# Seminar Report: PAXY Seminar

Shiva Toutounchiavval, Fernando Morales, Cristián Ramón-Cortés

May 23, 2015


UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

# Contents

**1 Introduction** **1**

**2 Work Done** **1**

**3 Experiments** **2**
3.1 Initial sleep time . . . . . 2
3.2 Introducing Delays in the Acceptor . . . . . 3
3.3 Avoid Sending Sorry Messages . . . . . 4
3.4 Randomly Dropping Messages in the Acceptor . . . . . 5
3.5 Increase the Number of Acceptors and Proposers . . . . . 8
3.6 Split the PAXY Module . . . . . 8
3.7 Fault Tolerance . . . . . 9
3.8 Improvement Based on Sorry Message . . . . . 10

**4 Personal Opinion** **11**

# List of Figures

1 Execution of initial sleep experiment - Default values . . . . . 2
2 Execution of initial sleep experiment - Incremental values . . 3
3 Execution with delay on both prepare and accept messages . 4
4 Execution avoiding sorry messages . . . . . 5
5 Execution dropping promise messages . . . . . 5
6 Execution dropping accept messages . . . . . 6
7 Execution dropping both promise and accept messages . . . 6
8 Execution dropping 1/5 messages . . . . . 7
9 Execution dropping 1/2 messages . . . . . 7
10 Execution of split PAXY . . . . . 9
11 Execution of fault tolerance without restart . . . . . 10
12 Execution of fault tolerance with restart . . . . . 10
13 Execution of fault tolerance with restart . . . . . 11

# List of Tables

1 Results delaying prepare messages . . . . . 3
2 Results delaying accept messages . . . . . 3
3 Results delaying prepare messages . . . . . 4
4 Results avoiding sorry messages . . . . . 4
5 Results dropping messages in the Acceptor . . . . . 6
6 Results increasing the number of dropped messages . . . . . 7
7 Results increasing the number of acceptors . . . . . 8
8 Results increasing the number of proposers . . . . . 8

9 Results increasing the number of both acceptors and proposers 8

# 1 Introduction

The seminar is mainly about the Paxos algorithm which is used to gain consensus in a distributed system. Paxos is a flexible and fault tolerant protocol for solving the consensus problem, where participants in a distributed system need to agree on a common value. For the algorithm correctness it doesn't matter what this value is, but it must ensure that a single one among the proposed values is ever chosen.

The Paxos algorithm has three different processes: proposers, acceptors and learners although all three are often included in one single process. This seminar pretends to provide an implementation for the proposer and acceptor processes but excludes the learner process since it is not needed to reach a consensus.

The main topic related to distributed systems that this seminar covers is about how to gain consensus in a distributed system under the condition of perfect communication and faulty processes.

# 2 Work Done

In this assignment we have implemented the PAXOS protocol in Erlang in order to see some basic functionality. The simulation scenario consisted of a system with three *Proposers* and five *Acceptors*. This dimensions are far away from real distributed systems, but in turn it allowed us to understand easily the key steps of PAXOS. A graphical user interface combined with a terminal provided all the necessary information to check each step of the simulations.

First, we completed the provided source code (*Paxy*) following the theory explained in class. There were some implementation-related issues not covered in theory that raised our interest. For example, it was interesting to see in the code how *Proposers* only consider messages from their current rounds, ignoring the rest of them in their mailboxes even they were sent by them. This gives an idea about the asynchronous nature of this problem.

After completing the code, we added some delay to the acceptors replies in order to simulate latency in the network, approaching this way more realistic conditions. In addition to this, we inserted message dropping in order to simulate an unreliable transport channel. This will be explained more in detail later.

This assignment also includes some improvements of the original version. We removed the sorry messages to understand how relevant are they inside the PAXOS protocol. We also added fault tolerancy to our processes so they could store their information in a persistent storage device and recover it in case of a crash. We also separated the PAXY implementation to run the acceptors and the proposers on different machines. Finally, we implemented

an improvement based on the proper interpretation of sorry messages in order to finish PAXOS earlier. This will be discussed in detail along the assignment.

# 3 Experiments

**Experiment Strategy:** Each individual experiment is done for 10 times and average round number for all proposers is calculated every time. Finally the average round number for the whole experiment in all 10 experiments is calculated.

## 3.1 Initial sleep time

In this experiment we modify the initial sleep time. The code used for this experiment is located at the *src/main* folder. We first run the original code launching the proposers at the same time: *paxy:start([200,200,200])*. Then we run again the code delaying some of the proposers: *paxy:start([1000,2000,3000])*.

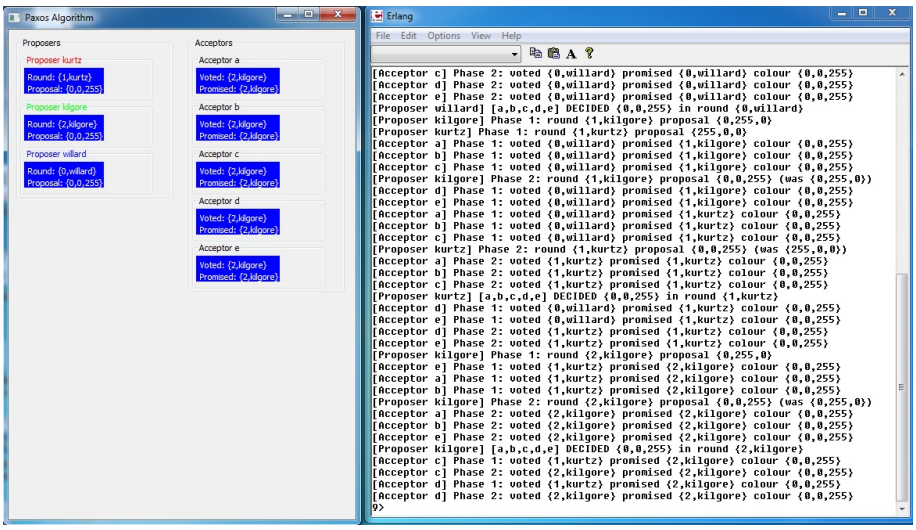


Figure 1: Execution of initial sleep experiment - Default values

As shown in Figures 1 and 2, wWe find that it takes more rounds to reach consensus when proposers run at the same time than when they are delayed.

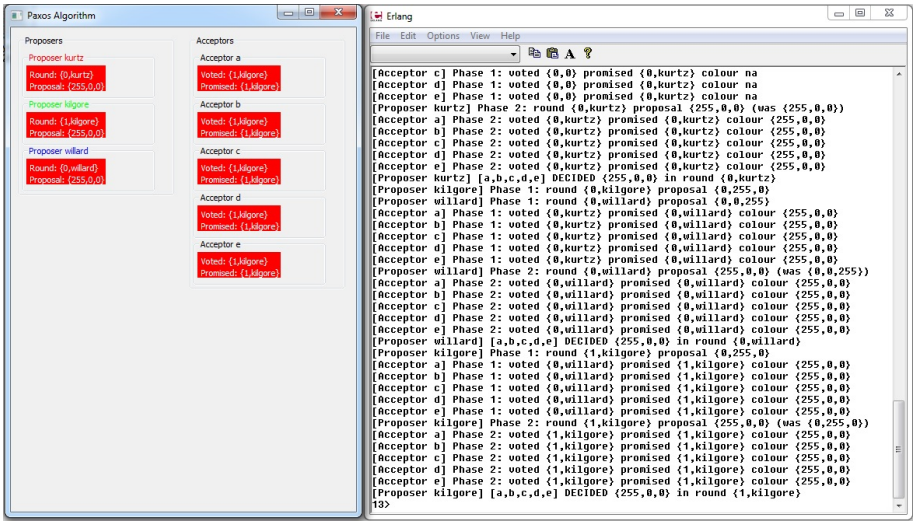


Figure 2: Execution of initial sleep experiment - Incremental values

### 3.2 Introducing Delays in the Acceptor

In this experiment we delay the prepare messages from the acceptors before sending them to the proposers and we check if the algorithm terminates or not. For this case the used code is located under *src/delay* folder.

We do three kinds of experiments with 3 proposers and 5 acceptors:

1. Delays before the prepare messages

| | Proposer A | Proposer B | Proposer C | Average |
|---|---|---|---|---|
| Number of Rounds | 1.1 | 2.6 | 1.9 | 1.87 |

Table 1: Results delaying prepare messages

2. Delays before the accept messages

| | Proposer A | Proposer B | Proposer C | Average |
|---|---|---|---|---|
| Number of Rounds | 1.1 | 3.1 | 2.2 | 2.13 |

Table 2: Results delaying accept messages

3. Delays before the prepare and accept messages

| | Proposer A | Proposer B | Proposer C | Average |
|---|---|---|---|---|
| Number of Rounds | 2 | 2.6 | 2.6 | 2.4 |

Table 3: Results delaying prepare messages

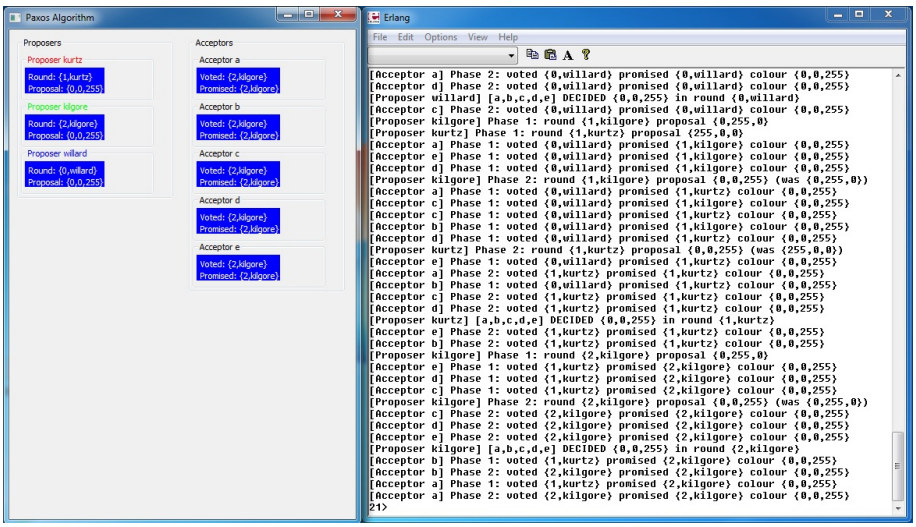


Figure 3: Execution with delay on both prepare and accept messages

As shown in Table 1, Table 2 and Table 3, the more delays there are, the more rounds it takes to reach consensus.

### 3.3 Avoid Sending Sorry Messages

We do three kinds of experiments with 3 proposers and 5 acceptors avoiding to send sorry messages in different phases:

- Not sending in the 1st phase
- Not sending in the 2nd phase
- Not sending in both phases

| | Always Send | No send 1 | No send 2 | No send 1-2 |
|---|---|---|---|---|
| Number of Rounds | 1 | 1 | 1 | 1 |

Table 4: Results avoiding sorry messages

As we can see in Table 4, not sending sorry messages has no effect on the number of rounds to reach consensus. Moreover, *Proposers* always reach consensus even if the sorry messages are avoided or not.

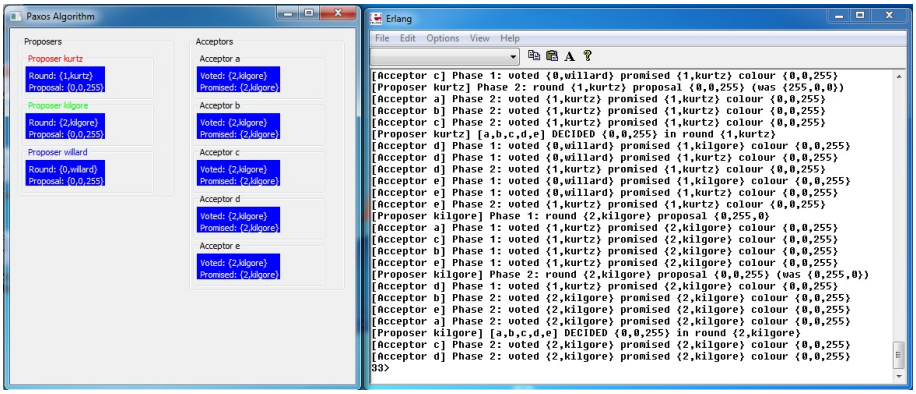


Figure 4: Execution avoiding sorry messages

### 3.4 Randomly Dropping Messages in the Acceptor

In this section we try to evaluate the impact of dropping messages from the acceptor. For this purpose the code located under the *src/drop* folder is used. A first set of experiments is done to check if dropping promise or vote messages have a different impact on the execution:
– Experiment 1: Only drop 1/10 promise messages

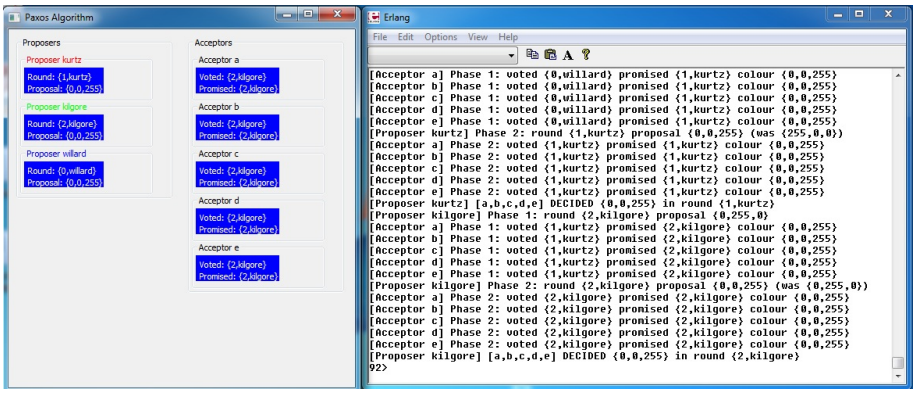


Figure 5: Execution dropping promise messages

– Experiment 2: Only drop 1/10 vote messages

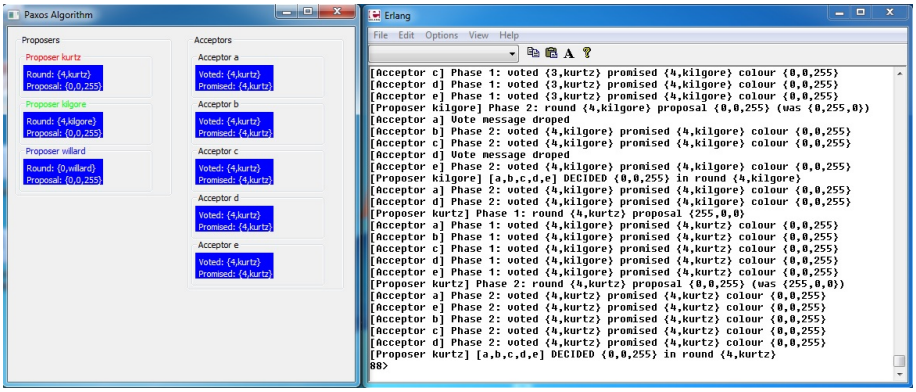


Figure 6: Execution dropping accept messages

– Experiment 3: Drop both 1/10 promise and 1/10 vote messages

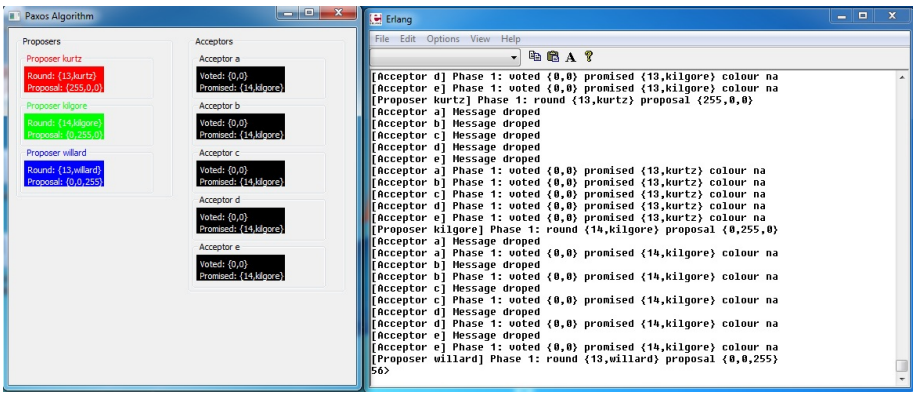


Figure 7: Execution dropping both promise and accept messages

| | No drop | 1/10 Promise | 1/10 Vote | 1/10 both |
|---|---|---|---|---|
| Number of Rounds | 1 | 1.1 | 1.13 | 2.3 |

Table 5: Results dropping messages in the Acceptor

As shown in Table 5 dropping message always leads the algorithm to arrive to a consensus. However, while promise and vote messages separately have a little and similar effect on its execution, dropping both messages has a big effect on the algorithm performance.

In a second set of experiments we increase the percentage of dropped messages:

– drop 1/5 promise and vote messages

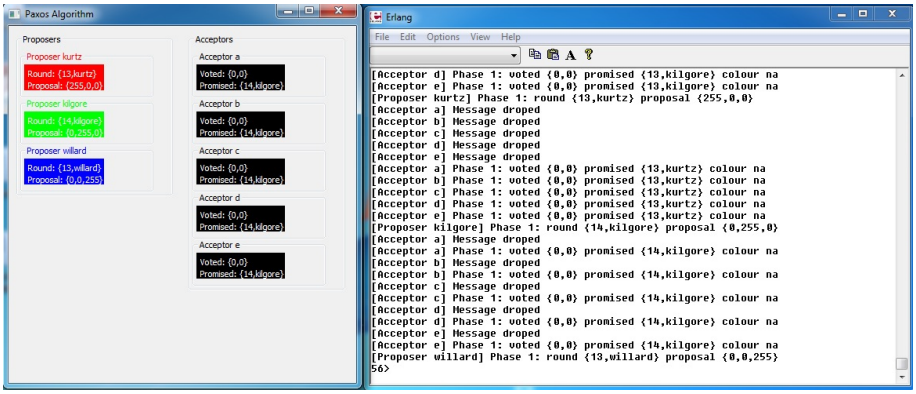


Figure 8: Execution dropping 1/5 messages

– drop 1/2 promise and vote messages

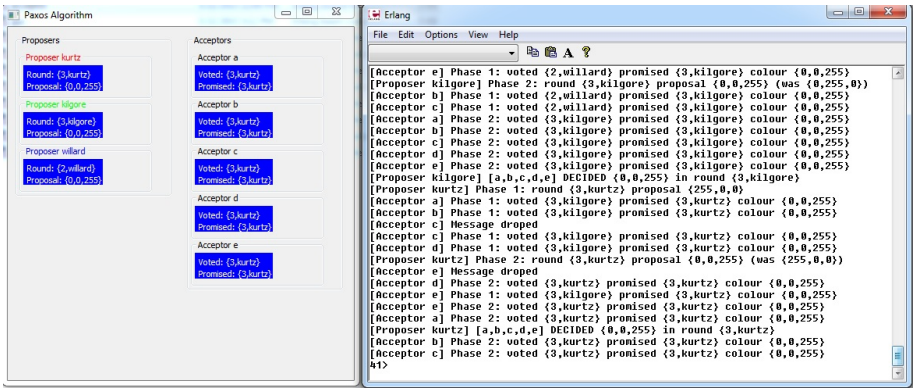


Figure 9: Execution dropping 1/2 messages

– drop all promise and vote messages

| | No drop | 1/5 both | 1/2 Both | 1 both |
|---|---|---|---|---|
| Number of Rounds | 1 | 1.23 | 6.47 | never |

Table 6: Results increasing the number of dropped messages

As shown in Table 6, the effect of dropping a percentage of both promise and vote messages is exponentially increasing. Moreover, it is possible that the algorithm reports conflicting answers when a majority of accepters have voted for a proposal and some of the vote messages are dropped. The proposer should have reach consensus if no messages are droped. But in the case of dropping messages, the proposer will continue.

### 3.5 Increase the Number of Acceptors and Proposers

In this experiments we increase the number of proposers and acceptors using the basic code located under *src/main*. Next, the experiment results are shown:
• Only increase the number of acceptors 1,2,3, which means there are 3 proposers and 6,7,8 acceptors.

| | [3,6] | [3,7] | [3,8] |
|---|---|---|---|
| Number of Rounds | 1 | 1 | 1 |

Table 7: Results increasing the number of acceptors

• Only increase the number of proposers 1,2,3, which means there are 4,5,6 proposers and 5 acceptors.

| | [4,5] | [5,5] | [6,5] |
|---|---|---|---|
| Number of Rounds | 1.5 | 1.68 | 0 |

Table 8: Results increasing the number of proposers

• Increase the number of both acceptors and proposers 1,1,2,2,3,3 which means there are 4 proposers 6 acceptors, 5 proposers 7 acceptors and 6 proposers 8 acceptors.

| | [4,6] | [5,7] | [6,8] |
|---|---|---|---|
| Number of Rounds | 1.5 | 1.92 | 2.38 |

Table 9: Results increasing the number of both acceptors and proposers

As is shown in Table 7, Table 8 and Table 9, the number of acceptors has no effect on the number of rounds it takes for proposers to reach consensus. However, the number of proposers has a big effect, and the larger the number is, the more rounds it will need to reach consensus.

### 3.6 Split the PAXY Module

The goal of this set of experiments is to run the *Acceptors* and the *Proposers* on different machines. For this purpose the code saved under the *src/splitPaxy* folder have been developped. The gui.erl has been modified to allow separated gui's for *Acceptors* and *Proposers* and the paxy.erl module has been modified to start separately *Acceptors* and *Proposers*.

When starting Erlang, we make it network aware by providing a name:
• erl -sname acceptors -setcookie 123456

• erl -sname proposers -setcookie 123456

Then, Erlang processes can communicate with each other in different machines by using the following command:
{*ACCEPTOR_ID, acceptors@NODE_NAME*} *! message*

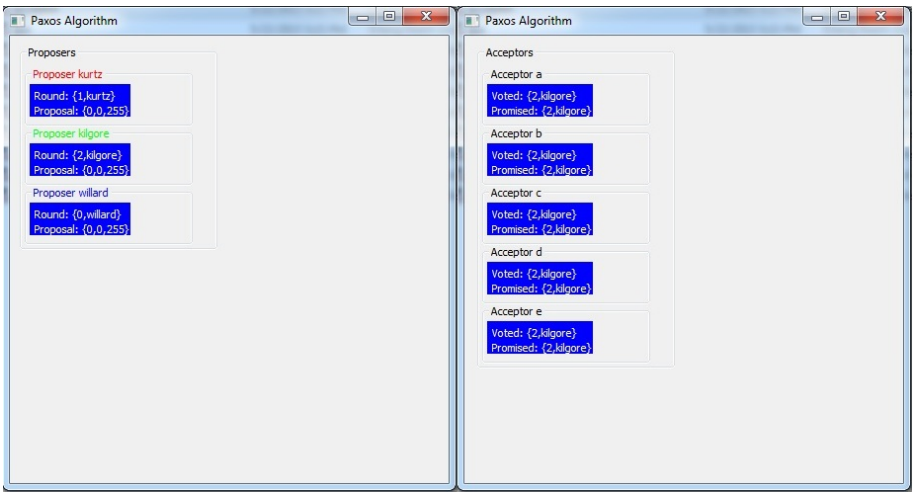


Figure 10: Execution of split PAXY

Figure 10 shows the execution after splitting the PAXY module.

### 3.7 Fault Tolerance

For this experiment we have added the pers module to the acceptor so it can save and restore its state. Basically we have added the pers:open(Name) and pers:read(Name) method calls in *Acceptor* init(Name,PaneId) and the pers:store(...) every time that the *Acceptor* updates its state. The code used for these experiments can be found under the *src/faultTolerance* folder. In order to make sure the acceptor really recovers, we do two experiments:
• Not restart the acceptor after exiting
• Restart the acceptor after exiting

A shown in Figure 11, when *Acceptor 1* is not restarted after exiting, it remains crashed and never recovers. However, as shown in Figure 12 when the acceptor is restarted after exiting, it recovers and the algorithm can reach a consensus. Therefore, we can conclude that our fault tolerance version works well after crashing.

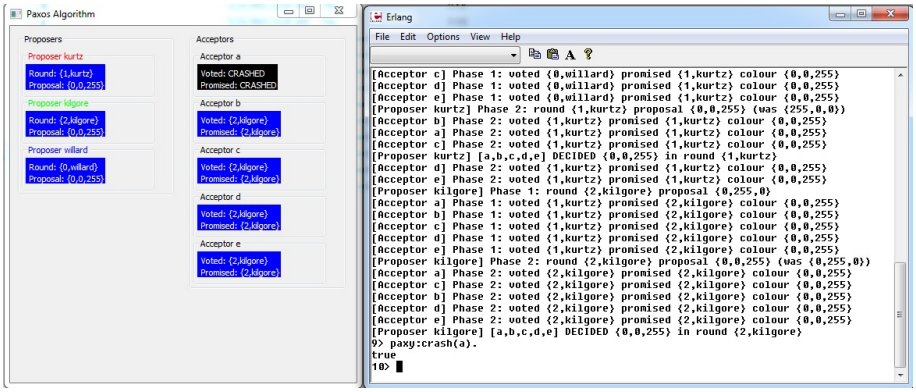


Figure 11: Execution of fault tolerance without restart

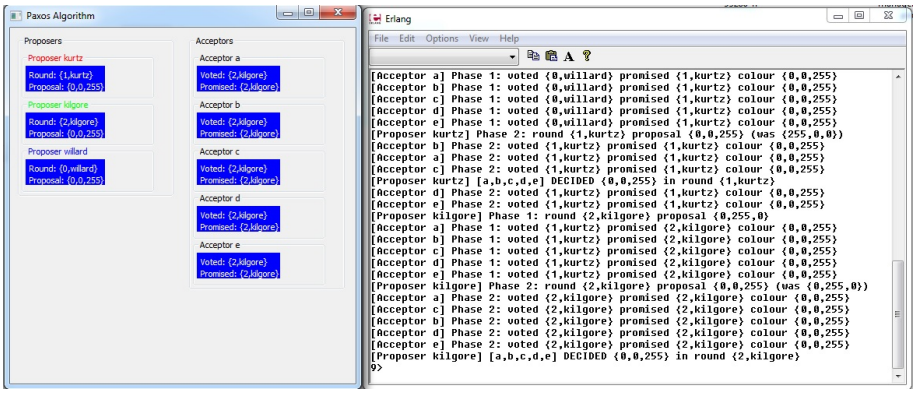


Figure 12: Execution of fault tolerance with restart

## 3.8 Improvement Based on Sorry Message

After receiving sorry messages from a majority of acceptors, it means the proposer will never receive promise messages or vote messages from a majority of acceptors. Therefore, the ballot will abort in order to get better efficiency. For this experiments we modify the collect and vote methods in the *proposer* code adding a parameter to count the recieved sorry messages. The code can be found under the *src/sorryImprovement* folder.

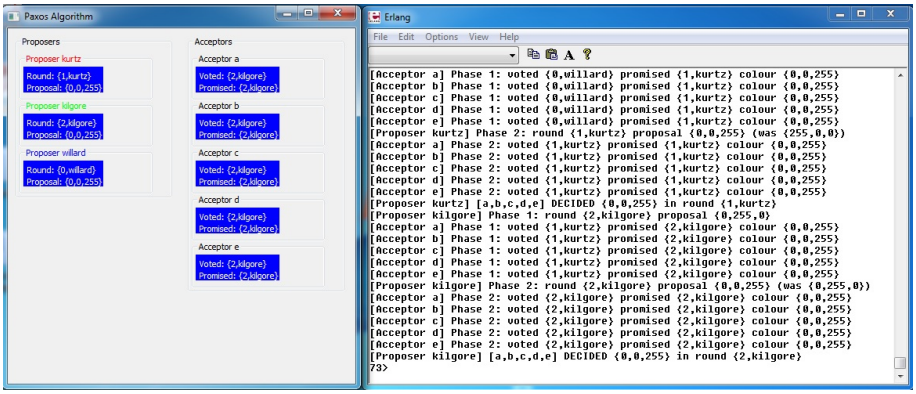


Figure 13: Execution of fault tolerance with restart

After having executed this new version we can check that the ballot is aborted earlier and thus, it achieves the next round more efficiently.

# 4 Personal Opinion

From our point of view, this seminar was great to understand the basic functionality of an asynchronous consensus algorithm, as PAXOS is. We were able to learn some more about Erlang and its graphical capabilites that resulted very useful in order to follow the protocol life cycle. We have experienced how comfortable is Erlang when working with processes.

This assignment was done in a very simple scenario with 8 processes running. This helped us understanding the algorithm on a first phase. However, once the key points were covered, this scenario became trivial and we lacked of maybe a second phase with a deeper analysis. For instance, seeing real-world implementations and how PAXOS is implemented in other programming languages, some use-cases and real-world scenarios would have resulted in a very attractive learning.