CPDS - Paxos Algorithm

Alfonso Brown Matthias Nadal Oscar Sandblom

November 2024

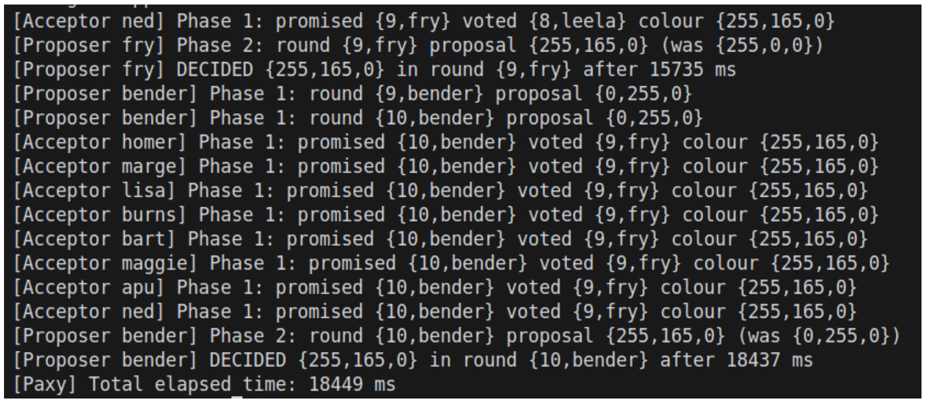
`https://github.com/Matnadal1/cpds_paxy.git`

1 Introduction

This report investigates the Paxos algorithm by performing several different experiments. The purpose of the report is to gain deeper knowledge about how various parameters influence the algorithm's behavior. The following experiments were built upon an existing codebase provided as part of the course materials.

1.1 Environment setup

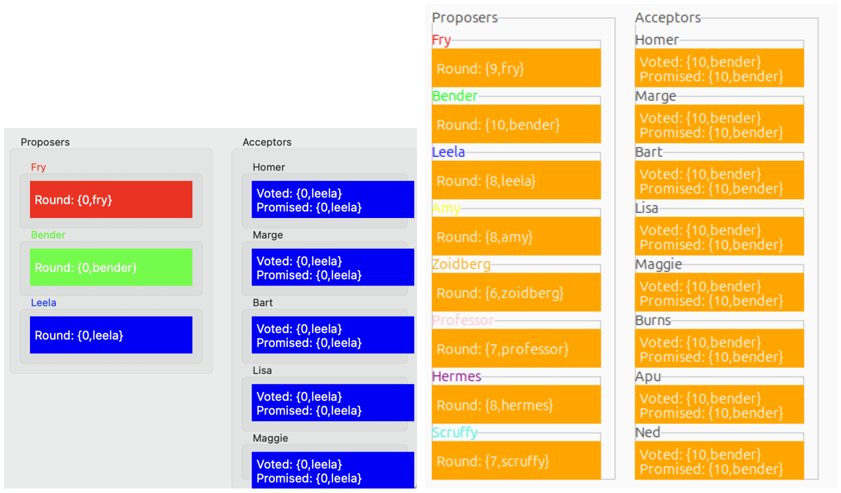
The provided codebase already included an implementation that output all information to the terminal, as shown in figure 1. This terminal output was later used, along with a testing program, to log the data for each test.



```
[Acceptor ned] Phase 1: promised {9,fry} voted {8,leela} colour {255,165,0}
[Proposer fry] Phase 2: round {9,fry} proposal {255,165,0} (was {255,0,0})
[Proposer fry] DECIDED {255,165,0} in round {9,fry} after 15735 ms
[Proposer bender] Phase 1: round {9,bender} proposal {0,255,0}
[Proposer bender] Phase 1: round {10,bender} proposal {0,255,0}
[Acceptor homer] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Acceptor marge] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Acceptor lisa] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Acceptor burns] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Acceptor bart] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Acceptor maggie] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Acceptor apu] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Acceptor ned] Phase 1: promised {10,bender} voted {9,fry} colour {255,165,0}
[Proposer bender] Phase 2: round {10,bender} proposal {255,165,0} (was {0,255,0})
[Proposer bender] DECIDED {255,165,0} in round {10,bender} after 18437 ms
[Paxy] Total elapsed time: 18449 ms
```

Figure 1: Terminal output from paxy algorithm.

The code base also included a working GUI that made it possible to follow the progress live of the algorithm. The GUI shows the proposers on the left, with their names in the color that they are requesting votes for; and the acceptors in the right. The boxes change colors whenever the acceptors and proposers accept a new value. This is shown in figure 2.



Proposers
Fry
Round: {0,fry}
Bender
Round: {0,bender}
Leela
Round: {0,leela}

Acceptors
Homer
Voted: {0,leela}
Promised: {0,leela}
Marge
Voted: {0,leela}
Promised: {0,leela}
Bart
Voted: {0,leela}
Promised: {0,leela}
Lisa
Voted: {0,leela}
Promised: {0,leela}
Maggie
Voted: {0,leela}
Promised: {0,leela}

Proposers
Fry
Round: {9,fry}
Bender
Round: {10,bender}
Leela
Round: {8,leela}
Amy
Round: {8,amy}
Zoidberg
Round: {6,zoidberg}
Professor
Round: {7,professor}
Hermes
Round: {8,hermes}
Scruffy
Round: {7,scruffy}

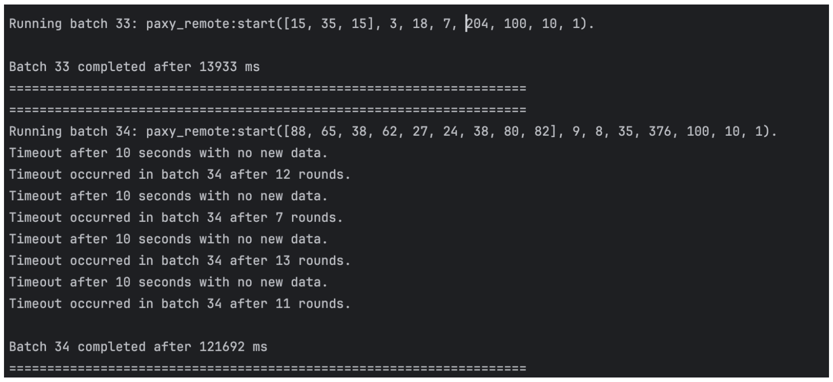
Acceptors
Homer
Voted: {10,bender}
Promised: {10,bender}
Marge
Voted: {10,bender}
Promised: {10,bender}
Bart
Voted: {10,bender}
Promised: {10,bender}
Lisa
Voted: {10,bender}
Promised: {10,bender}
Maggie
Voted: {10,bender}
Promised: {10,bender}
Burns
Voted: {10,bender}
Promised: {10,bender}
Apu
Voted: {10,bender}
Promised: {10,bender}
Ned
Voted: {10,bender}
Promised: {10,bender}

Figure 2: The GUI provided with the existing code base. To the left the starting state with 3 proposers and 5 acceptors. To the right a finishing consensus state for 8 proposers and 8 acceptors.

To do the experiments, we created a modular version of the paxy code, that can receive the following parameters when starting:

- **Sleep** Array with the ms it takes to each proposer to wake up.
- **Number of proposers** How many proposers participate out of a list of 11.
- **Number of acceptors** How many acceptors participate out of a list of 30.
- **Delay** Used to delay acceptor's messages uniformly at random from 0 to D.
- **Drop rate** Value from 0 to 100 containing the percentage of messages that the acceptors will fail to deliver.
- **Propser timeout** Time that the proposer will wait for quorum messages before attempting to start a new round.
- **Proposer Backoff** Time that the proposers will wait before starting a new round.

A Python script was also created that starts an Erlang process and runs a set of predefined tests n times each, to try to get average results with some



```
Running batch 33: paxy_remote:start([15, 35, 15], 3, 18, 7, 204, 100, 10, 1).

Batch 33 completed after 13933 ms
====================================================================
====================================================================
Running batch 34: paxy_remote:start([88, 65, 38, 62, 27, 24, 38, 80, 82], 9, 8, 35, 376, 100, 10, 1).
Timeout after 10 seconds with no new data.
Timeout occurred in batch 34 after 12 rounds.
Timeout after 10 seconds with no new data.
Timeout occurred in batch 34 after 7 rounds.
Timeout after 10 seconds with no new data.
Timeout occurred in batch 34 after 13 rounds.
Timeout after 10 seconds with no new data.
Timeout occurred in batch 34 after 11 rounds.

Batch 34 completed after 121692 ms
====================================================================
```

Figure 3: Terminal output from the python script showing the results of running two batches.

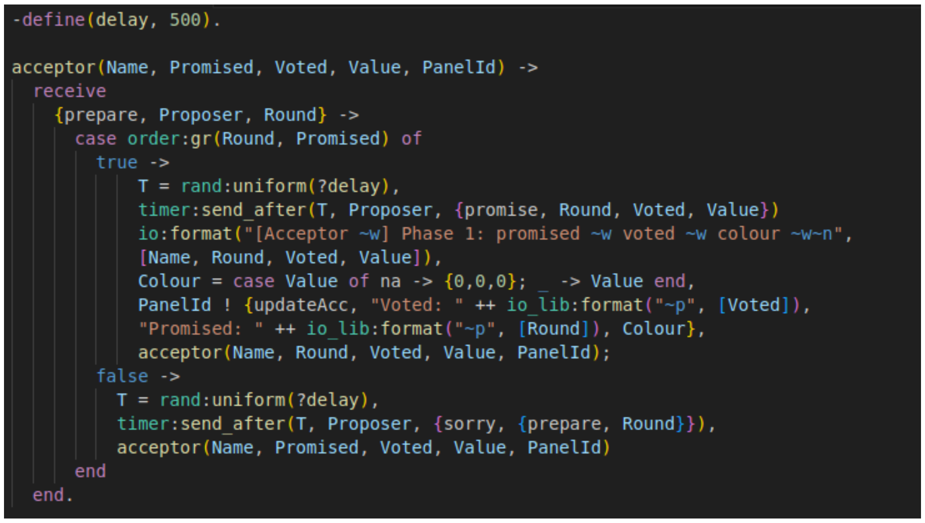
statistical significance. The Python script is also able to collect the results from the stdout of the erlang process, and output plots displaying the results. An example of the terminal output of the script is shown in figure 3.

Finally, to complete the analysis, from the same Python script we decided to record all the tests and results into a csv file, that we then imported into R studio to do the final analysis of how the outputs correlate to the inputs.

2 Experiments Results

2.1 Variation in delay

For the first experiment, we introduced random delays to simulate the time it takes for a message to travel from an acceptor to a proposer. The code implementation for this experiment is shown in figure 4.



```
-define(delay, 500).

acceptor(Name, Promised, Voted, Value, PanelId) ->
  receive
    {prepare, Proposer, Round} ->
      case order:gr(Round, Promised) of
        true ->
            T = rand:uniform(?delay),
            timer:send_after(T, Proposer, {promise, Round, Voted, Value})
            io:format("[Acceptor ~w] Phase 1: promised ~w voted ~w colour ~w~n",
            [Name, Round, Voted, Value]),
            Colour = case Value of na -> {0,0,0}; _ -> Value end,
            PanelId ! {updateAcc, "Voted: " ++ io_lib:format("~p", [Voted]),
            "Promised: " ++ io_lib:format("~p", [Round]), Colour},
            acceptor(Name, Round, Voted, Value, PanelId);
        false ->
          T = rand:uniform(?delay),
          timer:send_after(T, Proposer, {sorry, {prepare, Round}}),
          acceptor(Name, Promised, Voted, Value, PanelId)
      end
  end.
```

Figure 4: Delay implementation for acceptors.

The first thing we noticed is that the delay parameter does not have much effect on its own, and its importance is strictly relative to the proposer's timeout. For this reason, when plotting the results we decided to use the relative delay, defined as *delay*/*timeout*.

As seen in figure 5, when the delay is smaller than the timeout (relative delay < 1), consensus is reached on average in the same number of rounds, although the time increases linearly. However, the behavior changes drastically once the delay surpasses the timeout, and the number of rounds and the consensus time increase greatly.

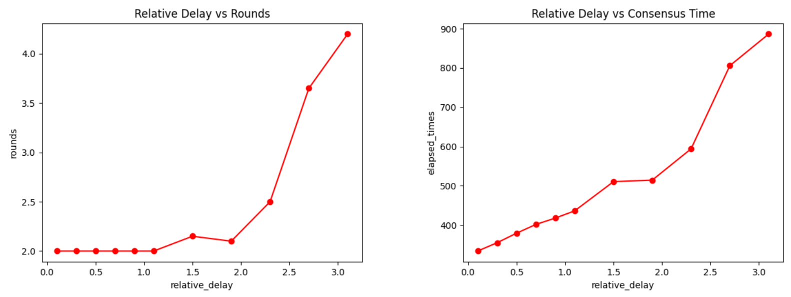


Figure 5: Results for the average number of rounds and the average time to reach consensus when increasing the acceptors random delays.

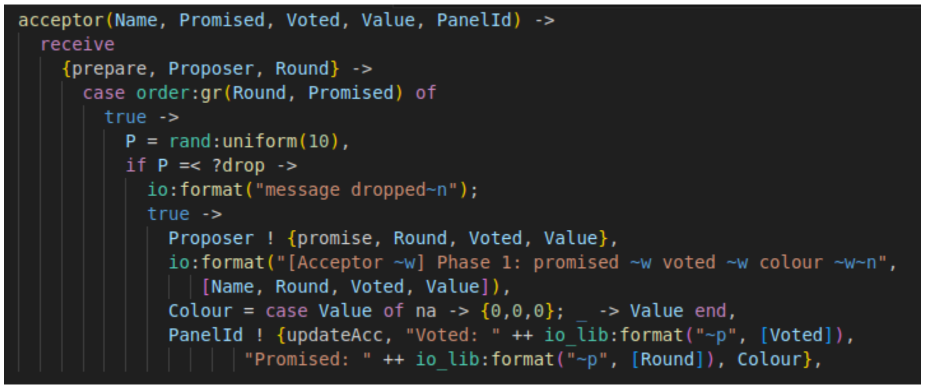
The algorithm is resilient enough to still get consensus, because there is still a probability that the delay is smaller than the timeout for some messages. But if all messages were delayed even more, proposers would be stuck never reaching quorum, and a consensus would never be reached.

2.2 Removing sorry messages

In the next experiment we removed the sorry messages from the acceptors. We noticed that without any further logic for counting the amount of sorries, the sending of sorries does not have any effect on the performance. Even if we get 100 sorries, the proposer will only stop on the first of two conditions: either it gets votes/promises from the majority of acceptors, or it reaches timeout.

2.3 Randomly dropping messages

Next, we introduced a random droppage of messages from the acceptors to simulate failures in the communication. A random number from 0 to 100 is generated to represent the percentage of messages that will be dropped.



```
acceptor(Name, Promised, Voted, Value, PanelId) ->
  receive
    {prepare, Proposer, Round} ->
      case order:gr(Round, Promised) of
        true ->
          P = rand:uniform(10),
          if P =< ?drop ->
            io:format("message dropped~n");
            true ->
              Proposer ! {promise, Round, Voted, Value},
              io:format("[Acceptor ~w] Phase 1: promised ~w voted ~w colour ~w~n",
                  [Name, Round, Voted, Value]),
              Colour = case Value of na -> {0,0,0}; _ -> Value end,
              PanelId ! {updateAcc, "Voted: " ++ io_lib:format("~p", [Voted]),
                        "Promised: " ++ io_lib:format("~p", [Round]), Colour},
```

Figure 6: Code showing the random number used to determine if a message should be dropped.

The results, plotted in figure 15 show that once the drop rate for messages gets around 65% the algorithm can no longer reach consensus with our script's timeout setting. After we have reached this point, our script has a difficulty in recording correct data, because only the lucky runs that finish before the timeout are averaged and plotted. If the paxos algorithm does not reach consensus before timeout, those data points are discarded. This explains the unusual behavior of the data once the drop rate approaches 70%. In reality, if we had an infinite timeout until consensus, we believe it should be reached, even if it takes many more rounds and a lot of time.

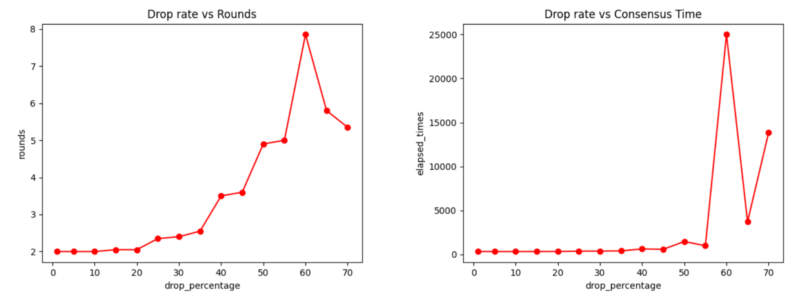


Figure 7: Results for the average number of rounds and the average time to reach consensus when increasing the acceptors percentage of dropped messages.

2.4 Increasing the number of acceptors and proposers.

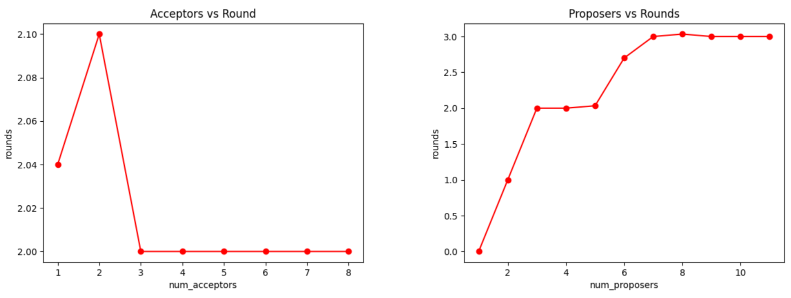


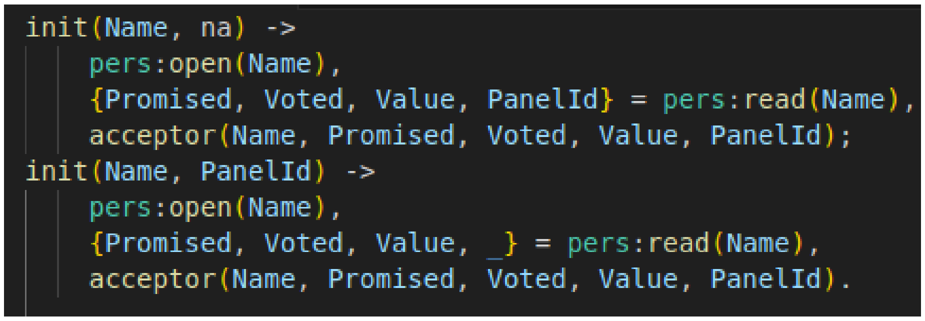
Figure 8: Results for the average number of rounds to reach consensus when increasing the number of acceptors and proposers.

The results in figure 8 show that the number of acceptors does not affect the number of rounds required to reach consensus, as the average was always very close to 2. However, when increasing the number of proposers from one to seven, there is an increase in the number of rounds required to reach consensus. Beyond seven proposers, the results indicate there is no increase in the number of rounds required to reach consensus.

From this we can also conclude that increasing the number of proposers is only detrimental up to a certain point, but after that point there is no real issue with having more.

3 Fault tolerance

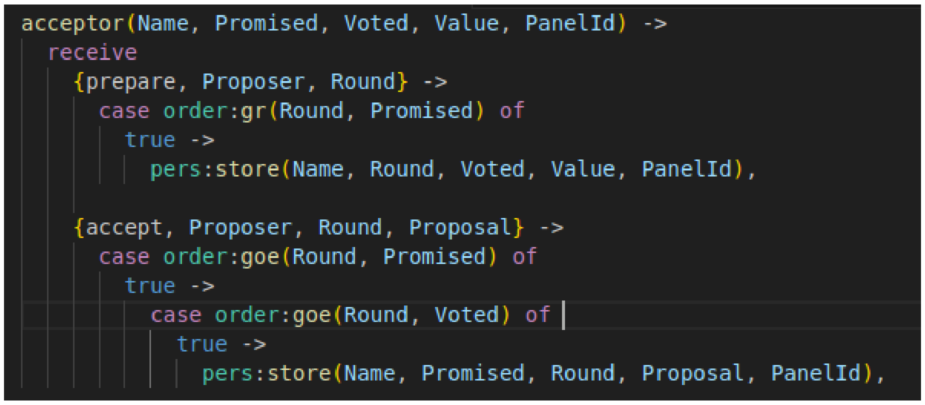
To improve the fault tolerance of paxos, a save state function was implemented for the acceptors. Whenever the acceptors make a new promise or a vote, that information is saved for each acceptor in an individual file. Once an acceptor restarts after a crash, the init function checks if a file containing it's last saved state exists. If a file is found, the stored values are loaded, allowing the acceptor to continue. This ensures that consensus will eventually be reached, even if an acceptor crashes.



```
init(Name, na) ->
    pers:open(Name),
    {Promised, Voted, Value, PanelId} = pers:read(Name),
    acceptor(Name, Promised, Voted, Value, PanelId);
init(Name, PanelId) ->
    pers:open(Name),
    {Promised, Voted, Value, _} = pers:read(Name),
    acceptor(Name, Promised, Voted, Value, PanelId).
```

Figure 9: New init function for acceptor.

Two different init functions are created to handle the cases when PanelId is available and when it is not. If a PanelId is available, it indicates that the acceptor was created during the startup and has not yet crashed. However whenever the PanelId has the value na, it indicates that the init function is being run after a crash and will read from the file to use the stored PanelId value.

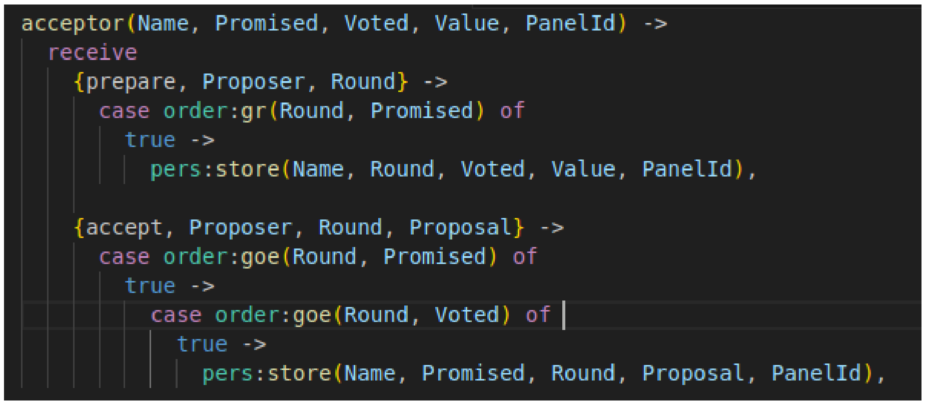


```
acceptor(Name, Promised, Voted, Value, PanelId) ->
  receive
    {prepare, Proposer, Round} ->
      case order:gr(Round, Promised) of
        true ->
          pers:store(Name, Round, Voted, Value, PanelId),

    {accept, Proposer, Round, Proposal} ->
      case order:goe(Round, Promised) of
        true ->
          case order:goe(Round, Voted) of
            true ->
              pers:store(Name, Promised, Round, Proposal, PanelId),
```

Figure 10: Store code for acceptor.

As soon as the acceptor has decided to make a promise or a vote, that updated value will be stored in the acceptor individual file.

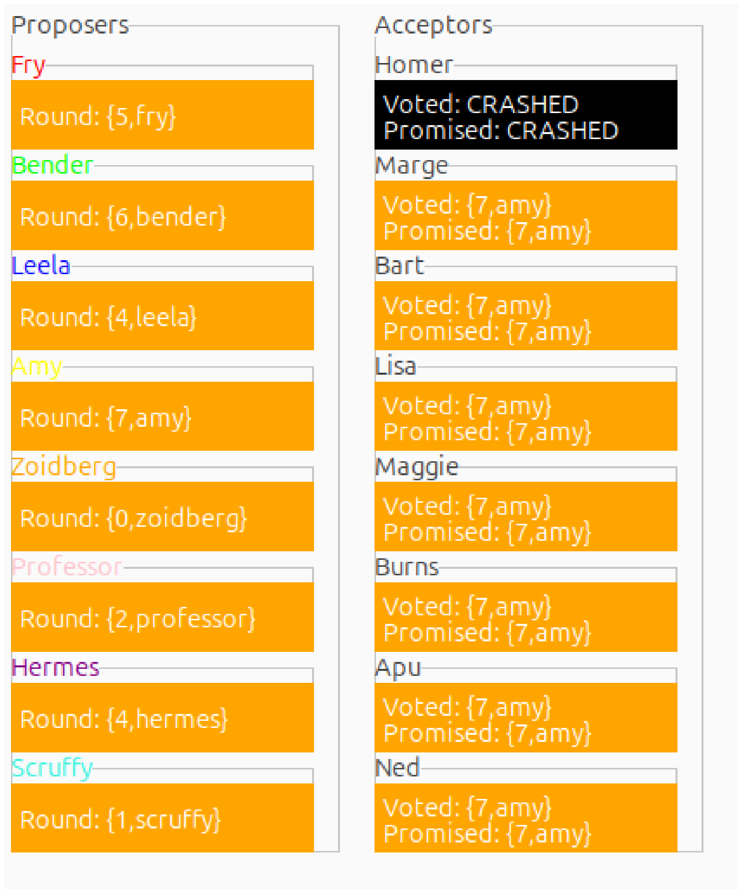


```
acceptor(Name, Promised, Voted, Value, PanelId) ->
  receive
    {prepare, Proposer, Round} ->
      case order:gr(Round, Promised) of
        true ->
          pers:store(Name, Round, Voted, Value, PanelId),

    {accept, Proposer, Round, Proposal} ->
      case order:goe(Round, Promised) of
        true ->
          case order:goe(Round, Voted) of
            true ->
              pers:store(Name, Promised, Round, Proposal, PanelId),
```

Figure 11: Store code for acceptor.

3.1 Experiment



Proposers

Fry
Round: {5,fry}

Bender
Round: {6,bender}

Leela
Round: {4,leela}

Amy
Round: {7,amy}

Zoidberg
Round: {0,zoidberg}

Professor
Round: {2,professor}

Hermes
Round: {4,hermes}

Scruffy
Round: {1,scruffy}

Acceptors

Homer
Voted: CRASHED
Promised: CRASHED

Marge
Voted: {7,amy}
Promised: {7,amy}

Bart
Voted: {7,amy}
Promised: {7,amy}

Lisa
Voted: {7,amy}
Promised: {7,amy}

Maggie
Voted: {7,amy}
Promised: {7,amy}

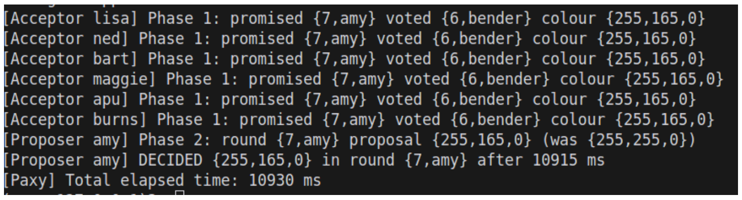
Burns
Voted: {7,amy}
Promised: {7,amy}

Apu
Voted: {7,amy}
Promised: {7,amy}

Ned
Voted: {7,amy}
Promised: {7,amy}

Figure 12: Store code for acceptor.

By initilizig a crash in the Erlang sheel running the acceptors, the GUI will display the selected acceptor as crashed. After the crash sleep it will start working again unless the decision already have been made.



```
[Acceptor lisa] Phase 1: promised {7,amy} voted {6,bender} colour {255,165,0}
[Acceptor ned] Phase 1: promised {7,amy} voted {6,bender} colour {255,165,0}
[Acceptor bart] Phase 1: promised {7,amy} voted {6,bender} colour {255,165,0}
[Acceptor maggie] Phase 1: promised {7,amy} voted {6,bender} colour {255,165,0}
[Acceptor apu] Phase 1: promised {7,amy} voted {6,bender} colour {255,165,0}
[Acceptor burns] Phase 1: promised {7,amy} voted {6,bender} colour {255,165,0}
[Proposer amy] Phase 2: round {7,amy} proposal {255,165,0} (was {255,255,0})
[Proposer amy] DECIDED {255,165,0} in round {7,amy} after 10915 ms
[Paxy] Total elapsed time: 10930 ms
```

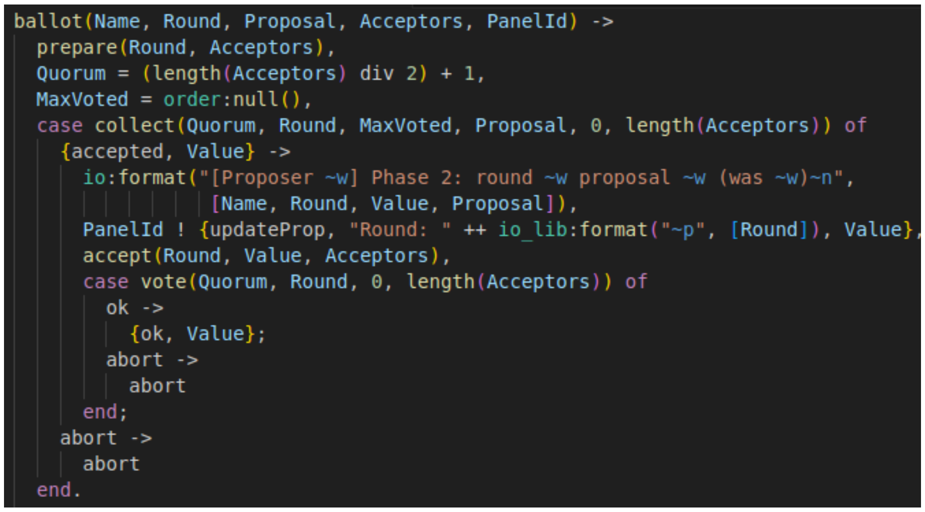
Figure 13: Store code for acceptor.

However since a majority can still be reached without the presenc of one of the acceptor a decision can still be reached seen in the 13

4 Improving by using sorry messages

The previous code for the proposer did not make use of the received sorry messages to reach a consensus faster.

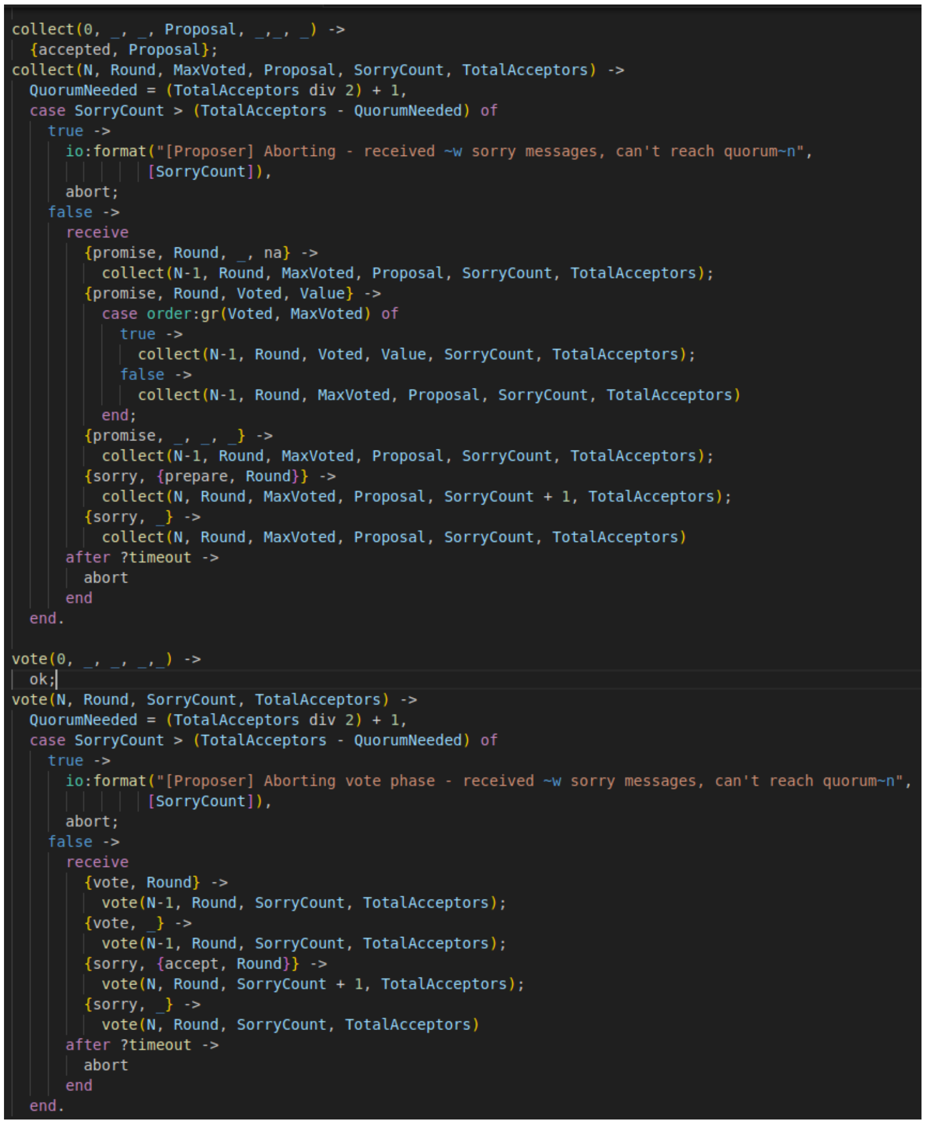
An improvement can be made by calculating the required quorum and count the number of sorry messages to reach consensus faster. Once enough sorries have been received that make it imposible to reach quorum, the phase is aborted.



```erlang
ballot(Name, Round, Proposal, Acceptors, PanelId) ->
  prepare(Round, Acceptors),
  Quorum = (length(Acceptors) div 2) + 1,
  MaxVoted = order:null(),
  case collect(Quorum, Round, MaxVoted, Proposal, 0, length(Acceptors)) of
    {accepted, Value} ->
      io:format("[Proposer ~w] Phase 2: round ~w proposal ~w (was ~w)~n",
                [Name, Round, Value, Proposal]),
      PanelId ! {updateProp, "Round: " ++ io_lib:format("~p", [Round]), Value},
      accept(Round, Value, Acceptors),
      case vote(Quorum, Round, 0, length(Acceptors)) of
        ok ->
          {ok, Value};
        abort ->
          abort
      end;
    abort ->
      abort
  end.
```

Figure 14: Proposer ballot improvement.

Modifications are made in the proposer's ballot. Both the first collect and vote call include the lenght of the acceptors. A zero is also included to be used as the recursive count variable.



```
collect(0, _, _, Proposal, _,_, _) ->
  {accepted, Proposal};
collect(N, Round, MaxVoted, Proposal, SorryCount, TotalAcceptors) ->
  QuorumNeeded = (TotalAcceptors div 2) + 1,
  case SorryCount > (TotalAcceptors - QuorumNeeded) of
    true ->
      io:format("[Proposer] Aborting - received ~w sorry messages, can't reach quorum~n",
                [SorryCount]),
      abort;
    false ->
      receive
        {promise, Round, _, na} ->
          collect(N-1, Round, MaxVoted, Proposal, SorryCount, TotalAcceptors);
        {promise, Round, Voted, Value} ->
          case order:gr(Voted, MaxVoted) of
            true ->
              collect(N-1, Round, Voted, Value, SorryCount, TotalAcceptors);
            false ->
              collect(N-1, Round, MaxVoted, Proposal, SorryCount, TotalAcceptors)
          end;
        {promise, _, _, _} ->
          collect(N-1, Round, MaxVoted, Proposal, SorryCount, TotalAcceptors);
        {sorry, {prepare, Round}} ->
          collect(N, Round, MaxVoted, Proposal, SorryCount + 1, TotalAcceptors);
        {sorry, _} ->
          collect(N, Round, MaxVoted, Proposal, SorryCount, TotalAcceptors)
      after ?timeout ->
        abort
      end
  end.

vote(0, _, _, _,_) ->
  ok;
vote(N, Round, SorryCount, TotalAcceptors) ->
  QuorumNeeded = (TotalAcceptors div 2) + 1,
  case SorryCount > (TotalAcceptors - QuorumNeeded) of
    true ->
      io:format("[Proposer] Aborting vote phase - received ~w sorry messages, can't reach quorum~n",
                [SorryCount]),
      abort;
    false ->
      receive
        {vote, Round} ->
          vote(N-1, Round, SorryCount, TotalAcceptors);
        {vote, _} ->
          vote(N-1, Round, SorryCount, TotalAcceptors);
        {sorry, {accept, Round}} ->
          vote(N, Round, SorryCount + 1, TotalAcceptors);
        {sorry, _} ->
          vote(N, Round, SorryCount, TotalAcceptors)
      after ?timeout ->
        abort
      end
  end.
```

Figure 15: Proposer ballot improvement.

Whenever a sorry message is received for the collect and vote function, the next recursive call will increase the sorry count by one, if its the correct round.

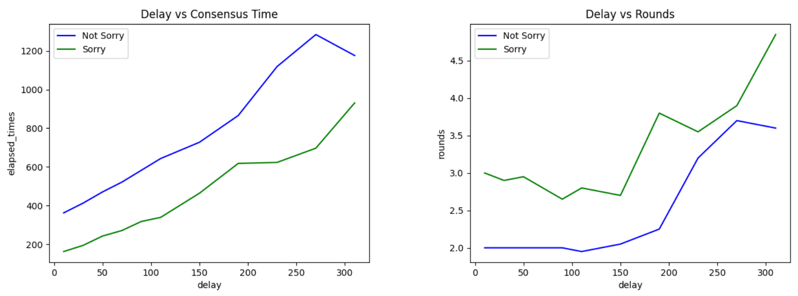


Figure 16: Results for the average time to reach consensus and the average number of rounds comparing when proposers count the number of sorries (shown in green) and when they don't (shown in blue).

The results obtained from our experiment in figure 16 show a decrease in the time needed to reach consensus when the sorry messages are counted. We also see that when sorry messages are counted, there is an increase in the required number rounds to reach consensus. This is likely because on average the proposers take less time to determine they will not reach quorum, and that they have to start a new round. This also means that on average the time spent in each round is shorter, whereas before it was always similar to the proposers timeout configuration.

5 Analysis summary

A Principal Component Analysis (PCA) was conducted using 250 different parameter combinations to examine the relationships between inputs and outputs in the Paxos algorithm. These combinations were carefully bounded within specified ranges and selected randomly using a uniform probability distribution. To ensure the results were unbiased and statistically significant, each combination was executed five times using a Python script that automated the process.

The data collected, including input parameters and outputs such as the number of rounds, total elapsed time, and the number of timeouts, was used to perform the PCA. This analysis aimed to identify correlations, dependencies, and the relative influence of different parameters on the outcomes.

5.1 PCA Results and Key Insights

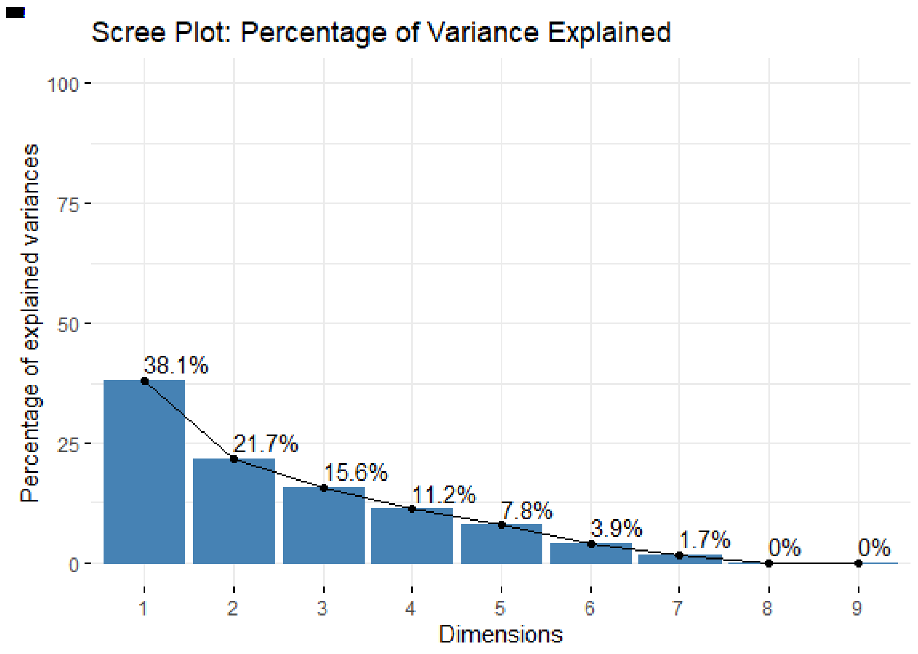


Figure 17: Scree plot of the PCA : Used to select the number of dimension to keep for the study

- **Dimensionality and Variance Distribution:** The total inertia analysis revealed that the first two principal components captured 59.8% of the variance, indicating that a significant portion of the data's structure could be explained by these two dimensions. However, an elbow was observed in the variance plot after the third dimension, suggesting diminishing returns for higher dimensions. Consequently, we focused on the first three dimensions, as these offered the most meaningful insights into parameter relationships:

  Dimension 1

  Contrasts individuals with high *timeout_count*, *rounds*, *num_proposers*, *elaps- ed_times*, *total_times*, and *drop_percentage* against those with low values for these variables. Relative delay and delay are distinct for a separate group with low values in other variables.

  Dimension 2

  Opposes individuals with high *relative_delay* and *delay* against those with low values for these variables. These two variables strongly correlate with the dimension.

Dimension 3

Contrasts individuals with high *drop_percentage* and *timeout_count* but low rounds and *num_proposers* against those with high *num_acceptors* and *num_pro- posers* but low *timeout_count* and *drop_percentage*. Another group shows high rounds with low *drop_percentage*, and *timeout_count*.

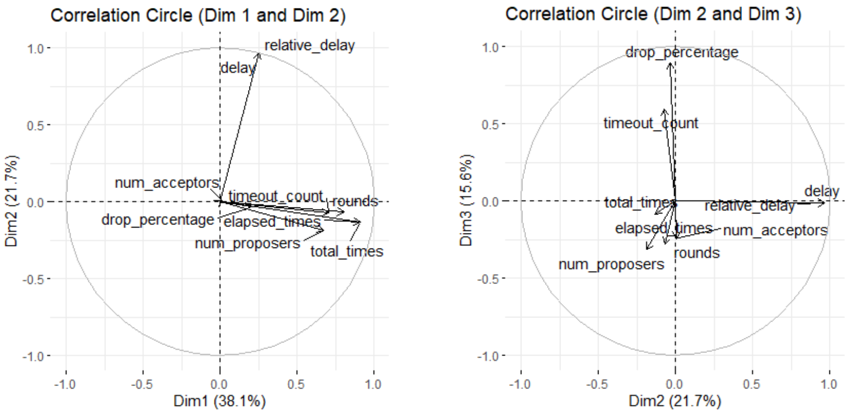


Figure 18: Representation of PCA axes showing the relationships between variables and their contributions to the principal components. The length and direction of the arrows indicate the strength and nature of the correlations.

- **Major Correlations (Dimensions 1 and 2):** The PCA revealed a strong correlation between the number of proposers, the number of rounds, and the total elapsed time. This indicates that the parameter with the most significant impact on the time required to reach consensus is the number of proposers. Interestingly, beyond a certain threshold, increasing the number of proposers no longer contributed to delays, suggesting an optimal upper limit for scalability.

- **Impact of the Drop Rate (Dimensions 2 and 3):** A positive correlation between the drop rate and the number of timeouts was observed, confirming that the drop rate is the most critical factor impeding consensus. This finding underscores the importance of minimizing communication failures to ensure the robustness of the Paxos algorithm in distributed systems.

- **Marginal Parameters:** Parameters such as the number of acceptors showed minimal impact on the outcomes, indicating that their influence is negligible in typical configurations. Similarly, the delay parameter was highly correlated with relative delay but showed little direct effect on consensus, except in extreme conditions where it surpassed timeout thresholds.

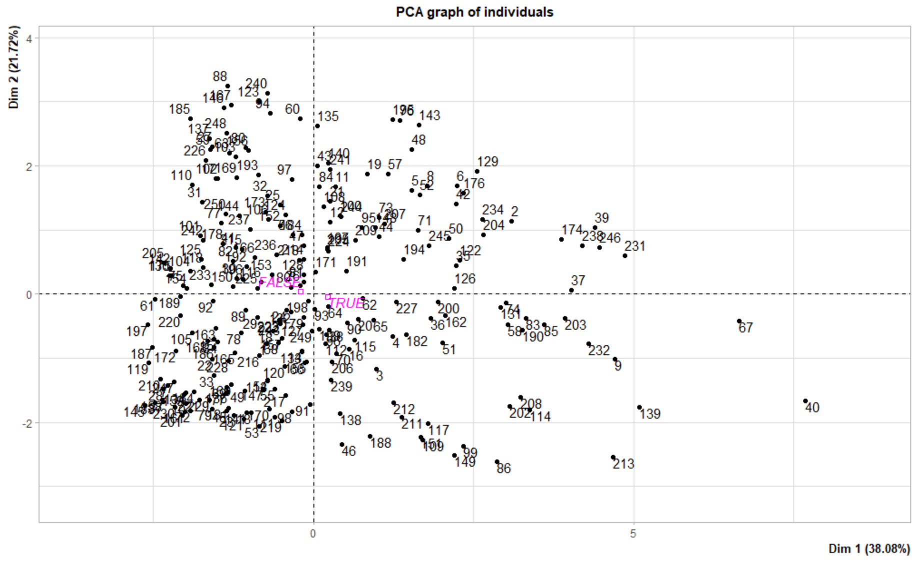


Figure 19: The scatterplot (cloud of points) represents the projection of the dataset onto the selected principal components. Each point corresponds to an observation, and its position indicates how it relates to the principal axes.

- **"Sorry" Logic Observations:** While the PCA highlighted a weak influence of the "Sorry" logic on the total program duration, it was insufficient to draw robust conclusions. This suggests that additional experimentation or algorithmic modifications, such as leveraging "Sorry" messages more effectively, could further enhance efficiency.

5.2 Broader Implications and Future Directions

This analysis offers practical insights into optimizing the Paxos algorithm for real-world applications. The identification of key parameters, such as the number of proposers and the drop rate, highlights areas where system architects can focus their optimization efforts. By controlling these variables, it is possible to improve the efficiency and fault tolerance of consensus processes in distributed systems.

However, this study also raises important questions and opens avenues for further exploration:

- **Scalability and Configuration:** How does the algorithm behave under larger-scale configurations with hundreds of proposers and acceptors? Would the relationships observed here persist, or would new patterns emerge?

- **Dynamic Parameter Tuning:** Can adaptive mechanisms be developed to dynamically adjust parameters like proposer timeouts or drop rates based on real-time system feedback?
- **Alternative Consensus Mechanisms:** How do these findings compare to other consensus algorithms like Raft or Byzantine Fault Tolerance (BFT)? Are there lessons that can be cross-applied to improve Paxos further?
- **Enhanced Use of "Sorry" Messages:** What new strategies can be implemented to make better use of "Sorry" messages to accelerate consensus or reduce unnecessary rounds?

5.3 Conclusion

In conclusion, this study provided a comprehensive analysis of the Paxos algorithm, revealing the dominant role of the number of proposers and the drop rate in influencing consensus efficiency. While other parameters like the number of acceptors and absolute delay showed negligible impact, the findings underline the need for robust communication channels and judicious parameter selection.

These results not only validate the resilience of Paxos under varied conditions but also serve as a foundation for enhancing its performance. By addressing the questions raised and exploring the suggested future directions, we can continue to refine Paxos and its applications in distributed systems, ensuring its relevance in an era of increasing computational demands and complexity.