# Seminar Report: Paxy seminar

**Oriol Martínez Acón & Imanol Rojas Pérez**

December 1, 2021

## 1 Introduction

The goal of this seminar is understanding how Paxos algorithm works, implementing it in Erlang and conducting a series of experiments to prove its functionality and see how it reacts to changes in various parameters and paradigms.

Paxos is a family of distributed algorithms for solving consensus in a network of unreliable or fallible processors. The Paxos algorithm is based on a *simple majority* rule which ensures that only consistent resulting values can be achieved. Essentially, Paxos compares each write request to a proposal. The Paxos protocol has the following entities:

a) **Proposers:** Proposers receive requests (values/proposals) from clients and try to convince acceptors to accept their proposed values.

b) **Acceptors:** The Acceptors accept certain proposed values from proposers and let proposers know if something else was accepted. A response represents a vote for a particular proposal.

c) **Learners:** Learners announce the outcome. In this seminar this entities will be dismissed and thus, we will only have Acceptors and Proposers.

Each proposal can be broken down into two phases: phase 1 (*Prepare & Promise*) and phase 2 (*Accept & Accepted*). Proposers interact with the acceptors twice.

1. **Phase 1:** Proposer selects a proposal number n and asks all acceptors to accept that prepare request. If acceptors receive a *prepare requests* with number n and the value of n is greater than the number of all *prepare requests* it has already responded to, then it will promise that no proposal with a number less than n is accepted.
2. **Phase 2:** If the Proposer receives a response from the majority of Acceptors for its *prepare requests* (also as n), then it sends an accept request for the proposal with the number n and the value as v to Acceptors, where v is the value of the proposal with the highest number in the response received. If acceptors receive an *accept request* with a number n, it can accept the proposal as long as it has not responded to a *prepare request* with a number greater than n.

The main idea of the protocol is: the proposer first learns the latest content of the proposal from the majority of Acceptors and then forms a new *accept request* based on the learned proposal with the largest number. If the proposal is accepted/voted by the majority of acceptors, it is passed. That is why, it is also known as the majority protocol.

# 2 Experiments

We conducted some experiments to analyze how Paxos reacts. We did changes as: putting delays on the acceptor promise and vote messages, avoiding sorry messages, droping promise and vote messages from the acceptor, increase the acceptors and proposers quantity, create prorposers/acceptors in a remote Erlang instance and, finally, testing the fault tolerance of the process.

For all the experiments we used 1ms of delay in all the proposers.

## 2.1 Delaying the acceptors' promise and vote messages

For the first experiment we are going to delay the sending of promise or vote messages of the acceptors separately. We analyzed how Paxy time increased with the time delay (Figure 1) and how rounds increased with delay (Figure 2).

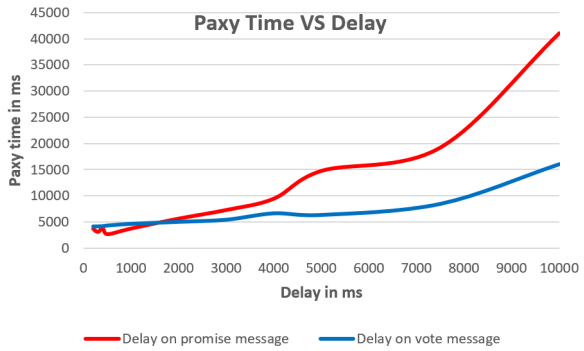


Figure 1: Total Paxy time vs delay applied to the promise/vote messages.

Clearly, we observe that increasing the delay in the promise messages has a bigger impact in the performance of the algorithm than applying the same amount of delay in the vote messages. That makes sense due that increasing the time of the promise or vote messages makes the proposers abort the procedure (because of the timeout time of 2000ms in the proposers) and this translates in a larger amount of rounds and thus, a larger amount of time.

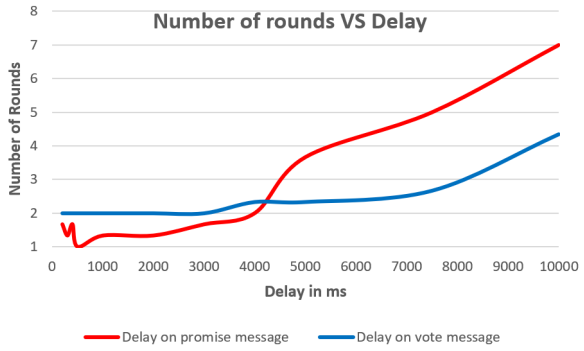


Figure 2: Number of rounds vs delay applied to the promise/vote messages.

Regarding the number of rounds, we observe that delaying the promise message, overall, has more rounds as a result. That means that, as in Paxy time graph (Figure 1), delaying promise messages worsens the performance of Paxos more than delaying vote messages. However we can also see that, at the beginning, with the same delays applied, the system does more rounds (a minimum of 2 rounds) if the delay is applyed to the vote messages. Is not until the delay is 4000ms (approximately) that the "delay on promise mesages" surpasses in number of rounds the "delay in vote messages".

The amount of time to reach consensus seems to increase proportionally to the number of rounds that the algorithm has to do in order to have a winner of the election.

## 2.2 Avoiding sorry messages

Avoiding sorry messages, overall, does not vary the time spent until consensus nor the number of rounds needed to finish the algorithm. The explanation is that the functionality of those messages in the vanilla version of the program is not implemented. As it will be shown in section 2.7, where the code will be improved in order to give utility to the sorry messages, those will make the algorithm faster.

## 2.3 Dropping acceptors' promise and vote messages

We dropped randomly certain number of promise/vote messages (separately) and made graphs of the time and rounds spent until a consensus is reached. The results are the following.

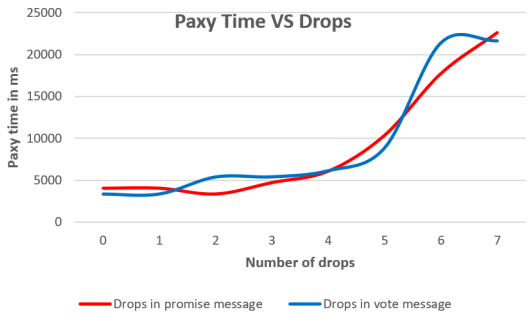


Figure 3: Paxy time vs number of drops of the promise/vote messages per each 10 messages.

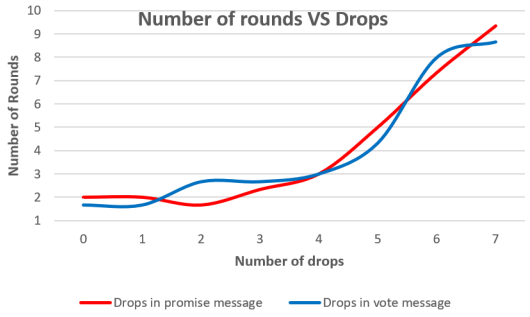


Figure 4: Number of rounds vs number of drops of the promise/vote messages per each 10 messages.

As in the previous experiment, the time and rounds spent until consensus is higher when the drop probability is increased. However, this time the type of message is not that important. Both, promise and vote messages, follow (approximately) the same increase in terms of time and rounds when drop chance is increased.

Note that we did not experiment with 8 or 9 drops because the process had a lot of problems to finish. That makes sense since approaching the 10 messages dropped means loosing all interaction from the side of the acceptors. Reaching 10 drops out of 10 messages (no matter what kind of message) would mean infinite time to finish.

## 2.4 Increasing the number of proposers or acceptors

The objective of this experiment is show how the algorithm reacts when the number of proposers or accepptors is increased. For this, we did 2 experiments separately. One experiment increasing the acceptors and leaving 3 proposers (Figure 5), and other increasing the proposers and leaving 5 acceptors (Figure 6). Then we did a graph showing the rounds it takes to reach consensus compared to the number of acceptors/proposers (Figure 7).


Figure 5: Gui showing consensus with acceptors increased


Figure 6: Gui showing consensus with proposers increased

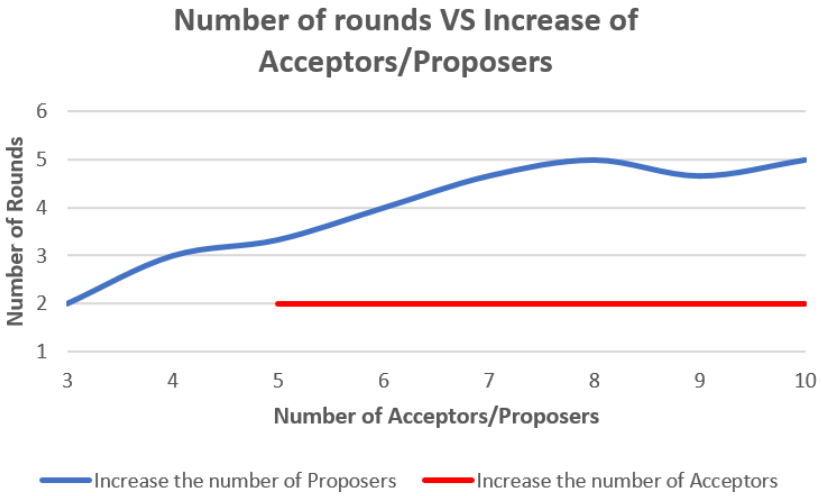


Figure 7: Number of rounds vs Number of Acceptors/Proposers

The graph shows that the increase of acceptors does not affect the performance of the algorithm. The number of rounds was always normal, as the rounds experienced in the vanilla version of the process. On the other hand, increasing the number of proposers makes a difference in terms of maximum amount of rounds. Increasing the number of proposers increases the number of rounds too.

## 2.5 Remote proposers and acceptors

In this experiment we try to deploy the acceptors and proposers nodes in a different remote instances. The following figure shows that the consensus is achieved.

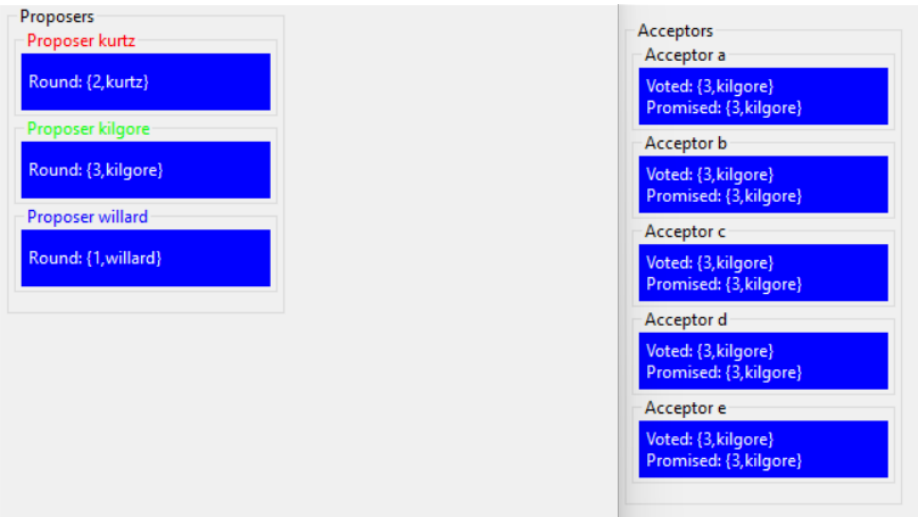


Figure 8: Consensus between remote acceptors and proposers.

With the following commands we can run the 3 different instances. One for the acceptors, paxy-acc, one for the proposers, paxy-pro and the last one, paxy-adm, will be instance that will deploy all the remote acceptors and proposers nodes.

`erl -name paxy-adm@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

```
paxy:start([1,1,1], 'paxy-acc@127.0.0.1', 'paxy-pro@127.0.0.1').
```

To implement the code for the remote acceptors and remote proposers we have modified a little bit the "gui.erl" file. Firstly, we have create to new functions that will initialize the window of the gui. These functions receive as parameter the acceptors/proposers and their size, the way to be printed in the gui.

```
1 start_acceptors(Acceptors, AccPanelHeight) ->
2   State = make_window_acceptors(Acceptors, AccPanelHeight),
```

```
  gui_acceptors(State).

start_proposers(Proposers, PropPanelHeight) ->
  State = make_window_proposers(Proposers, PropPanelHeight),
  gui_proposers(State).
```

Notice that we also split all the functions that were related for the sending messages between the paxy and gui into two different functions, i.e. `make_window` and gui now are: `make_window_acceptors`, `make_windows_proposers`, `gui_acceptors` and `gui_proposers`.

Secondly, we have made the deploy of proposers and acceptors in the different remote nodes, by making two spawns in the start function of the paxy.erl file. These spawns specify the address location where the deploy will take place. For the deploy of the remote proposers it receives the Sleep parameter and the address (paxy-acc@127.0.0.1), this last one to know where will be the acceptors located and their respective names. For the deploy of the remote acceptors it doesn't receive any parameter. It updates the gui with the corresponding messages of the proposers.

```
start(Sleep, AccepNode, PropNode) ->
  spawn(PropNode, fun() -> remoteProposers(Sleep, AccepNode) end),
  spawn(AccepNode, fun() -> remoteAcceptors() end).

remoteProposers(Sleep, AccepNode) ->
  ProposerNames = [{"Proposer kurtz", ?RED}, {"Proposer kilgore", ?GREEN},
                   {"Proposer willard", ?BLUE}],
  PropInfo = [{kurtz, ?RED}, {kilgore, ?GREEN}, {willard, ?BLUE}],
  AccRegisterRemote = [{a, AccepNode}, {b, AccepNode}, {c, AccepNode}, {d, AccepNode}, {e,
     AccepNode}],
  PropPanelHeight = length(ProposerNames)*?InSizerMinHeight + 10,
  register(gui_proposers, spawn(fun() -> gui:start_proposers(ProposerNames,
    PropPanelHeight) end)),
  gui_proposers ! {reqStateProp, self()},
   receive
        {reqStateProp, State} ->
          {PropIds} = State,
          start_proposers(PropIds, PropInfo, AccRegisterRemote, Sleep, self()),
          wait_proposers(length(PropIds))
      end.

remoteAcceptors() ->
  AcceptorNames = ["Acceptor a", "Acceptor b", "Acceptor c", "Acceptor d",
                   "Acceptor e"],
  AccRegister = [a, b, c, d, e],
  proposers, PropNode !  accReg, AccRegister,AccPanelHeight =
    length(AcceptorNames*?InSizerMinHeight + 10,
  register(gui_acceptors, spawn(fun() -> gui:start_acceptors(AcceptorNames, AccPanelHeight
    ) end)),
  gui_acceptors ! {reqStateAccep, self()},
  receive
    {reqStateAccep, State} ->
      {AccIds} = State,
      start_acceptors(AccIds, AccRegister)
  end.
```

Finally, we modified the function to stop. Now, the function `stop(gui)` is split into `stop(gui_acceptors)` and `stop(gui_proposers)` in order to update the two different gui windows.

```
stop() ->
  stop(a),
  stop(b),
  stop(c),
  stop(d),
  stop(e),
  stop(gui_acceptors),
  stop(gui_proposers).
```

## 2.6 Testing the fault tolerance of the process

In this experiment we try to make a node crash to see how the system reacts. In Figure 9 we can observe how, even with a node down, the algorithm is able to reach consensus. The code will be explained during this section.

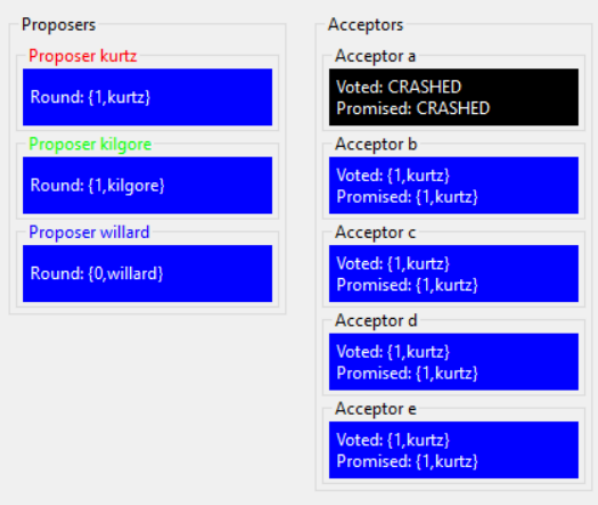


Figure 9: Even with a faulty node the consensus is reached.

For the implementation of the crash we have made a new function in the acceptor.erl called `storeAcceptor()`, this function receives, Name, Promised, Voted, Value and PanelId as parameters, and it calls its respective "pers" functions (open, store and close). We made this new function because we want to avoid to have the file open during the whole consensus.

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

This `storeAcceptor()` function it's called everytime the acceptors modify their new values.

In the init function of the acceptor the file is opened and the values of the Promised (Pr), Voted (Vot), Value (Val), PanelId(Pn) are read from the table 'Name'.

```
init(Name, PanelId) ->
  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 ->
        case Val == na of
          true ->Pn ! {updateAcc, "Voted: " ++ io_lib:format("~p", [Vot]),
            "Promised: " ++ io_lib:format("~p", [Pr]), {0,0,0}};
          false-> Pn ! {updateAcc, "Voted: " ++ io_lib:format("~p", [Vot]),
                        "Promised: " ++ io_lib:format("~p", [Pr]), Val}
        end,
        acceptor(Name, Pr, Vot, Val, Pn)
        end
  end.
```

In the paxy.erl file we made a new function call `crash()`. This function will receive the "Name" as a parameter and will implement the simulation of the node crashed.

```
crash(Name) ->
  case whereis(Name) of
    undefined ->
      ok;
    Pid ->timer:sleep(1000),
      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.
```

To implement the stop functionality in the program we have modified the function `stop(Name)` and we make a call for the pers delete function.

```
stop(Name) ->
  case whereis(Name) of
    undefined ->
      ok;
    Pid ->
      case Name == gui of
```

```
        false->
          pers:delete(Name)
        end,

      Pid ! stop
  end.
```

Finally to implement the fault tolerance experiment, we made the respective calls to the new crafted functions in the start function. The crash function, which it receives the name of the node to be crashed, will be called before the `wait_proposers()` and the stop() function at the end.

## 2.7 Improvement based in sorry messages

To make the experiment of improvement based on sorry messages we modified the original proposer code. The first thing we changed is the parameters that we send to the function collect and vote. Now, collect use 5 parameters instead of the initial 4, and vote use 3 parameters instead of the initial 2. This additional parameter of collect and vote, Sm (Sorry messages), is the same, and it's the value of the quorum.

```
collect(0, _, _, _, Proposal) ->
  {accepted, Proposal};
collect(_, 0, _, _, _) ->
  abort;
collect(N, Sm, Round, MaxVoted, Proposal) ->
  receive
    {promise, Round, _, na} ->
      collect(N-1, Sm, Round, MaxVoted, Proposal);
    {promise, Round, Voted, Value} ->
      case order:gr(Voted, MaxVoted) of
        true ->
          collect(N-1, Sm, Round, Voted, Value);
        false ->
          collect(N-1, Sm, Round, MaxVoted, Proposal)
      end;
    {promise, _, _,  _} ->
      collect(N, Sm, Round, MaxVoted, Proposal);
    {sorry, {prepare, Round}} ->
      collect(N, Sm-1, Round, MaxVoted, Proposal);
    {sorry, _} ->
      collect(N, Sm-1, Round, MaxVoted, Proposal)
  after ?timeout ->
    abort
  end.
```

In the collect function the value of the Sm is only decremented by one, if there's a sorry message. And if we receive the same amount of sorry messages as the value of the quorum then we abort.

```
vote(0, _, _) ->
  ok;
vote(_, 0, _) ->
  abort;
vote(N, Sm, Round) ->
  receive
```

```
7       {vote, Round} ->
8         vote(N-1, Sm, Round);
9       {vote, _} ->
10        vote(N, Sm, Round);
11      {sorry, {accept, Round}} ->
12        vote(N, Sm-1, Round);
13      {sorry, _} ->
14        vote(N, Sm-1, Round)
15    after ?timeout ->
16      abort
17  end.
```

As in the vote function, the value of the Sm is only decremented by one, if there's a sorry message. Also if the Sm value is equal to 0, it means that we have received the same amount of sorry message that the value of Quorum and then we have to abort.

```
Eshell V12.1  (abort with ^G)
1> paxy:start([1,1,1]).
[Gui] state requested
[Proposer kurtz] Phase 1: round {0,kurtz} proposal {255,0,0}
[Proposer kilgore] Phase 1: round {0,kilgore} proposal {0,255,0}
[Proposer willard] Phase 1: round {0,willard} proposal {0,0,255}
<0.97.0>
[Acceptor b] Phase 1: promised {0,kurtz} voted {0,0} colour na
[Acceptor a] Phase 1: promised {0,kurtz} voted {0,0} colour na
[Acceptor c] Phase 1: promised {0,kurtz} voted {0,0} colour na
[Acceptor d] Phase 1: promised {0,kurtz} voted {0,0} colour na
[Acceptor e] Phase 1: promised {0,kurtz} voted {0,0} colour na
[Proposer kurtz] Phase 2: round {0,kurtz} proposal {255,0,0} (was {255,0,0})
2> [Acceptor b] Phase 1: promised {0,willard} voted {0,0} colour na
2> [Acceptor a] Phase 1: promised {0,willard} voted {0,0} colour na
2> [Acceptor c] Phase 1: promised {0,willard} voted {0,0} colour na
2> [Proposer willard] Phase 2: round {0,willard} proposal {0,0,255} (was {0,0,255})
2> [Acceptor d] Phase 1: promised {0,willard} voted {0,0} colour na
2> [Acceptor e] Phase 1: promised {0,willard} voted {0,0} colour na
2> [Acceptor a] Phase 2: promised {0,willard} voted {0,willard} colour {0,0,255}
2> [Acceptor c] Phase 2: promised {0,willard} voted {0,willard} colour {0,0,255}
2> [Acceptor b] Phase 2: promised {0,willard} voted {0,willard} colour {0,0,255}
2> [Proposer willard] DECIDED {0,0,255} in round {0,willard} after 0 ms
2> [Acceptor d] Phase 2: promised {0,willard} voted {0,willard} colour {0,0,255}
2> [Acceptor e] Phase 2: promised {0,willard} voted {0,willard} colour {0,0,255}
2> [Proposer kilgore] Phase 1: round {1,kilgore} proposal {0,255,0}
2> [Acceptor b] Phase 1: promised {1,kilgore} voted {0,willard} colour {0,0,255}
2> [Acceptor a] Phase 1: promised {1,kilgore} voted {0,willard} colour {0,0,255}
2> [Acceptor c] Phase 1: promised {1,kilgore} voted {0,willard} colour {0,0,255}
2> [Acceptor d] Phase 1: promised {1,kilgore} voted {0,willard} colour {0,0,255}
2> [Acceptor e] Phase 1: promised {1,kilgore} voted {0,willard} colour {0,0,255}
2> [Proposer kilgore] Phase 2: round {1,kilgore} proposal {0,0,255} (was {0,255,0})
2> [Acceptor b] Phase 2: promised {1,kilgore} voted {1,kilgore} colour {0,0,255}
2> [Acceptor a] Phase 2: promised {1,kilgore} voted {1,kilgore} colour {0,0,255}
2> [Acceptor c] Phase 2: promised {1,kilgore} voted {1,kilgore} colour {0,0,255}
2> [Acceptor d] Phase 2: promised {1,kilgore} voted {1,kilgore} colour {0,0,255}
2> [Acceptor e] Phase 2: promised {1,kilgore} voted {1,kilgore} colour {0,0,255}
2> [Proposer kilgore] DECIDED {0,0,255} in round {1,kilgore} after 12 ms
2> [Proposer kurtz] Phase 1: round {1,kurtz} proposal {255,0,0}
2> [Acceptor a] Phase 1: promised {1,kurtz} voted {1,kilgore} colour {0,0,255}
2> [Acceptor b] Phase 1: promised {1,kurtz} voted {1,kilgore} colour {0,0,255}
2> [Acceptor c] Phase 1: promised {1,kurtz} voted {1,kilgore} colour {0,0,255}
2> [Acceptor d] Phase 1: promised {1,kurtz} voted {1,kilgore} colour {0,0,255}
2> [Acceptor e] Phase 1: promised {1,kurtz} voted {1,kilgore} colour {0,0,255}
2> [Proposer kurtz] Phase 2: round {1,kurtz} proposal {0,0,255} (was {255,0,0})
2> [Acceptor a] Phase 2: promised {1,kurtz} voted {1,kurtz} colour {0,0,255}
2> [Acceptor b] Phase 2: promised {1,kurtz} voted {1,kurtz} colour {0,0,255}
2> [Acceptor c] Phase 2: promised {1,kurtz} voted {1,kurtz} colour {0,0,255}
2> [Proposer kurtz] DECIDED {0,0,255} in round {1,kurtz} after 14 ms
2> [Acceptor d] Phase 2: promised {1,kurtz} voted {1,kurtz} colour {0,0,255}
2> [Acceptor e] Phase 2: promised {1,kurtz} voted {1,kurtz} colour {0,0,255}
2> [Paxy] Total elapsed time: 17 ms
2> [Gui] <0.80.0> closing window
2>
```

Figure 10: Output of paxy with the sorry improvement.

As we can see the time of the execution has improved by the use of the new parameter, Sm. This is because the sorry message now informs proposers that they they can not be the ones elected and

this makes them get out of the election process decreasing the execution time.

# 3 Open questions

In this section we are going to answer the open questions proposed in the guide of the seminar.

**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?**

In all of the cases conducted in the experiment of section 2.1, the algorithm achieves consensus. As mentioned before in the same section, when the delay makes some instances(vote or promise messages from acceptors) timeout (2000ms in the proposers), the proposers abort the procedure and thus, they have to start again. This translates in more rounds done by the algorithm. The more rounds the algorithm does, the more overall time it takes to reach consensus. This relation can be seen in figures 1 and 2. The increase of time and rounds is proportional. We can also see that the delay promise messages have more impact in terms of performance than the delay of the vote messages.

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

As mentioned in section 2.2, in the vanilla version of the algorithm code, avoiding sorry messages does not have any impact on the performance of the algorithm or the possibility of reaching a consensus. This is because the functionality of the sorry messages is not implemented. This functionality is implemented in section 2.7.

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

As seen in section 2.3, the percentage of messages dropped that makes the process unable to reach consensus or work properly, is from 70% to 100% of the messages. We also can conclude that the type of message is not relevant due that in both cases the algorithm reacts, almost, the same way. That is, taking longer times to reach the consensus and needing more rounds.

**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 seen in section 2.4, we can conclude that increasing the number of acceptors does not worsen the performance of the algorithm. This is because the proposers send their prepare message at the same time, no matter the number of acceptors present in the system. That means that every acceptor decides to promise the same thing in the first round. On the other hand, the increase in the number of proposers has a big impact in terms of the number of rounds reached to have a consensus. The happens because as there are more proposers it is more difficult for them to receive the vote messages and thus they have to do more rounds.

# 4 Personal opinion

This seminar has been useful to understand how Paxos works. The different experiments made us have a deeper knowledge of the algorithm and the implication of each component of the process (vote messages, promise messages, sorry messages, acceptors, proposers, etc.). We did not have enough time to get totally used to Erlang, which we think is a useful language in terms of investigation in distributed systems but we had not seen in our academic lives. So, summing up, we think it is a good way of learning the insides of the Paxos algorithm but we found difficult the programming language that is used on the seminar.

# 5 Appendix: Tables

In this section all tables used to plot the graphs of the seminar are attached.

| Delay on promise message | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Delay** | **Paxy Time** | | | | **Number of Rounds** | | | |
| **200** | 4227 | 4312 | 2234 | 3591 | 2 | 2 | 1 | 1,66666667 |
| **300** | 2381 | 2384 | 4367 | 3044 | 1 | 1 | 2 | 1,33333333 |
| **400** | 2307 | 4322 | 4521 | 3716,66667 | 1 | 2 | 2 | 1,66666667 |
| **500** | 2749 | 2515 | 2609 | 2624,33333 | 1 | 1 | 1 | 1 |
| **1000** | 3572 | 2443 | 5105 | 3706,66667 | 1 | 1 | 2 | 1,33333333 |
| **2000** | 4793 | 7315 | 4707 | 5605 | 1 | 2 | 1 | 1,33333333 |
| **3000** | 8511 | 4907 | 8351 | 7256,33333 | 2 | 1 | 2 | 1,66666667 |
| **4000** | 11162 | 4734 | 12372 | 9422,66667 | 3 | 1 | 2 | 2 |
| **5000** | 13208 | 10742 | 20295 | 14748,3333 | 3 | 3 | 5 | 3,66666667 |
| **7500** | 15524 | 22284 | 19833 | 19213,6667 | 4 | 6 | 5 | 5 |
| **10000** | 9371 | 21584 | 92414 | 41123 | 3 | 5 | 13 | 7 |

Table 1: Delay on promise message.

| Delay on vote message | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Delay** | **Paxy Time** | | | | **Number of Rounds** | | | |
| **200** | 4175 | 4201 | 4132 | 4169,33333 | 2 | 2 | 2 | 2 |
| **300** | 4176 | 4196 | 4176 | 4182,66667 | 2 | 2 | 2 | 2 |
| **400** | 4184 | 4219 | 4277 | 4226,66667 | 2 | 2 | 2 | 2 |
| **500** | 4332 | 4369 | 4349 | 4350 | 2 | 2 | 2 | 2 |
| **1000** | 4584 | 4739 | 4687 | 4670 | 2 | 2 | 2 | 2 |
| **2000** | 4758 | 5186 | 5228 | 5057,33333 | 2 | 2 | 2 | 2 |
| **3000** | 5108 | 5579 | 5683 | 5456,66667 | 2 | 2 | 2 | 2 |
| **4000** | 5640 | 8632 | 5775 | 6682,33333 | 2 | 3 | 2 | 2,33333333 |
| **5000** | 5122 | 7511 | 6485 | 6372,66667 | 2 | 3 | 2 | 2,33333333 |
| **7500** | 11276 | 8535 | 5657 | 8489,33333 | 4 | 2 | 2 | 2,66666667 |
| **10000** | 14647 | 16856 | 16764 | 16089 | 5 | 4 | 4 | 4,33333333 |

Table 2: Delay on vote message.

| Drops in promise message | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Drops** | **Paxy Time** | | | | **Number of Rounds** | | | |
| **0** | 4068 | 4061 | 4071 | 4066,66667 | 2 | 2 | 2 | 2 |
| **1** | 4068 | 4076 | 4078 | 4074 | 2 | 2 | 2 | 2 |
| **2** | 4078 | 4048 | 2019 | 3381,66667 | 2 | 2 | 1 | 1,66666667 |
| **3** | 6115 | 4046 | 4074 | 4745 | 3 | 2 | 2 | 2,33333333 |
| **4** | 4076 | 8146 | 6087 | 6103 | 2 | 4 | 3 | 3 |
| **5** | 6094 | 14765 | 10291 | 10383,3333 | 3 | 7 | 5 | 5 |
| **6** | 28281 | 10262 | 14681 | 17741,3333 | 10 | 5 | 7 | 7,33333333 |
| **7** | 14469 | 24737 | 28637 | 22614,3333 | 7 | 10 | 11 | 9,33333333 |

Table 3: Drops in promise message.

| Drops in vote message | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Drops** | **Paxy Time** | | | | **Number of Rounds** | | | |
| **0** | 2014 | 4061 | 4054 | 3376,33333 | 1 | 2 | 2 | 1,66666667 |
| **1** | 4069 | 4058 | 2029 | 3385,33333 | 2 | 2 | 1 | 1,66666667 |
| **2** | 4060 | 6098 | 6101 | 5419,66667 | 2 | 3 | 3 | 2,66666667 |
| **3** | 4068 | 8156 | 4062 | 5428,66667 | 2 | 4 | 2 | 2,66666667 |
| **4** | 4070 | 10329 | 4083 | 6160,66667 | 2 | 5 | 2 | 3 |
| **5** | 10235 | 10309 | 6084 | 8876 | 5 | 5 | 3 | 4,33333333 |
| **6** | 36700 | 21324 | 6108 | 21377,3333 | 12 | 9 | 3 | 8 |
| **7** | 17976 | 10185 | 36679 | 21613,3333 | 8 | 5 | 13 | 8,66666667 |

Table 4: Drops in vote message

| Increase the number of Acceptors | | | | |
|---|---|---|---|---|
| **# Acceptors** | **Number of rounds** | | | |
| **5** | 2 | 2 | 2 | 2 |
| **6** | 2 | 2 | 2 | 2 |
| **7** | 2 | 2 | 2 | 2 |
| **8** | 2 | 2 | 2 | 2 |
| **9** | 2 | 2 | 2 | 2 |
| **10** | 2 | 2 | 2 | 2 |

Table 5: Rounds vs Increase of acceptors

| Increase the number of Proposers | | | | |
|---|---|---|---|---|
| **# Proposers** | **Paxy Time** | | | |
| **3** | 2 | 2 | 2 | 2 |
| **4** | 3 | 3 | 3 | 3 |
| **5** | 3 | 4 | 3 | 3,33333333 |
| **6** | 4 | 4 | 4 | 4 |
| **7** | 4 | 5 | 5 | 4,66666667 |
| **8** | 6 | 4 | 5 | 5 |
| **9** | 5 | 4 | 5 | 4,66666667 |
| **10** | 4 | 5 | 6 | 5 |

Table 6: Rounds vs Increase of proposers