Seminar Report: Paxy

Miguel Cruz; Javier Cano; Riccardo Cecco

December 26, 2021

1 Introduction

This seminar is about the Paxos protocol. The Paxos algorithm's objective is for a group of peers to come to an agreement on a value. If one peer believes a majority has agreed on a value, Paxos ensures that the majority will never agree on a different value. Any agreement must go through a majority of nodes, according to the protocol. Any subsequent efforts at agreement must pass via at least one of those nodes if they are to be successful. As a result, every node that makes a proposal after a decision has been made must interact with a majority node. The protocol ensures that the majority will learn the previously agreed-upon value. This is obviously related to distributed systems in the way they manage to reach consensus to make decisions.

2 Experiments

Here we describe the experiments we carried out during this seminar. For all of them, we define 1 second as the starting sleep time of the proposers.

2.1 Delays on acceptor messages

For the first experiment, we introduced different delays in the promise and vote messages sent by the acceptor. The delay chosen for each message is decided by a random number of milliseconds between 0 and the defined maximum delay. We took several samples per maximum delay and have compiled them in the following plots:

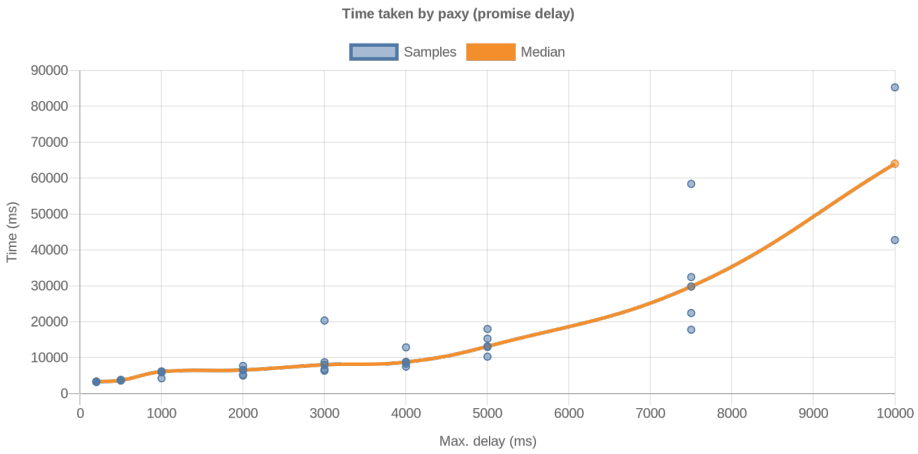

Figure 1: Time spent by paxy when promises are delayed.

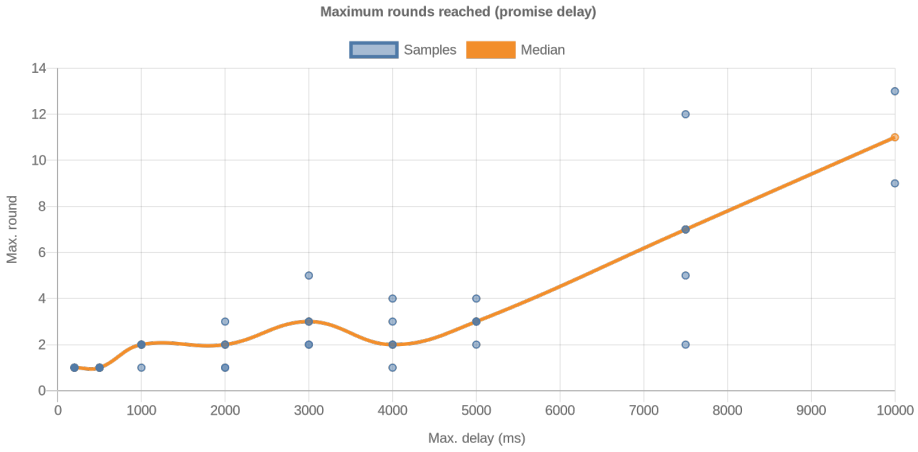

Figure 2: Rounds passed when promises are delayed.

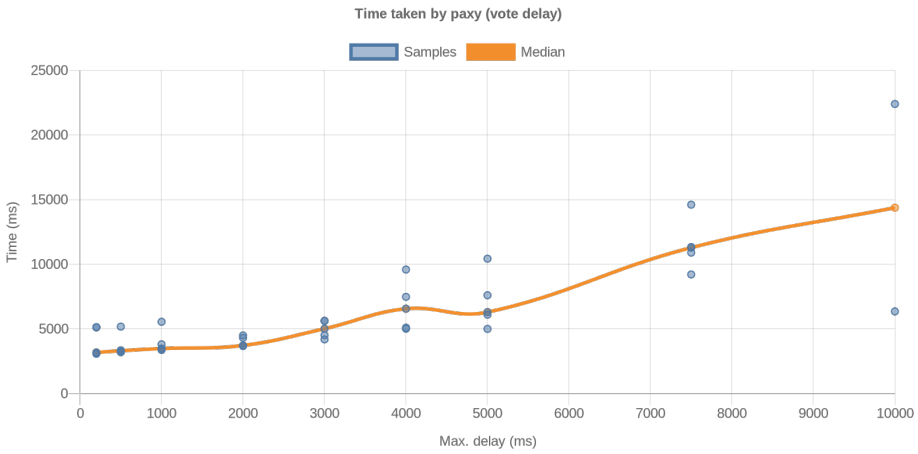

Figure 3: Time spent by paxy when votes are delayed.

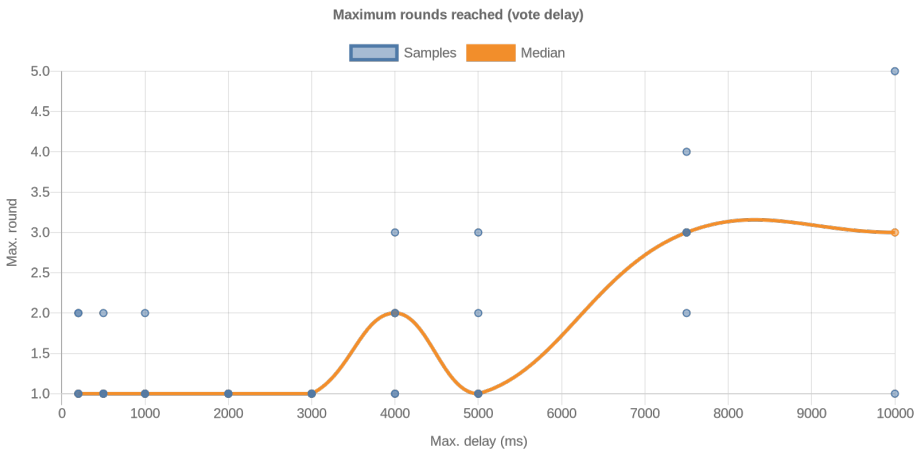

Figure 4: Rounds passed when votes are delayed.

By comparing figures 1 and 2 with figures 3 and 4, we can clearly see that the delays on promise affect the paxos algorithm much more negatively than the delays on vote. The time taken to reach consensus and number of rounds is always higher. We can observe how as the rounds increase, the time to reach consensus does too almost proportionally. However, the average number of rounds starts increasing earlier and reaches higher numbers in the case of delaying promises, which may explain the difference between both delays. Nevertheless, the time per round in the case of delaying promises is still higher.

2.2 Sorry messages avoidance

For the second experiment, we didn't send the sorry messages by commenting the corresponding lines of our code and we didn't notice much difference. The algorithm still works. We suspect that not sending sorry messages will in some cases make the process faster, as every time the proposer receives a sorry message, its timeout value resets.

2.3 Dropping acceptor messages

For the third experiment, we randomly dropped messages sent by the acceptor. We took several samples for vote and promise messages. These are the results:

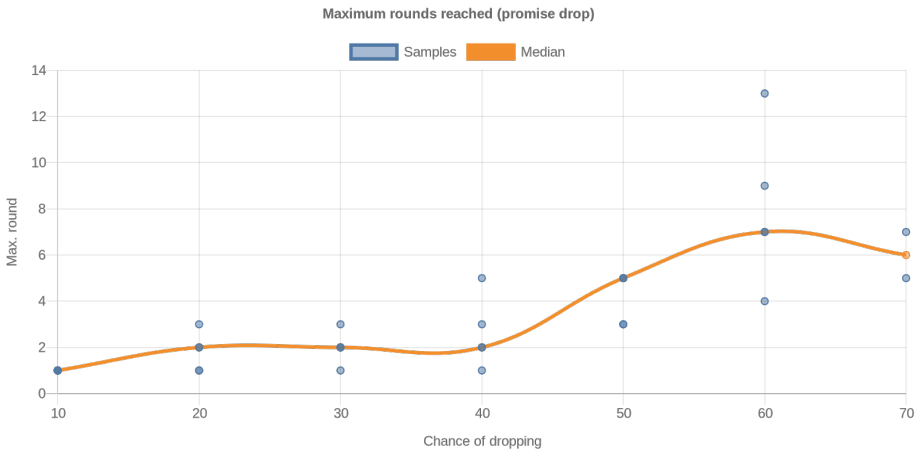

Figure 5: Rounds when promises are dropped.

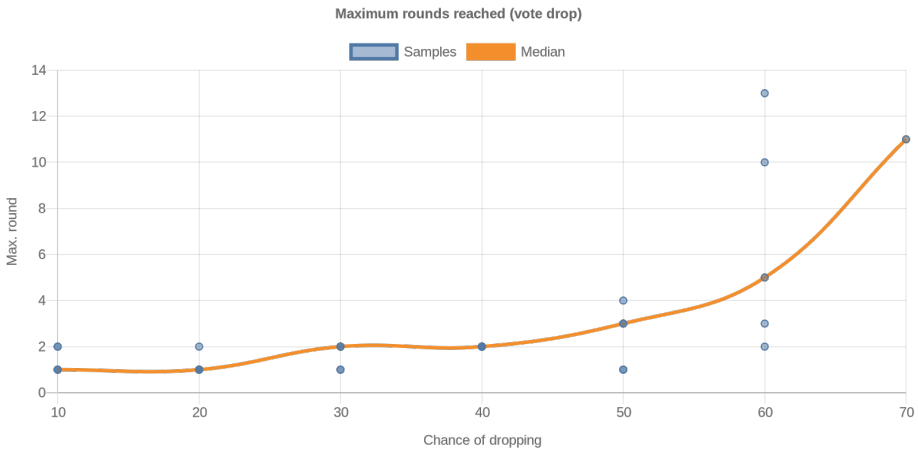

Figure 6: Rounds when votes are dropped.

We can see a similar trend to the delay experiment. Votes dropped seem to impact the rounds number a bit less negatively. Also, after around 6 in 10 messages dropped, the number of rounds seemed to get pretty bad, passing 14 or 15 in some cases. At that point we considered a good idea to stop the execution, as it was taking a considerable amount of time per round. In the end, we determined that going further than 7 in 10 messages dropped didn't make much sense as with this number, the algorithm was failing most times (even if we report some successful attempts in the plots).

2.4 Increasing the number of acceptors or proposers

For the fourth experiment, we increased the number of acceptors and proposers to see if there were any changes in the process. As always, we took several samples and made some plots, and took some proof with consensus reached on up to 15 acceptors and 10 proposers.

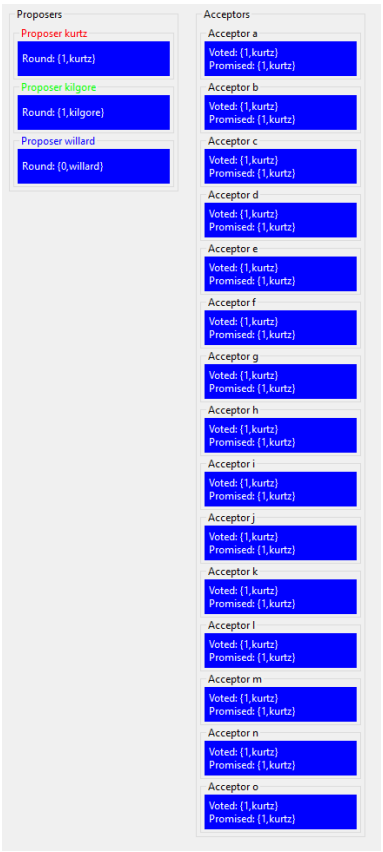

Figure 7: Consensus reached with 15 acceptors.

Figure 8: Consensus reached with 10 proposers

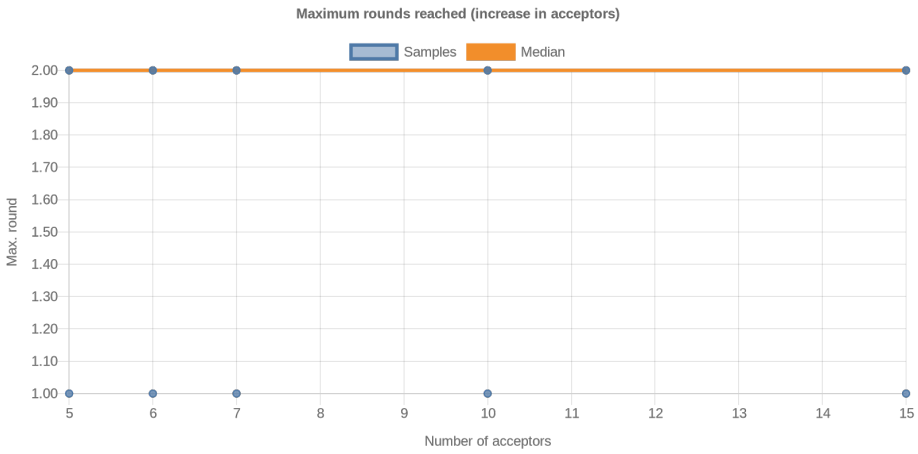

Figure 9: Rounds with increasing number of acceptors.

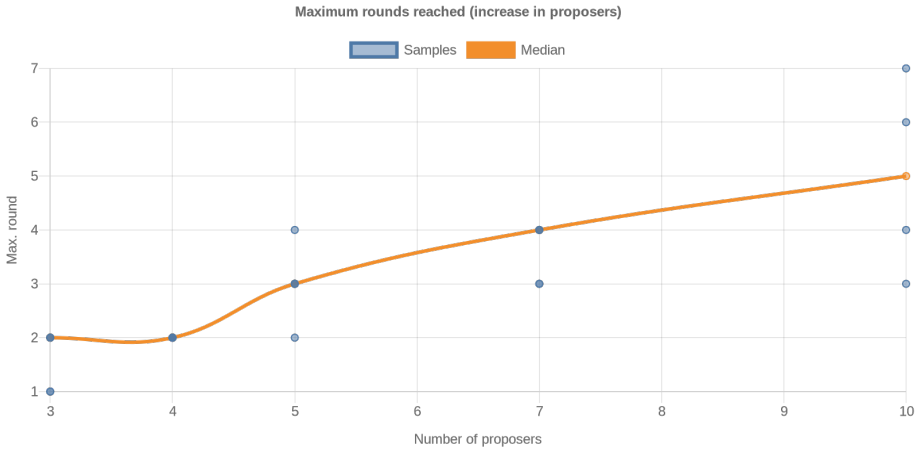

Figure 10: Rounds with increasing number of proposers.

The number of acceptors didn't make any difference, but the rounds seemed to increase slowly with the number of proposers.

2.5 Remote acceptors and proposers

For the fifth experiment, we edited the paxy module to create acceptors and proposers in two separate remote nodes. For this, we needed 3 instances of erlang started by the following commands:

- **erl -name paxy@127.0.0.1** (instance where paxy runs)
- **erl -name paxy-acc@127.0.0.1** (instance for the acceptors)

- **erl -name paxy-pro@127.0.0.1** (instance for the proposers)

Then, we needed some way to spawn the proposer and acceptor processes in another node. After some research, we decided to use the erlang remote procedure calls module (rpc). This module provides us with the following function:

- **call(Node, Module, Function, Args)**

This function can call Function of Module in Node with parameters Args. This allows us to spawn the acceptors and proposers in another node just by calling the start_proposers and start_acceptors functions wrapped in this call.

To call these functions so they can work correctly with names registered locally in other nodes, we had to make some changes in the code.

First, we added the corresponding exports and node names:

```
1 -export([start/1, stop/0, stop/1, start_acceptors/2,
      start_proposers/5]).
2
3 -define(acceptors_node, 'paxy-acc@127.0.0.1').
4 -define(proposers_node, 'paxy-pro@127.0.0.1').
```

Then, we created a new list, with tuples containing the names of the acceptors and the node where they would run on. This list should then be passed to the start_proposers function instead of the usual one, so the proposers know where to send messages:

```
1 AccRegisterRemote = [{a,?acceptors_node}, {b,?acceptors_node}, {c,?
      acceptors_node}, {d,?acceptors_node}, {e,?acceptors_node}],
```

Finally, we changed the way to call both functions:

```
1 rpc:call(?acceptors_node, paxy, start_acceptors, [AccIds,
      AccRegister]),
2 rpc:call(?proposers_node, paxy, start_proposers, [PropIds, PropInfo
      , AccRegisterRemote, Sleep, self()]),
```

With these changes, everything works as usual.

2.6 Fault tolerance

For the sixth experiment, We tested the fault tolerance of the process, by adding ways to store the state of the acceptors after every change. We used the provided pers module that contains functions to write and read states from files.

First, we added a helper function on the acceptor:

```
store(Name, Promised, Voted, Value, PanelId) ->
  pers:open(Name),
  pers:store(Name, Promised , Voted, Value,PanelId),
  pers:close(Name).
```

This function allows us to record the state of the acceptor, by calling it just before a recursive call to acceptor, with the same parameters.

We used the provided crash function to take down an acceptor:

```
crash(Name) ->
    case whereis(Name) of
        undefined ->
            ok;
        Pid ->
            pers:open(Name),
            {_, _, _, Pn} = pers:read(Name),
            Pn ! {updateAcc, "Voted: CRASHED", "Promised: CRASHED",
     {0,0,0}},
            pers:close(Name),
            unregister(Name),
            exit(Pid, "crash"),
            timer:sleep(3000),
            register(Name, acceptor:start(Name, na))
    end.
```

We needed also a way to know if the acceptor was being started by the crash function, or at the start of the regular process. We achieved this by editing the init function on the acceptor:

```
init(Name, PanelId) ->
  io:format("CREATING!!!!! Name:~w~n",
                    [Name]),
  pers:open(Name),
  Promised = order:null(),
  Voted = order:null(),
  Value = na,
  case PanelId == na of
    false ->pers:store(Name, Promised, Voted, Value, PanelId),
          pers:close(Name),
          acceptor(Name, Promised, Voted, Value, PanelId);
    true -> {Pr, Vot, Val, Pn}=pers:read(Name),
      pers:close(Name),
      case Pn == na of
        true->
          ok;
        false ->
          io:format("RECOVERING!!!!! PID:~w~n",
                    [Pn]),
          case Val == na of
            true ->Pn ! {updateAcc, "Voted: " ++ io_lib:format("~p"
    , [Vot]),
                        "Promised: " ++ io_lib:format("~p", [Pr
    ]), {0,0,0}};
```

```
23              false-> Pn ! {updateAcc, "Voted: " ++ io_lib:format("~p
    ", [Vot]),
24                           "Promised: " ++ io_lib:format("~p", [Pr
    ]), Val}
25                           end,
26            acceptor(Name, Pr, Vot, Val, Pn)
27          end
28    end.
```

When its being called at the start of the process, it records its initial state. If it's being called by the crash procedure during recovery, it tries to read its state from disk and makes some checks for null values.

Finally, we added some logic to paxy's stop function, to make it delete the file states after a successful run:

```
1  stop(Name) ->
2    case whereis(Name) of
3      undefined ->
4        ok;
5      Pid ->
6        case Name == gui of
7          false->
8            io:format("DELETED!!!!! Name:~w~n",
9                      [Name]),
10           pers:delete(Name)
11         end,
12
13       Pid ! stop
14   end.
```

With these changes and the provided changes to the proposer's send function logic, we reported a correct recovery of the acceptors after crashing. However, we see some erratic behaviour sometimes if we let paxy delete every state after a successful run, and an acceptor tries to recover after this (because its saved state has been deleted).

2.7 Improvement based on sorry messages

In this experiment, we adjust the Paxos algorithm to abort when sorry messages reach the majority of quorum. So, for that, we had to change the call/4 and vote/2 procedures.
Thus, we added a variable X that decreases when a sorry message is sent (in both procedures).

```
1  collect(0, _, _, _, Proposal) ->
2    {accepted, Proposal};
3  collect( _, 0, _, _, _) ->
4    abort;
```

```
collect(N, X, Round, MaxVoted, Proposal) ->
  receive
    {promise, Round, _, na} ->
      collect(N-1, X, Round, MaxVoted, Proposal);
    {promise, Round, Voted, Value} ->
      case order:gr(Voted, MaxVoted) of
        true ->
          collect(N-1, X, Round, Voted, Value);
        false ->
          collect(N-1, X, Round, MaxVoted, Proposal)
      end;
    {promise, _, _,  _} ->
      collect(N, X, Round, MaxVoted, Proposal);
    {sorry, {prepare, Round}} ->
      collect(N, X-1, Round, MaxVoted, Proposal);
    {sorry, _} ->
      collect(N, X-1, Round, MaxVoted, Proposal)
  after ?timeout ->
    abort
  end.
```

```
vote(0, _, _) ->
  ok;
vote(_, 0, _) ->
  abort;
vote(N, X, Round) ->
  receive
    {vote, Round} ->
      vote(N-1, X, Round);
    {vote, _} ->
      vote(N, X, Round);
    {sorry, {accept, Round}} ->
      vote(N, X-1, Round);
    {sorry, _} ->
      vote(N, X-1, Round)
  after ?timeout ->
    abort
  end.
```

The X variable is initialised equal to the N variable.

Due to these changes, the expected results would be a decrease in algorithm execution time. And this is what the figures below show us.

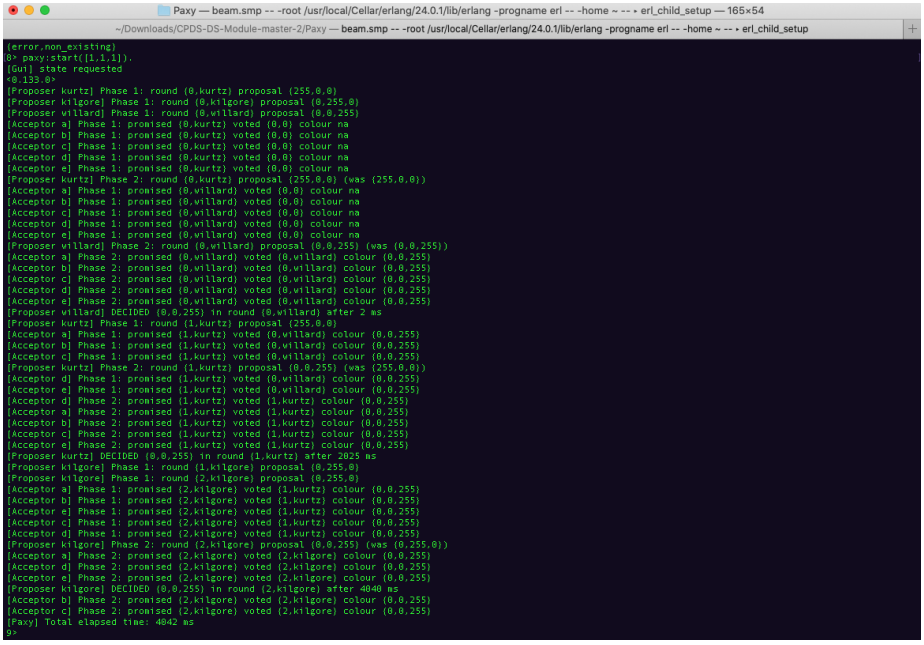

Figure 11: Paxos algorithm without sorry messages improvement.

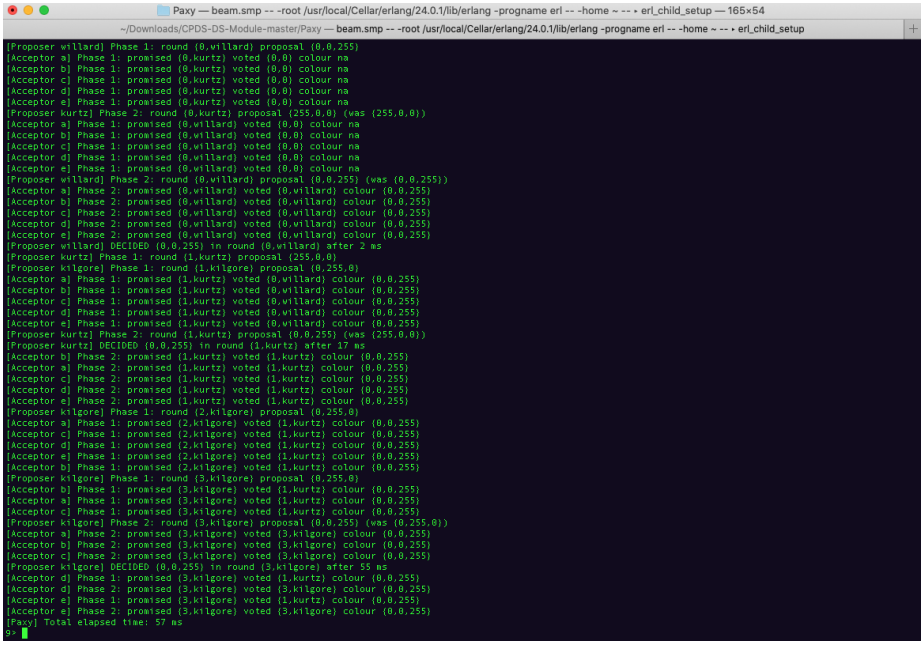

Figure 12: Paxos algorithm with sorry messages improvement.

3 Open questions

Q1) Try introducing different delays in the promise and/or vote messages sent by the acceptor. Does the algorithm still terminate? Does it require more rounds? How does the impact of message delays depend on the value of the timeout at the proposer?

According to the first experiment described in the previous section, the algorithm terminates correctly in most cases. The behaviour defined by the delay code, introduces a delay between 0ms and delay ms. Quite obviously, this makes the proposer abort its procedure and try again if at some point a message doesn't arrive for "timeout" ms (timeout being defined as 2000ms at the proposer by default). So considering this, the larger the range of numbers to be picked between 0 and the defined maximum delay, the more chances two different instances will be separated by more than 2000ms (the defined timeout). We start seeing this behaviour happen more commonly at some point between 5000 and 7500 as max delay, where the rounds started to increase.

Overall, after the 10000 mark on max delay, the algorithm was taking too long so we consider that it doesn't terminate in a reasonable time.

Q2) Could you come to an agreement when sorry messages are not sent?

As we already wrote on the second experiment above, we didn't find any problem on reaching consensus after not sending any sorry messages. This is because the proposer doesn't do anything after receiving them.

Q3) What percentage of messages can we drop until consensus is not longer possible?

According to the data generated by the third experiment, we found that with 7 in 10 messages dropped, we could sometimes not find consensus in a reasonable time. Knowing this, we consider that dropping more than 60% of the messages may not be a good idea. Maybe some warning should arise if in a real implementation we detect some node going over this threshold.

Q4) What is the impact of having more acceptors while keeping the same number of proposers? What if we have more proposers while keeping the same number of acceptors?

As stated in the fourth experiment, the number of acceptors doesn't seem to matter. This is because no matter the number of acceptors, every proposer sends their prepare messages at roughly the same time, so every acceptor decides to promise the same thing in the first round. In the data provided by figure 9, we can see how up to 15 acceptors don't change the number of rounds.

On the other hand, increasing the number of proposers seems to have some impact, as they struggle to receive their required number of vote messages before some other proposer "steals" their turn and they have to make another round. According to figure 10, the number of rounds increases somewhat slowly as

the number of proposers increases. With 10 proposers we saw the algorithm sometimes reach a number of rounds as high as 7.

4 Personal opinion

We found it very useful to have to implement the Paxos algorithm by ourselves, because by developing the project we were able to better understand the process and its functioning.

We had the opportunity to learn a new programming language called Erlang which seems to be useful in the field of distributed systems.

Furthermore, in our case, we had the opportunity to work in an international team, which some of us hadn't done until this moment. It was a pleasure and to work better together we created a common repository on GitHub where each of us could develop the code and try new experiments. This will all be included in separate directories in the final product.

It was a good experience and in our opinion the seminar should be proposed again next year.