Seminar Report: Paxy

Ferran Delgà Fernández **Roberto Meroni**

Nicolás Zhilie Zhao

December 4, 2023

1 Introduction

Figure 1: Paxos, Greece

The objective of the Paxy seminar is to provide a hands-on understanding of the Paxos consensus algorithm among students. This involves actively engaging students in the implementation process, using Erlang to execute a basic version of the Paxos algorithm protocol. We will first go over the code layout, illustrate the experiments through tables and plots, and finally give our personal opinion about the seminar in the context of the Distributed Systems module.

2 Source code layout

The source code is organized into four main directories, each corresponding to the Erlang code used to address a specific section of the seminar:

- `normal`: Contains the finalized code obtained by completing the provided Erlang code template from the seminar. This corresponds to the *Paxos* section of the seminar hands-on.

- `local`: Includes the code used for running experiments outlined in the *Experiment* section of the seminar hands-on. This directory should not be relevant of the evaluation of the seminar, since we only added primitives for adding message delay and timeout which are provided in the hands-on. Furthermore, there are other utility functions we added in module `utils` which are responsible to read values from environment variables.
- `distributed`: Encompasses the code employed to create different proposers and acceptors on distinct nodes. This corresponds to the last experiment of the *Experiment* section.
- `fault_tolerance`: This directory corresponds to the final solution implemented to enable an acceptor to crash and recover its previous state.
- `sorry`: Corresponds to the solution code for implementing the *Improvement based on sorry messages* section of the seminar hands-on.

3 Experiments

In order to conduct the experiments, we adapted the code to expedite the measurement process. The modified code is accessible in the `local` directory of the submitted source code and reads input data for the experiments (delay, drop rate, number of proposers and acceptors) from the `run.sh` bash script which will run multiple times the program with different input data and collect the results of the experiments in a single csv file. For each experiment, we executed it five times to extract the average value. Due to practical time consumption reasons, we did not perform it more than five times.

3.1 Different delays for promise and/or receive messages

For taking the measurements, we set the `timeout` to a fixed value of 2000ms.

As illustrated in Figure 2, the average number of rounds remains approximately 2 for delays smaller than 3700. However, it starts to increase for values larger than 3700. Notably, for larger delays, the average number of rounds needed for consensus spikes up, indicating a prolonged termination process. Despite the extended duration for larger delays, we have confidence in the eventual consensus of our implementation. In fact, due to the randomized nature of the delays, the consensus should always be eventually reached.

It is important to note that in our delayed messages implementation, the delay is a random number between 0 and `delay`. In other words, the `delay` variable sets the upper bound of delay rather than its exact value. On the other hand, the value of `timeout` is fixed, establishing the time limit for message reception. Increasing the delay, especially over the `timeout` threshold, decreases the chances that a message received by proposers is promptly considered for the quorum, resulting in reaching majority slower.

In summary, the key determinant influencing the number of rounds is not the absolute values of `delay` and `timeout`, but rather the relative difference between them. When the `delay` is less than the `timeout`, the number of rounds tends to stabilize at 1, 2, or 3. Conversely, if the `timeout` is smaller than the `delay`, which can be achieved by increasing the `delay` or decreasing the

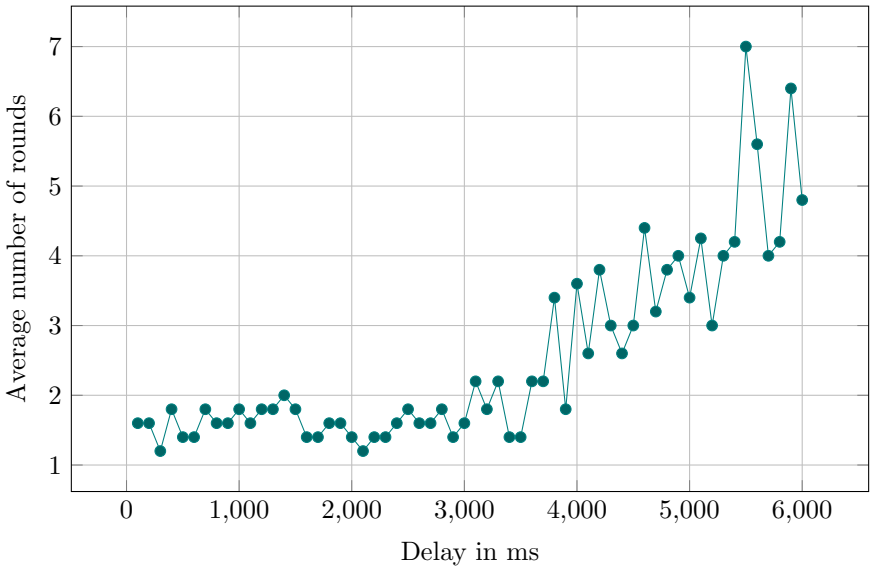


Figure 2: Alteration in number of rounds depending on the message delay

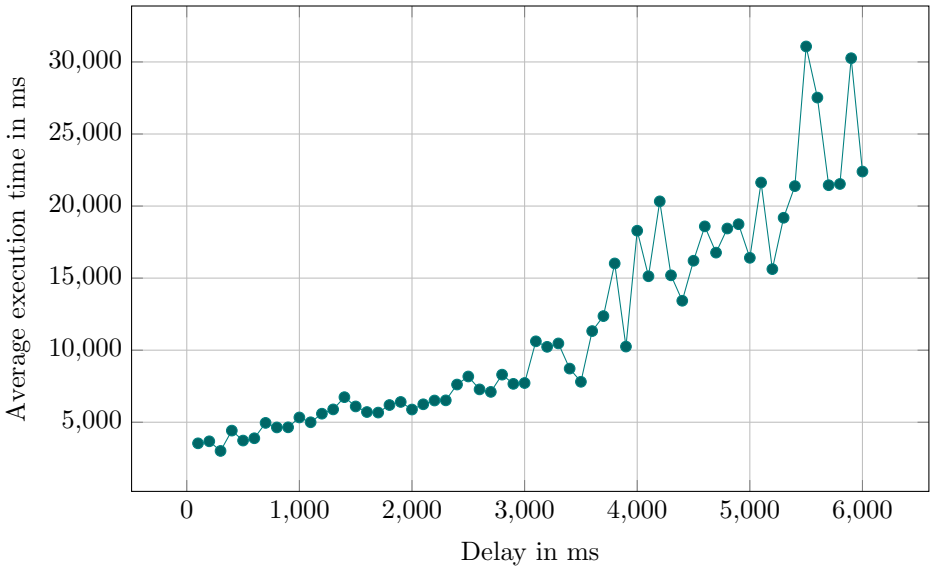


Figure 3: Alteration in execution time depending on the message delay

`timeout`, a larger number of rounds is expected for consensus to be reached. In contrast, the execution time is both impacted by the absolute values of `delay` and `timeout` and the relative difference between them, with the latter having a predominant role in the time increase. The previous observation is supported from the measurements plotted in Figure 3.

3.2 Removal of sorry messages

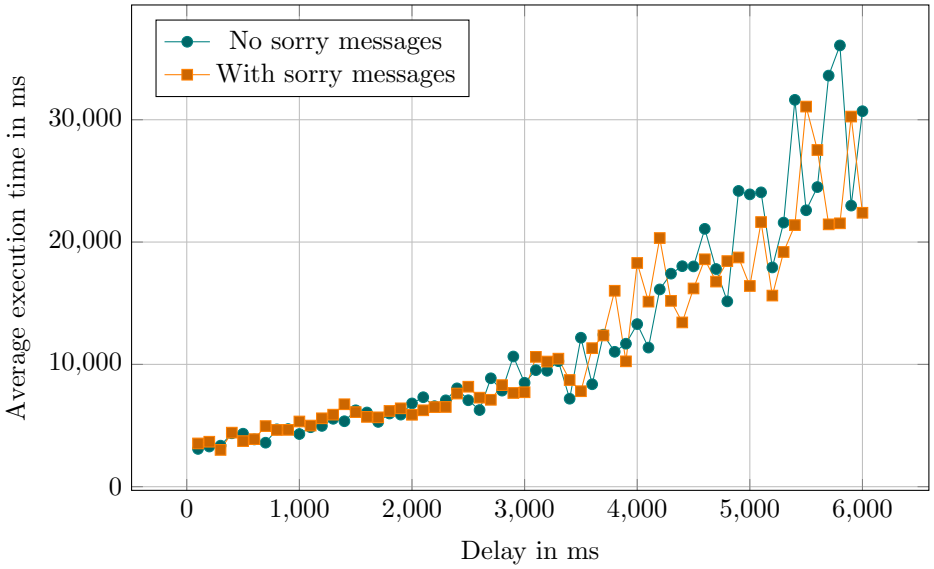


Figure 4: Impact of removing sorry messages on execution time given a certain message delay

As illustrated in Figure 4, the sorry messages don't significantly affect the execution time nor the achievement of consensus. Even if we are sending (from the acceptors) and receiving (from the proposers) `sorry` messages, we are not changing the behavior of the processes. After sending a `sorry` message, an acceptor simply continue is normal execution. After receiving a `sorry` message, a proposer (both in the `promise` and the `vote`) simply starts collecting messages again, without any changes in its state. The `sorry` messages may **minimally** increase the execution time, since sending a `sorry` message takes a very short amount of time. An improvement in the algorithm could be made if, when a proposer receives a quorum of `sorry` messages (related to a specific round) where they promised to not vote for a round less than N (we could even consider promised values larger than N, to include cases where messages got lost), the proposer doesn't try to send `promise` messages for round less than N, since a quorum of acceptors is not going to vote for it anyway. A simpler version of this improvement is asked in the seminar hands-on and we show the comparison before and after the improvement in Section 5.

3.3 Dropping promise and/or vote messages

We implemented dropping messages using a function that takes the proposer ID and the message. Using the `get_drop` function, we can modify the drop ratio and check how omitting messages affects execution as it is shown in Code 1 and 2.

```
get_drop() ->
```

```
  P = rand:uniform(100),
  P =< list_to_integer(os:getenv("drop")).
```

Code 1: Function to get the drop values from terminal. It returns true or false.

```
send_or_drop(Proposer, message) ->
  P = rand:uniform(10),
  if P =< ?drop ->
    io:format("message dropped~n"),
    false;
      true -> Proposer ! message, true
  end.
```

Code 2: Drop messages

To know how the drop ratio affects the number of rounds required to reach consensus, we increased the drop value from 5 to 95 in steps of 5. We saw that, as expected, when the probability to get a message drop increases, the time to reach consensus also increases, until we get an execution time too high to measure. This is illustrated in Figure 5.

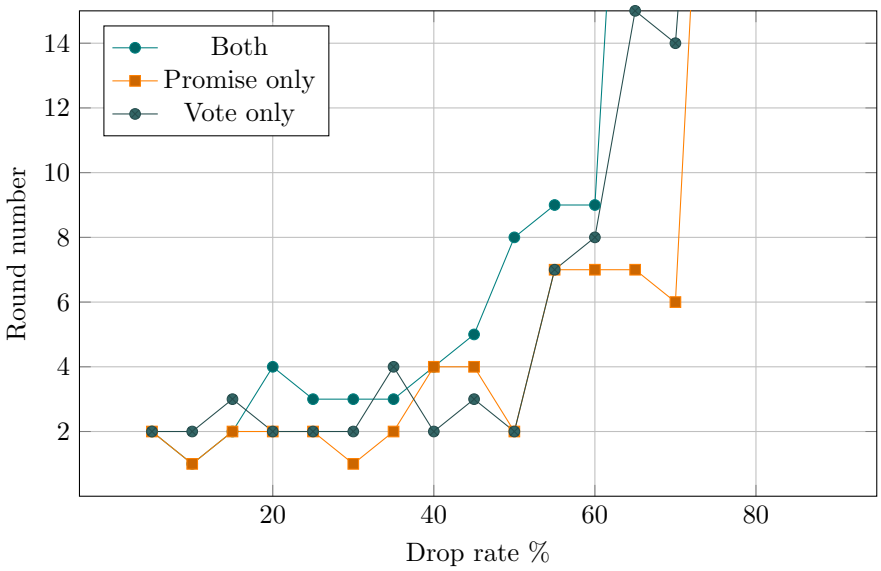


Figure 5: Alteration of round numbers depending on the message drop rate

Two observations need to be done in this part:

- Firstly, there is a distinct behavioral contrast between drop rates below 50% and those exceeding this threshold. In all instances, round numbers are minimal when the drop rate $< 50\%$, and escalate when $> 50\%$.
- The second point to consider is what occurs when we exclude both promise and vote messages. Achieving consensus within a reasonable number of rounds becomes more challenging in this scenario. The number of rounds shows an early increase even prior to the 60% mark and becomes immeasurable when the drop rate approaches 65%. In contrast, exclusively

dropping either only promise messages or only vote messages yields comparable results, reaching consensus with a decent number of rounds until we reach a dropping ratio $> 75\%$.

3.4 Increasing the number of acceptors and proposers

Number of acceptors \ Number of proposers	3	4	5	6	7	8
5	2927.8	4942.8	4546	6155.4	6979	8232.2
6	2930.6	4944.6	6177.8	4538.4	8628.8	7808.6
7	2926.8	3736	6170.6	4945.2	6157.2	6988.6
8	3735.6	3730.2	4558.2	6571.4	6973.4	6574
9	4139	4136.8	4544.2	5355.4	6581.8	6969.6
10	3735.8	4139.8	4953.4	5360.8	6989.4	6980.8

Table 1: Average execution time in ms for each combination of number of proposers and acceptors

Number of acceptors \ Number of proposers	3	4	5	6	7	8
5	1.4	2.4	2.2	3.0	3.4	4.0
6	1.4	2.4	3.0	2.2	4.2	3.8
7	1.4	1.8	3.0	2.4	3.0	3.4
8	1.8	1.8	2.2	3.2	3.4	3.6
9	2.0	2.0	2.2	2.6	3.2	3.4
10	1.8	2.0	2.4	2.6	3.4	3.4

Table 2: Average number of rounds for each combination of number of proposers and acceptors

As illustrated in Tables 1 and 2, the time impact of increasing the number of acceptors is not substantial. Given that the experiments have been done without any delay from the acceptors, every message is sent immediately, so we have more vote frequency. This is not considerably improving the execution time or the number of rounds because also the required quorum is proportionally increasing. However, raising the number of proposers influences both the number of rounds and the execution time, because in order to reach the consensus we need all the the proposers to receive a majority of validating votes. To clarify, as we move horizontally from left to right in the same table row, there is a noticeable increase, while moving from top to bottom within the same column does not reveal significant changes.

3.5 Proposers creation through remote Erlang instance

In order to implement the proposers and acceptors in distributed Erlang instances, we have implemented some changes in the `paxy.erl` file. We have defined two new variables:

```
-define(ProposerNode, 'paxy-pro@ferrandf').
-define(AcceptorNode,'paxy-acc@ferrandf').
```

Code 3: Definition of the remote instances

This time, we have to assign the node where the acceptors belong in the acceptors name. We have to initialize them by doing:

```
AccRegister = [{X, ?AcceptorNode} ||X <- [homer, marge, bart, lisa,
    maggie]],
```

and spawn in the `AcceptorNode`. In a similar way, we need to spawn the proposers in the `ProposerNode`:

```
...
spawn(?AcceptorNode, fun() ->
        start_acceptors(AccIds, AccRegister)
      end),
spawn(?ProposerNode, fun() ->
    Begin = erlang:monotonic_time(),
    start_proposers(PropIds, PropInfo, AccRegister, Sleep, self()),
...
```

We also have to implement a different `stop()` function to stop all the proposers and the acceptors that are not in the same node:

```
stop(Name) ->
  case whereis(Name) of
    undefined ->
        io:format("Deleting in ~w the acceptor ~w~n", [?AcceptorNode,
    Name]);
        {Name, ?AcceptorNode} ! stop;
    Pid ->
      Pid ! stop
end.
```

Code 4: Unique `stop()` function to kill both proposers and acceptors (in a different node)

Let's do the experiment:

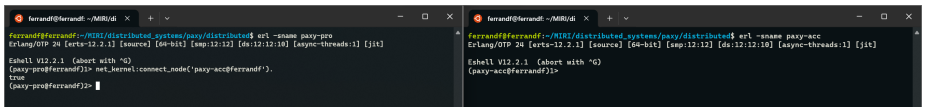


Figure 6: Check the connection between instances

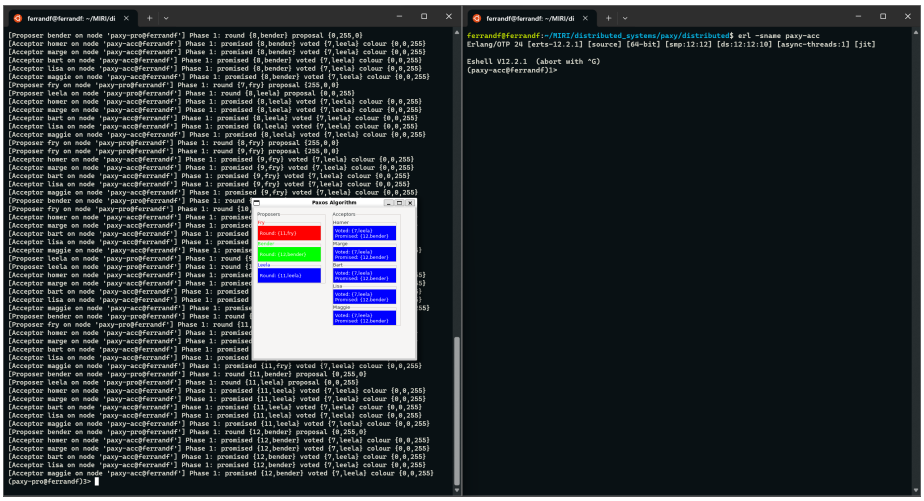
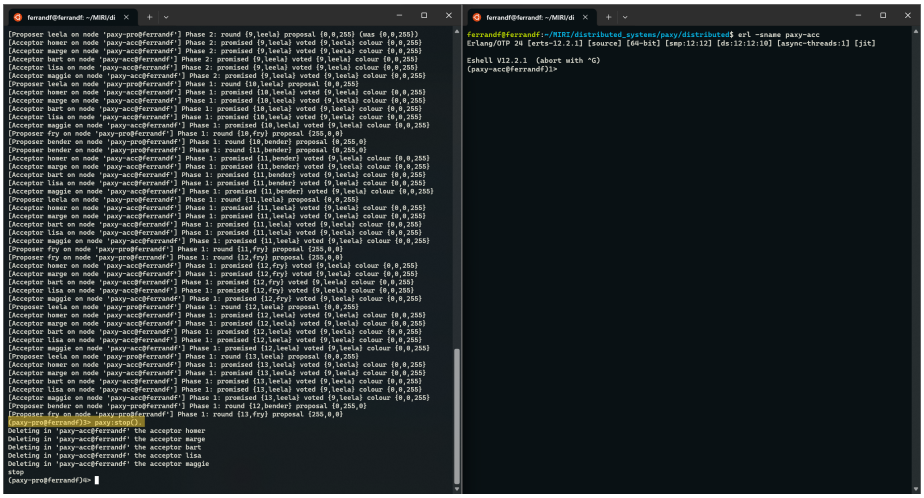


Figure 7: Execution running in both *paxy-pro* and *paxy-acc*. We added from which `node()` is a message sent, the information is displayed in the *paxy-pro* window.

We now can execute the `stop()` function in order to kill proposers, acceptors and the gui:



Another interesting experiment is see what happens if we abort the acceptors node and we try to start as before. The expected behaviour is that the acceptors should not participate in the execution:

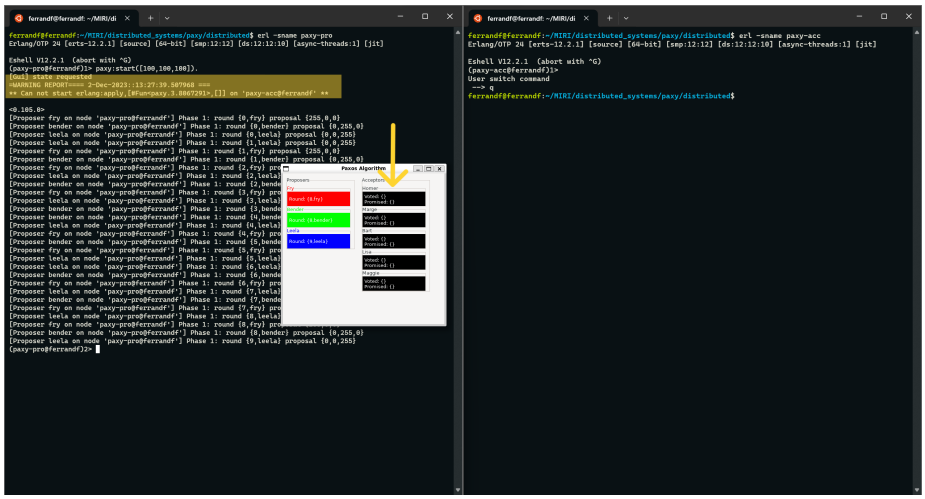


Figure 8: Note the *gui* shows that the acceptors are not there. Also, we get a warning report from erlang, that tells us it was not able to find *paxy-acc*

4 Fault tolerance

As stated in the assignment, we use the `pers` module to store the state of the acceptor. The main objective is to recover the state of an acceptor if a crash occurs. We need to satisfy:

1. The acceptor must save the most up-to-date version of its state.
2. We only need to save the state when it has been modified.
3. The acceptor's state must be removed from the disk when finishes its execution successfully.
4. The GUI panel of the acceptor must always show its current state.

An acceptor needs to remember the highest round it promised for, the highest round it voted for with the corresponding value and, in our case, the panel ID. In order to satisfy all the requirements, we placed the `pers:store` call:

- When we receive a `prepare, Proposer, Round` message from the *proposer* and `order:gr(Round, Promised)` is **`true`**. In this case we know we are going to send a promise back to the proposer promising a higher round number than the one stored.
- When we receive an {`accept, Proposer, Round, Proposal`}, `Round` is greater or equal than `Promised` and also `Round` is greater or equal than `Voted`. In this case we know we are going to send a message back to the proposer voting the round received and our state will be changed, since now our highest voted round is that `Round` and our voted value is the value related to this vote.

- We also need to store the state of the acceptor the first time it has been created, so that we save his `PanelId`. Notice that we at the event of a restart we register again the acceptor with `register(Name, acceptor:start(Name, na))`. This will send a `na` value to the `init()` function of the acceptor, so we should recover our initial `PanelId` by reading it from the persistent storage.

It's relevant to note that we intentionally store the modified state of the acceptor before sending the `vote` or `promise` messages to the proposer. If we perform the storage **after** the message sending, the following situation might happen: an acceptor might promise/vote for a proposer's round X and crash before storing its state. The acceptor then recovers from the crash and recovers its state from storage without acknowledging that X should be its promised/voted value, and if it receives an prepare/accept message with round number Y lower than X, but still higher than its previous promised/voted value, it proceeds to promise/vote Y. This makes the algorithm conceptually wrong, as it allows an acceptor to vote for a round lower than the promised and to not know what is its highest voted value.

There is no need to save the state when we know we are sending a `sorry` message, since the state of the acceptor doesn't change.

We also need to take into account that the acceptor could respond after some time crashed. This time, it should remember its state.

When we initialize the acceptor, it should open the file with its name and read it in order to get the {`Promised, Voted, Value, Pn`} info. We have the following situations:

- It is the first time we initialize the *acceptor*. In this case, the read from the file will return {`order:null(), order:null(), na, na`}. In other words, we should store in the persistent storage the `PanelId` assigned from the `Main`.

- We have restarted the acceptor after a crash. Obviously, we want to read our file and restore our last state. In this case, the only thing we need to take into account is that in restart, `Pn` is the correct value and not `PanelId` since when restarting an acceptor, the `Main` sends `PanelId` with the value `na` to indicate that the acceptor has recovered from a crash and should recover its state from persistent storage.

We added one extra message the *acceptor* can receive in order to distinct the `stop` situation and the `done` [1] one. In either case we need to close the *acceptors* files and delete their content.

In the video we started an execution and simulated a crash using `paxy:crash(bart)`. The acceptor named `bart` crashes and the GUI is updated showing a `crash` message. When it restarts, it shows information about its state, which is the last one it had before crashing:

[1] The `done` message sending function is added in `fault_tolerant/paxy.erl`

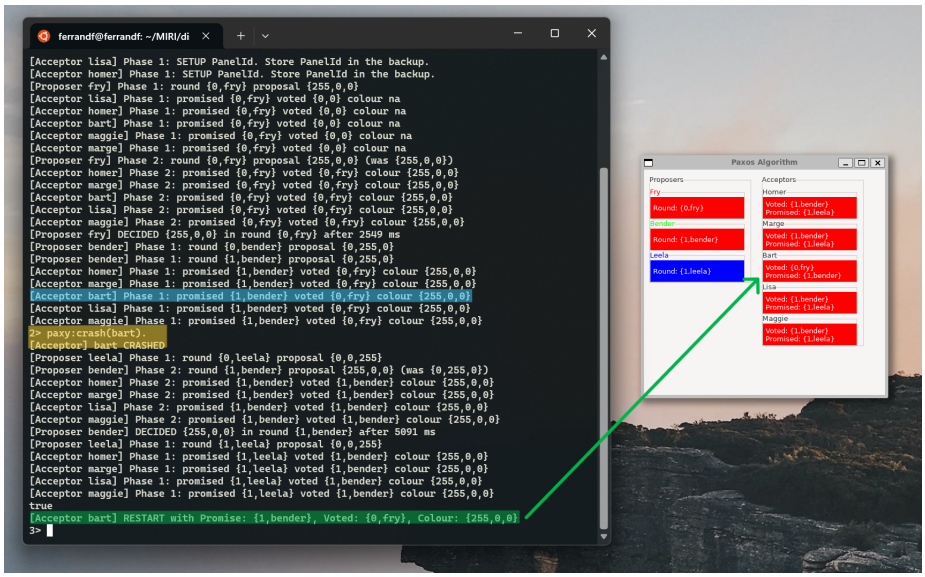


Figure 9: The blue highlighted text captures the last known state of the acceptor `bart`. In yellow we highlight when `bart` crashes. The green one shows the state at restart, which is the same we had on the blue part. We also note that the GUI has the correct restart state.

5 Improvement based on sorry messages

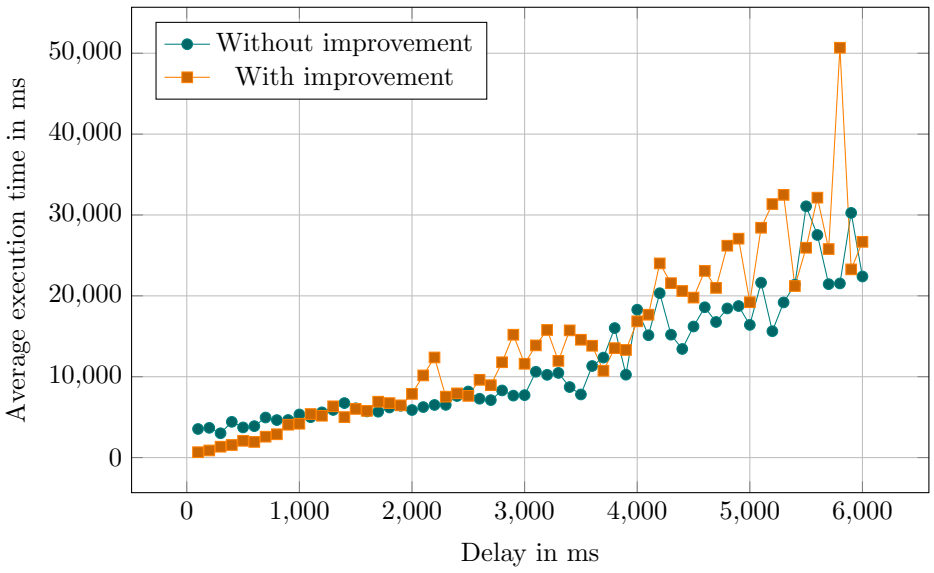


Figure 10: Alteration in execution time depending on the message delay

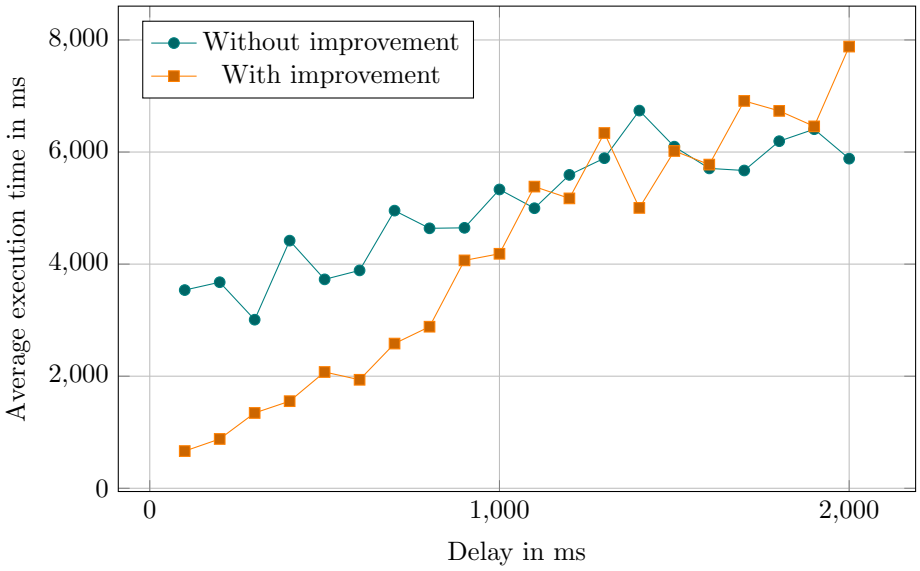


Figure 11: Fluctuation in execution time depending on the message delay

Until this point, when a proposer receives a `sorry` message, it simply ignores it and aborts the gathering of `promise` and `vote` messages once the predefined value of the `timeout` is surpassed. In contrast, our improved proposer (`sorry/proposer.erl`) can now reach a majority for `sorry` messages and can abort earlier. To test the improved code, we compare it with the results obtained from Section 3.1, specifically Figure 3.

For small delays (compared to timeout) we see time improvements as expected (Figure 11), since it's very likely that the sorry messages are gonna be received in the same round they have been sent. Therefore, a failing promise or vote round can be aborted before the timeout. For higher delays (compared to timeout) we actually see worse execution times (Figure 10).

6 Personal opinion

Figure 12: Antipaxos, **also** in Greece.

6.1 Ferran Delgà Fernández

The seminar proved to be both engaging and beneficial. It not only provided insights into the workings of Erlang but also emphasized the importance of understanding the algorithm for grasping the details of reaching consensus using PAXOS. The practical experiments allowed us to simulate and address real-world scenarios, noting their significance. I think both the seminar and such experiments should be kept for the upcoming course. I would appreciate more in-class sessions to get the code working as soon as possible so we could have more time to plan and execute the experiments.

6.2 Roberto Meroni

It seems to me that the structure of the seminar is too divisive between students that fully understood PAXOS and students that lack comprehension of some concepts of the algorithm. I assume that the role of the experiment section is to help students realize theirs misconceptions, but as long as they didn't make huge mistakes it's gonna be difficult to notice, since they are gonna analyze the results with the same misconceptions they had in mind. I understand that in this context is difficult to provide a better check system without giving out the solutions, so maybe it could be an idea to split the seminar in 2 parts and evaluate and correct the first part (Paxos and Experiments) before doing the second part (Fault Tolerance and Improvements). The students would be aware of what they didn't understand, have the opportunity to review it and apply it again in the second part.

6.3 Nicolás Zhilie Zhao

This seminar is useful for learning a basic implementation of the PAXOS consensus algorithm, however, I believe that some form of formal verification could

be used in order to formally prove the correctness of the algorithm. In the context of the subject I believe that getting started with some formal verification tools like TLA+ could be very interesting since Leslie Lamport himself was a big advocate of formal verification. Acknowledging the time constraints within the module, I understand that there may be limitations on in-depth exploration. Nevertheless, introducing students to TLA+ and similar tools can offer them an initial glimpse into the field of formal verification in distributed systems.