors keeps track of accepted 'pvalues' {Ballot Number, Slot, Command} and the current ballot number.

Liveness

To implement section 3 of the paper I added a timeout element to the self map in the Leader module. This meant that to know which leader to ping, I had add the leader process identifier to the ballot. {value, leader} → {value, leader, leader_PID}.
I added the very first liveness approach which is shown here.

```
defp monitor_leader(self, other_leader) do
    time = :os.system_time(:millisecond)
    send(other_leader, {:PING, self()})

    time =
      receive do
        {:PING_RESPONSE, ^other_leader} ->
          :os.system_time(:millisecond) - time
      end

    if time < self.config.max_response_time do
      monitor_leader(self, other_leader)
    end
  end
end
```

This function is right after receiving a :PREEMPTED message from either a scout or commander and will return after the leader no longer responds in time. The `max_response_time` was chosen to be 50ms, typical response times were between 0-2ms but sometimes ~20ms. The results of this are included in the evaluation.

To implement the other more involved liveness scheme, I then included a timeout field associated to each ballot.
{value, leader, leader_PID} → {value, leader, leader_PID, timeout}.

With this algorithm, each ballot has an initial timeout `min_timeout`. Then in the leader module;
If a ballot has been pre-empted:
    The next ballot uses a new timeout, = old timeout × `timeout_factor`
If a proposal has been chosen:
    Decrease the initial time out linearly, = old timeout − `decrease_amount`

Below are some code snippets which show the changes to the program to implement this liveness algorithm.

```
max_response_time: 50,
min_timeout: 10,
max_timeout: 150,
decrease_amount: 35,
timeout_factor: 1.5
```

The new fields inside config

```
{:PROPOSAL_CHOSEN} ->
  self = %{
    self| timeout: max(self.timeout - self.config.decrease_amount,
                       self.config.min_timeout)
  }
```

Inside leader, waiting for a :PROPOSAL_CHOSEN message to decrease the timeout.

```
{:PREEMPTED, {value, _preempting_leader, leader_PID, _timeout} = b1} ->
  # A scout or commander has told us that some acceptor has adopted a higher ballot number

  self = %{
    self | timeout: min(round(self.timeout * self.config.timeout_factor),
                        self.config.max_timeout)
  }
```

Now after receiving a :PREEMPTED message, the timeout value is increased before a scout is spawned and the ballot value is incremented.

Evaluation

I've been testing on my 2017 MacBook Pro on macOS Monterey, with a 2.9 GHz 4 core Intel i7 CPU and 16 GB of 2133 MHz RAM.

Experiment Results:

No Liveness:

Running the algorithm 20 times with 500 client requests each and a 15 second timeout:

- 10/20 times it livelocked and of the times it livelocked, it ended with mean of 2497.5 total requests completed.
- 10/20 times it completed all 2500 of the requests/
- All 20 terminal outputs from the runs can be seen in `/Findings/500clients/`

Running the algorithm 10 times with 2000 client requests each and a 60 second timeout:

- 10/10 Completed all 10000 requests.
- The mean time to reach 10000 requests was 41.5s.
- All 10 terminal outputs from the runs can be seen in `/Findings/2000clients/`

Basic Liveness – Simple ping

Running the algorithm 20 times with 500 client requests each and 15 second timeout:

- 7/20 times it livelocked and of the times it livelocked, it ended with a mean of 2497.5 total requests completed.
- 13/20 times it completed all 2500 of the requests.
- All 20 terminal outputs from the runs can be seen in `/Findings/500clientsliveness1/`

Running the algorithm 10 times with 2000 client requests each and a 60 second timeout:

- 2/10 times it live locked and of the times it livelocked, it ended with a mean of 9253.5 total requests completed.
- 8/10 times it completed all 10000 of the requests, the mean time to complete all the requests was 45.38s.
- All 10 terminal outputs from the runs can be seen in `/Findings/2000clientsliveness1/`

Interestingly this livelocked more than the normal Multi-Paxos algorithm which didn't livelock at all.

Liveness 2 – Variable timeouts

Running the algorithm 20 times with 500 client requests each and a 15 second timeout:

- 16/20 times it livelocked and of the times it livelocked, it ended with a mean of 2495.44 total requests.
- 4/20 times it completed all 2500 of the requests.
- All 20 terminal outputs from the runs can be seen in `/Findings/500clientsliveness2/`

Running the algorithm 10 times with 2000 client requests each and a 90 second timeout:

- 10/10 Completed all 10000 requests, of these requests the mean time to complete them all was 47.3.
- All 10 terminal outputs from the runs can be seen in `/Findings/2000clientsliveness2/`

Interestingly when the program is ran with a lower amount of client requests (500 each) it tends to livelock more than the equivalent algorithm with more client requests (2000 each). The extreme example was when running the liveness2 algorithm on 500 client requests where it livelocked 80% of the time.

In conclusion the second liveness algorithm seems to work much better with a larger of client requests.