Seminar Report: Paxy

Maria Gabriela Valdes and Victoria Beleuta

December 9, 2014

1 Introduction

This assignment gave us the opportunity to learn and implement the *Paxos* algorithm. This algorithm is a protocol for solving consensus between processes in a distributed system. Consensus consists in agreeing on a proposed value among a group of participants. In this particular case, we have a set of *Proposers* that propose different values, to a set of *Acceptors* until a single and unique value is agreed between all *Proposers* and *Acceptors*.

2 Work done

We have completed the code files provided to correctly implement the algorithm. Our source code consists of two folders: *src* and *src-sep*. The first one contains the *Erlang* files implementing the *Paxos* algorithm with both groups of proposers and acceptors starting in the same *Erlang* environment. In the folder *src-sep* we have modules to give the possibility to start the group of proposers and the acceptors in two different *Erlang* nodes.

To start the algorithm locally you have to call the function *start* of module *paxy* with only one argument, a list of int values of the same size of the number of proposers used. Each value represents the initial sleep time of each proposer (in milliseconds). For example:

paxy:start([2000,2000,2000]).

With this command the *Paxos* algorithm will start with three proposers, each one with 2 seconds of initial sleep, and five acceptors. Both group of proposers and acceptors will start in the same *Erlang* node.

To start the algorithm with proposers and acceptors both in different *Erlang* nodes you first have to start two different *Erlang* environments. For example:

- In one terminal type *erl -sname proposers*. This will represent an *Erlang* node name *proposers@server*.
- In another different terminal type *erl -sname acceptors*. This will represent a *Erlang* node name *acceptors@server*.

To start the proposers you have to call the function *start_sep_proposers* of module *paxy_sep* with two arguments, a list of int values of the same size of the number of proposers used, the same as explained before, and the name of the acceptor's node, in the previous example named *acceptors@server*. This command has to be called from the proposers's node. For example:

paxy_sep:start_sep_proposers([2000,2000,2000], acceptors@server).

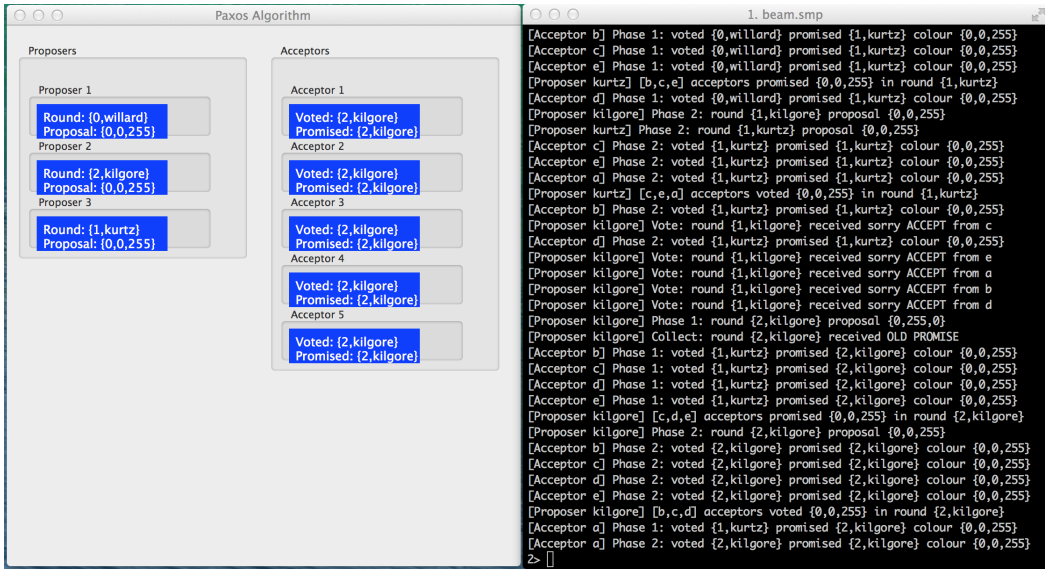
In a similar way to start the acceptors you have to call the function *start_sep_acceptors* of module *paxy_sep* with one argument, the name of the proposer's node, in the previous example named *proposers@server*. This command has to be called from the acceptors's node. For example:

paxy_sep:start_sep_acceptors(proposers@server).

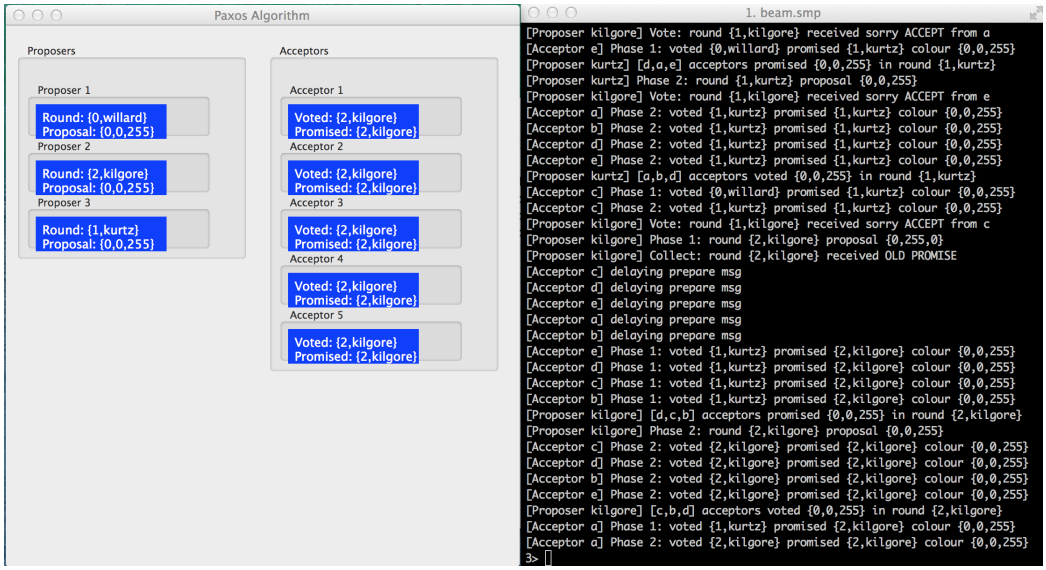
With this command the *Paxos* algorithm will start with 3 proposers, each one with 2 seconds of initial sleep, and 5 acceptors. Both group of proposers and acceptors will start in different *Erlang* nodes.

3 Experiments

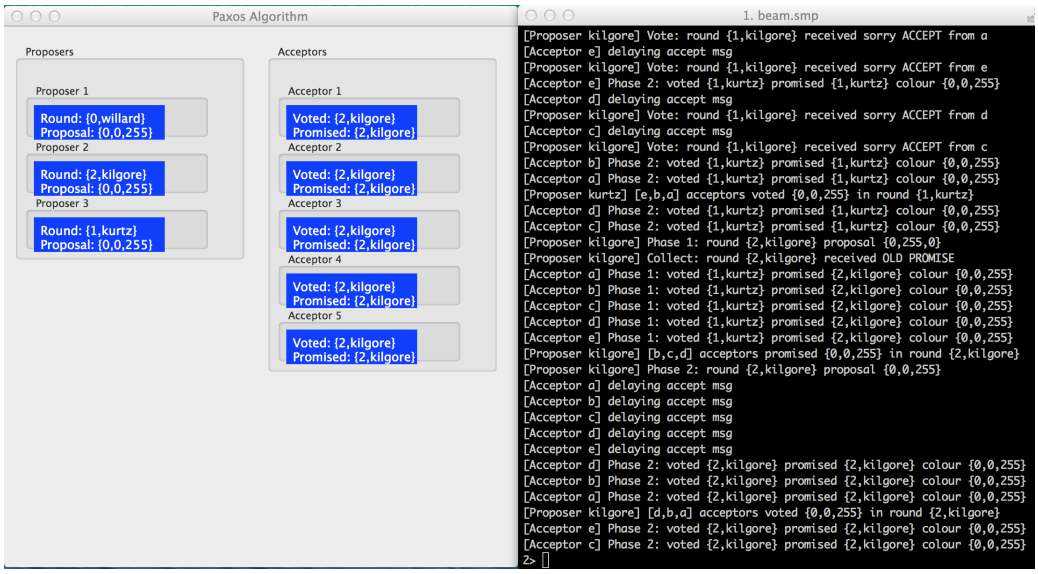
First we run an experiment in the same machine with 3 proposers and 5 acceptors. After compiling the code files located in *src* and running the paxy module *paxy:start([200,200,200])*. Proposer 0, *willard* proposes color *blue*, and in round 2, the acceptors voted for *kilgore* and decided on color *blue*.



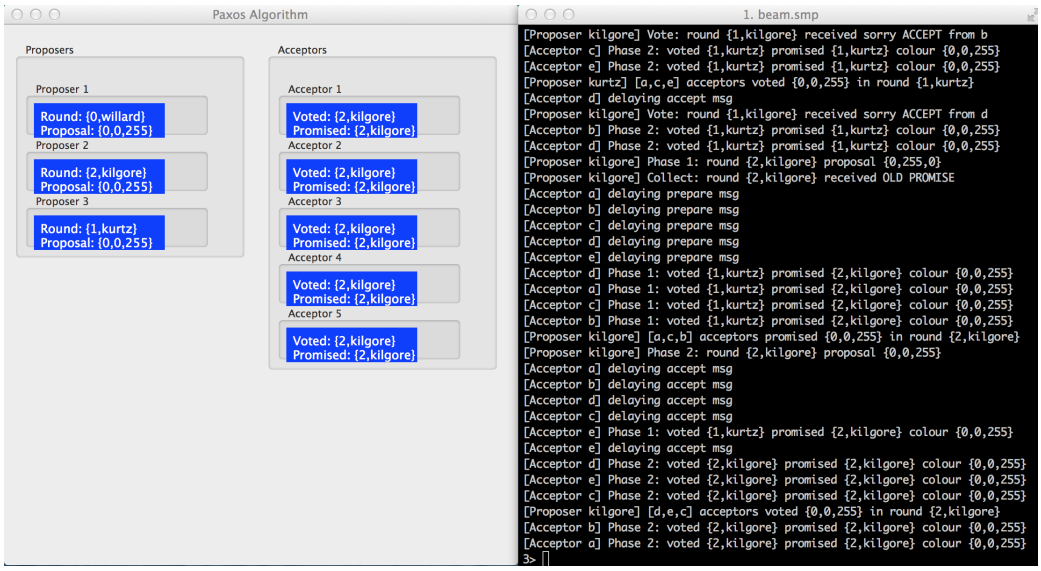
i) In this experiment we delay *prepare* messages before sending them to proposers and we see that the algorithm terminates even with the delays shown in the trace.



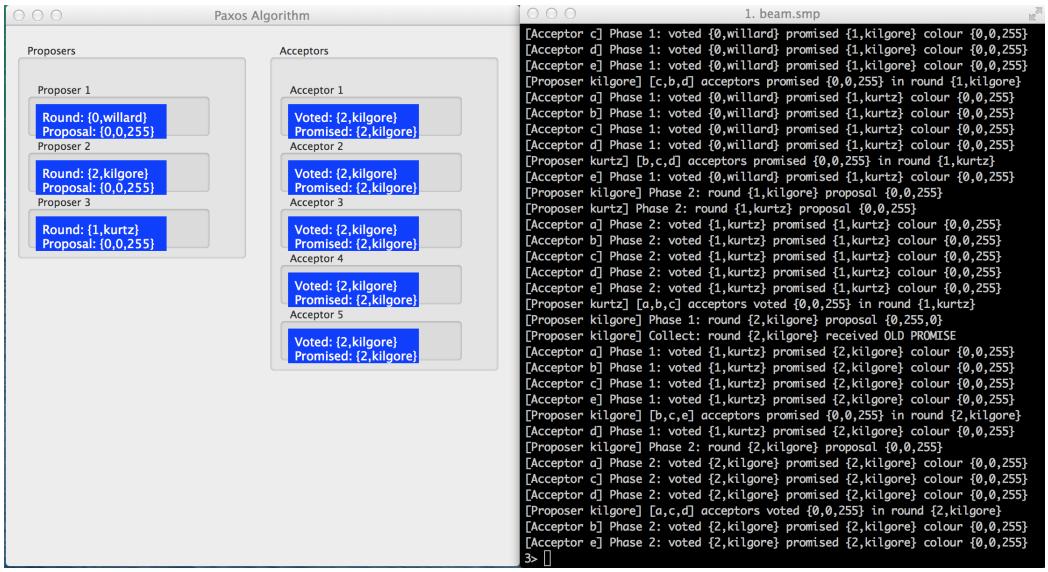
Now we add delays before the *accept* messages are sent to proposers, the algorithm terminates voting in round 2 for *kilgore* and color *blue*.



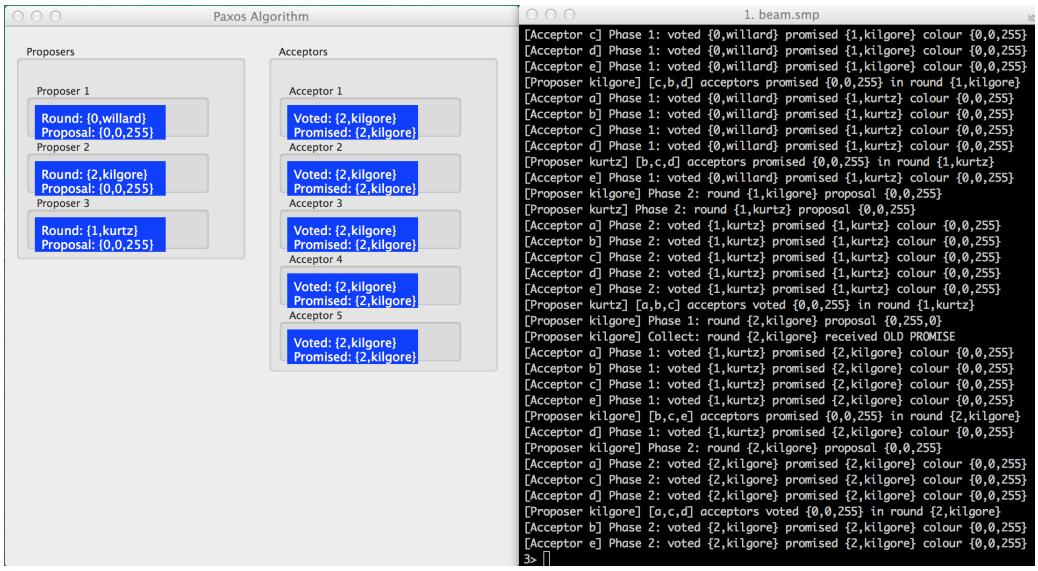
Lastly, we add delays both before *prepare* and *accept* messages are sent to proposers and the acceptors reach the same consensus again.



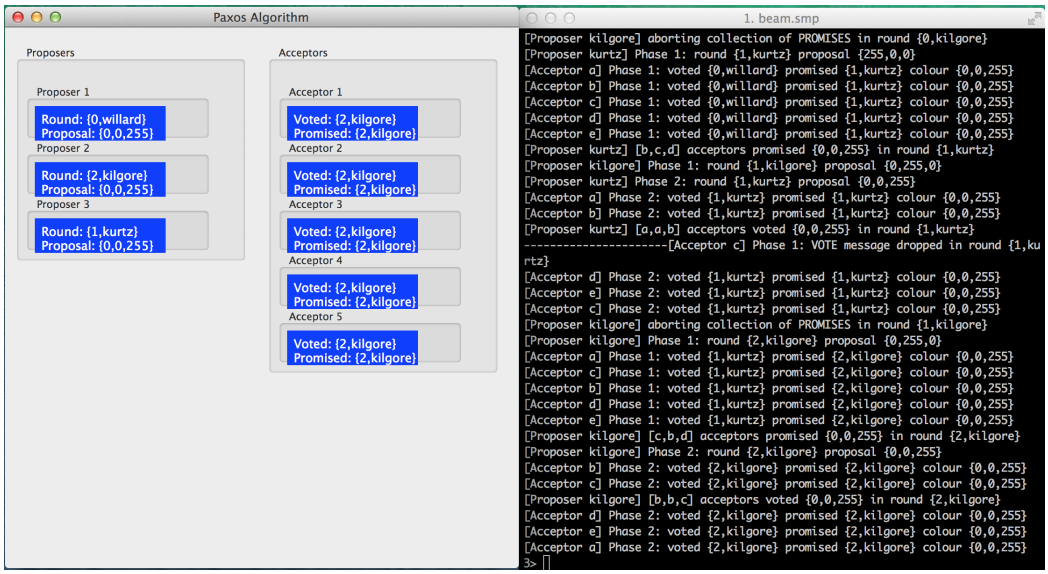
ii) We stop sending *sorry* messages to the processor and we run the experiment to try to reach a consensus.



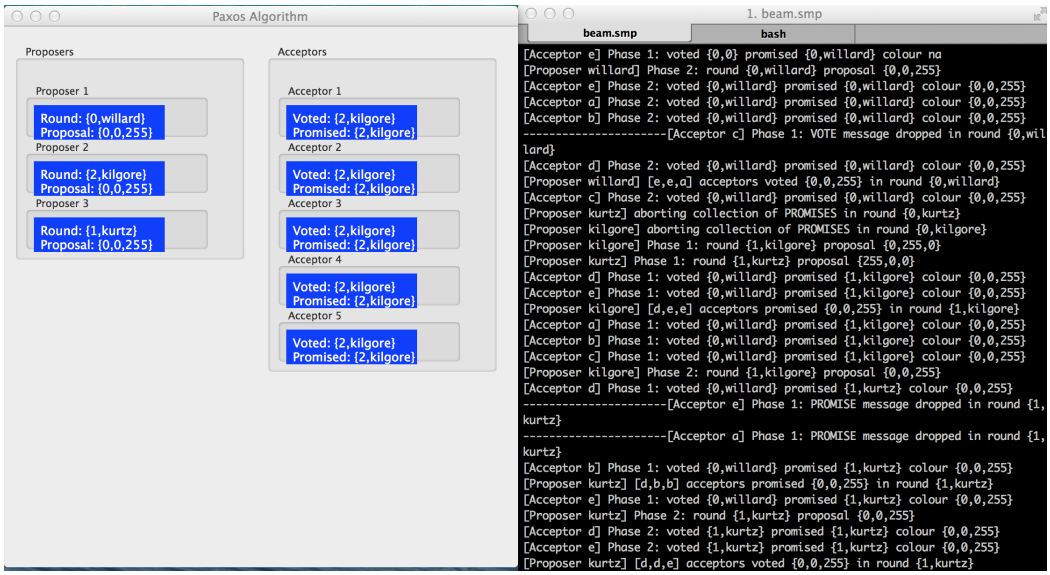
iii) In this experiment, we randomly drop *promise* messages, and we look out for the consensus that the acceptors arrive to. Even though a promise message in round 2 is dropped, they still agree on the color blue.



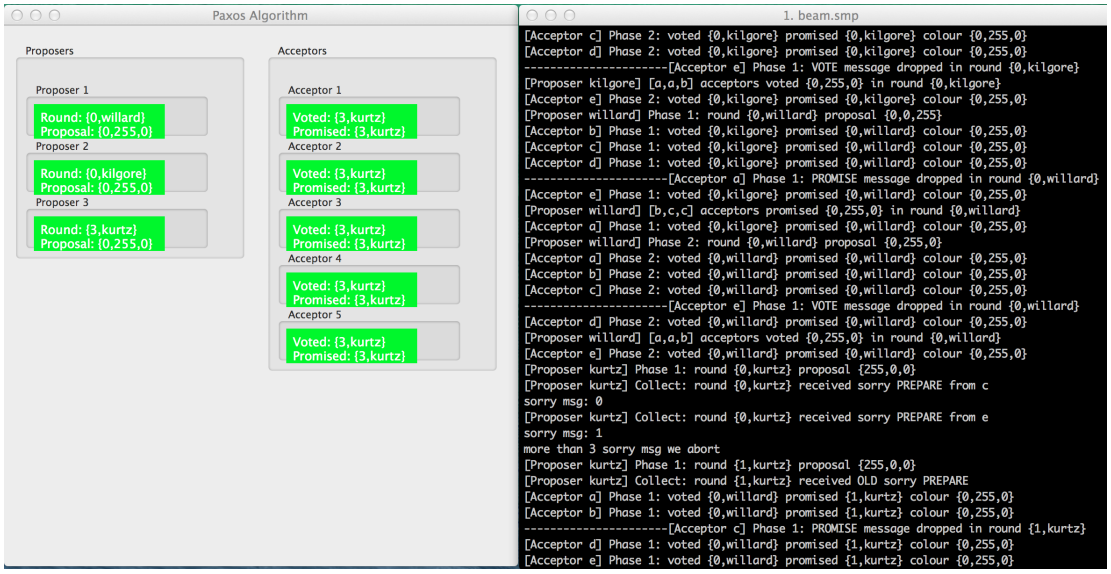
Now, we run an experiment dropping *vote* messages, this is, not replying to proposers with *sorry* messages when asked to vote. Again, the acceptors manage to agree on the proposed color blue, even with a *vote* message dropped because the other messages still amount to the majority of acceptors.



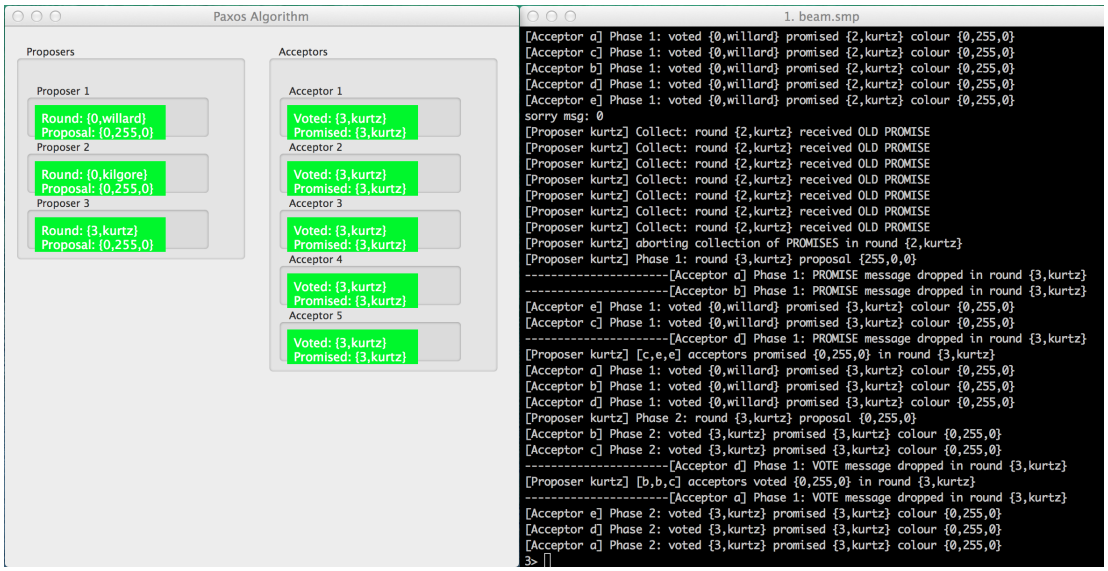
When we run the experiment dropping both *promise* and *vote sorry* messages, we see that the algorithm still terminates successfully due to the fact that the majority of acceptors whose messages were not dropped agreed.



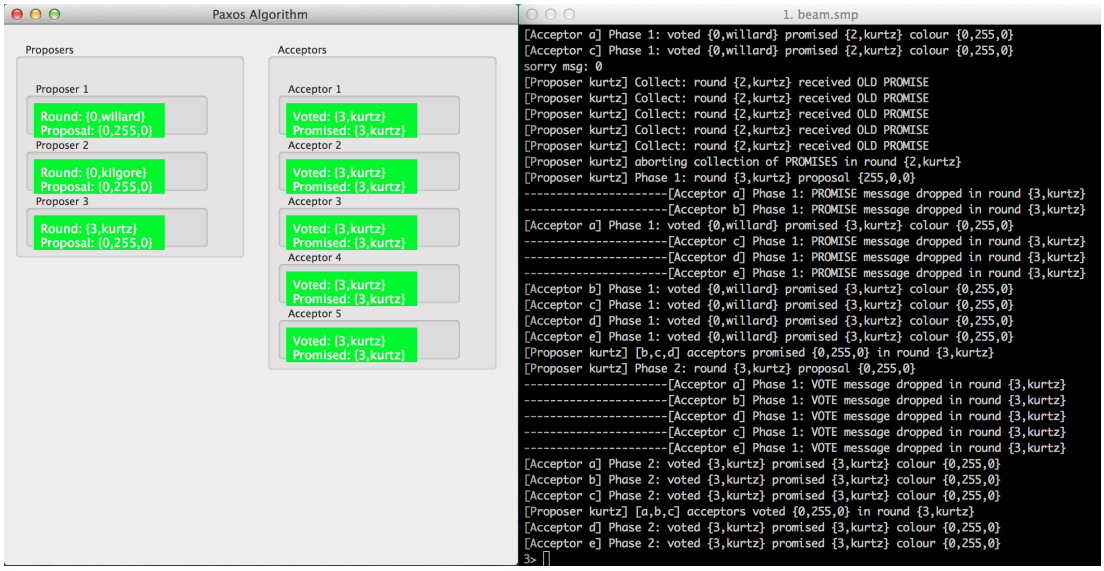
We also run the experiment by dropping more than one *promise* and *vote* messages in a total of 10 messages. Here is the result of dropping one message in 5. The algorithm still terminates and the acceptors reach the consensus.



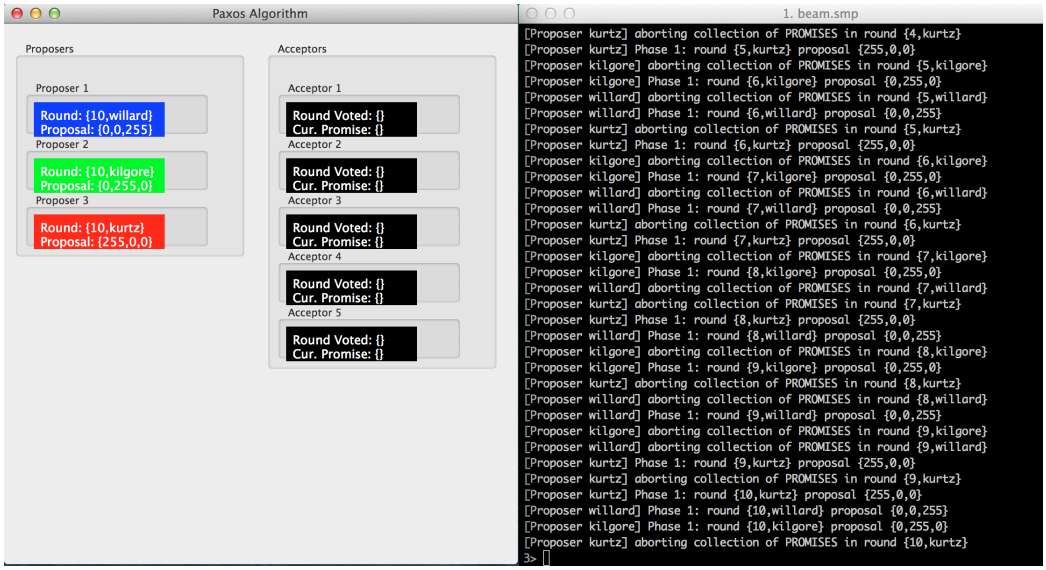
We ran more experiments by dropping one message in 3:



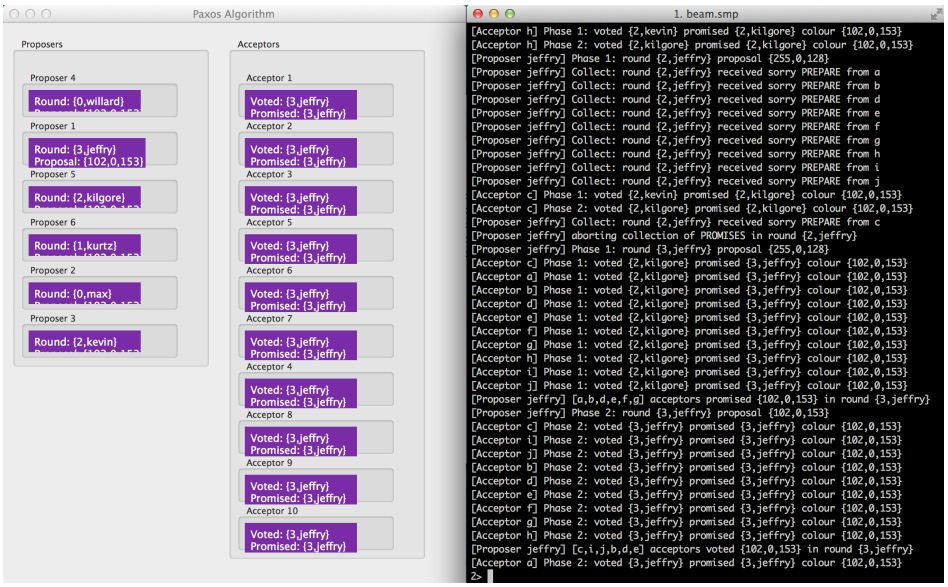
And one message in 1:



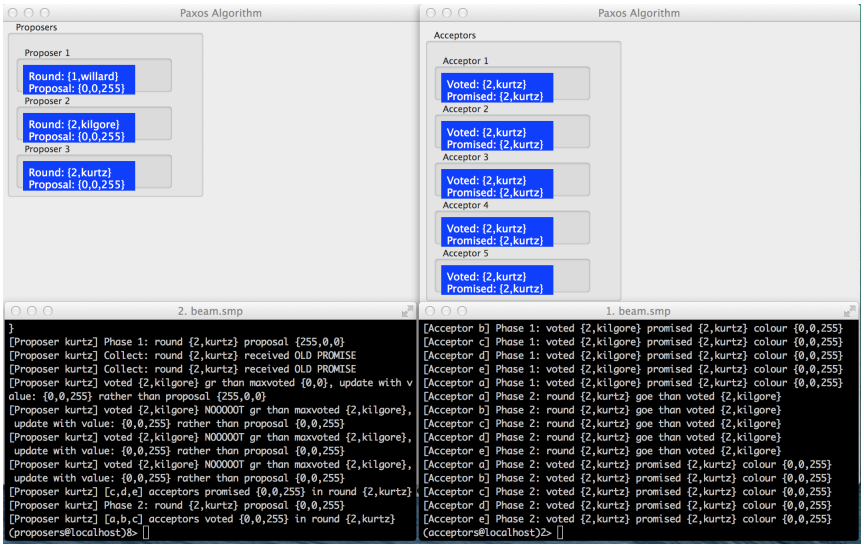
And even though more and more messages are dropped an agreement is eventually met. However if we add the line: *-define(drop, 0).* then the algorithm does not terminate.



iv) We increase the number of acceptors to 10 and the number or proposers to 6. The acceptors still get to a consensuns in round 3 on color *purple*.

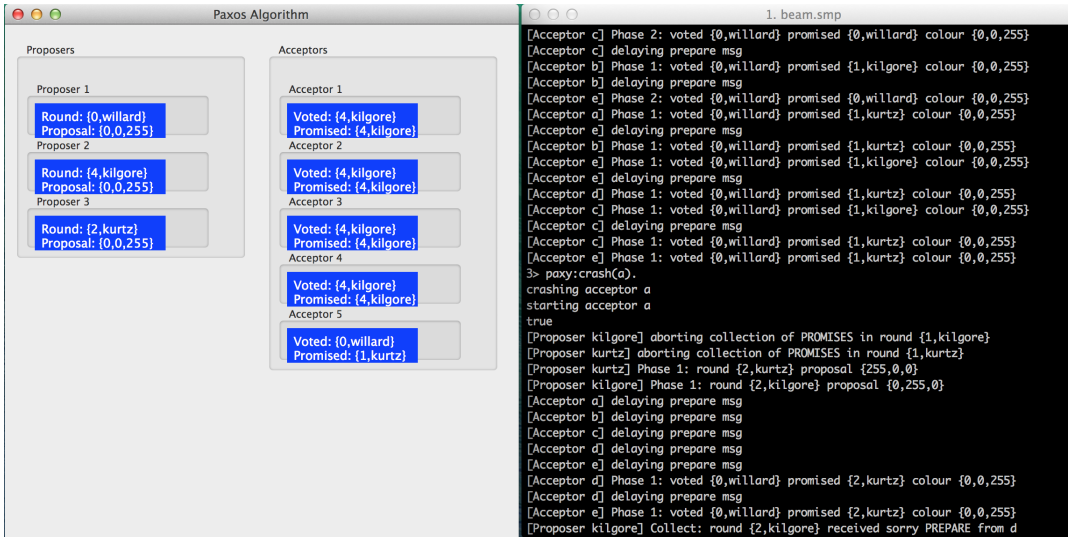


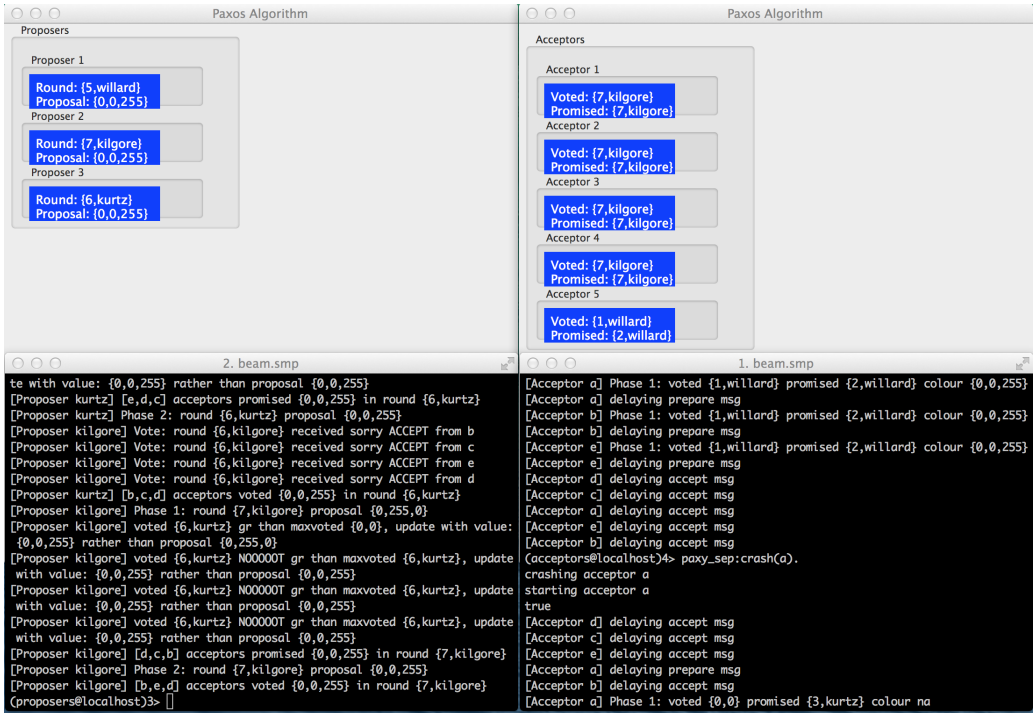
v) In this experiment we changed the modules to adjust them such that we can run the proposers in one node and the acceptors in a different one and as they communicate they reach an agreement on the color *blue* in round 2.



3.1 Fault Tolerance

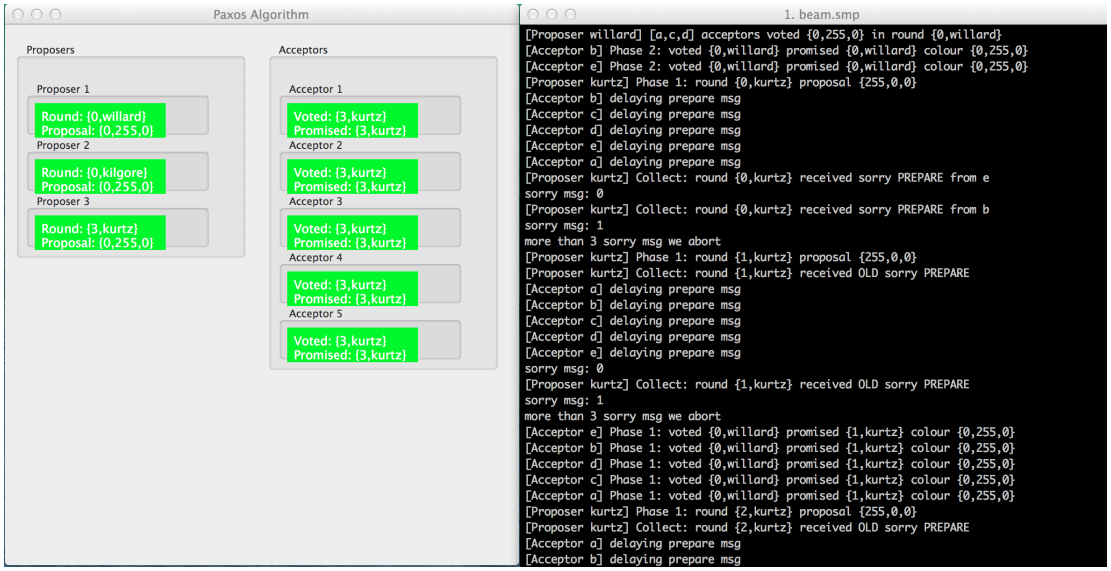
In order to test the fault tolerance we added the *crash* function in the *paxy* module and while running the experiment, we crashed one of the acceptors. Analyzing the trace we see that the acceptor starts again and continues to participate and the algorithm terminates. We ran the experiment both on the same machine and on separate machines for the proposers and acceptors.





3.2 Improvements

With five acceptors, a proposal needs to be voted on at least 3 times. We changed the *collect* and *vote* functions in the *proposer* module, such that once it receives 3 *sorry* messages it aborts for this proposal. In the experiments trace we see the count of the *sorry* messages and the moment it aborts.



4 Personal Opinion

The seminar had the right level of difficulty for this course and the amount of time available to us. We also got a broader understanding of the *Paxos* algorithm and we managed to run it locally as well as in a distributed system. We are aware that this implementation could have several improvements and modifications as described in the *Leslie Lamport* paper. For instance the algorithm can be extended to work with a group of *learners* processes, and ultimately it can run in a group of servers where each server can play all three roles.