Paxy

David Sarda, Kymry Burwell, Michael Sammler

December 16, 2017

1 Introduction

The Paxy seminar introduces and implements the Paxos algorithm. In the broad field of distributed systems, Paxos deals with state machine replication in asynchronous environments. One of the difficulties in working with distributed systems is gaining consensus amongst remote processes that may crash and re-spawn at anytime. The Paxos algorithm solves the consensus problem and guarantees safety, even if servers fails. For this report we implemented the Paxos algorithm in Erlang and evaluated it based on different experiments.

2 Work done

For this seminar we implemented the Paxos algorithm in Erlang based on the templates in the assignment. The Erlang files in the root folder contain the implementation of Paxos with all the variants described in the problem statement. The implementation in the remote folder is adapted to work with on multiple nodes as described in experiment v) of the problem statement. This implementation is described in the section regarding experiment v). The main implementation also contains code to make the experiments easier. For example a `paxy:wait/0` function was added, which waits, until all proposers have decided. Also some compile time constants were added to be able to control the delays and the drop rates for the experiments. experiment_1.escript contains a script, which runs the consensus, checks that all proposers decide on the same value and outputs the number of rounds it took until consensus was reached.

3 Experiments

Experiment i) For this experiment, we introduced different delays in the acceptor after receiving prepare and/or accept messages. The results of these experiments can be seen in figure 1. This figure shows, that a delay in the accept messages alone only slows down the algorithm by some rounds,

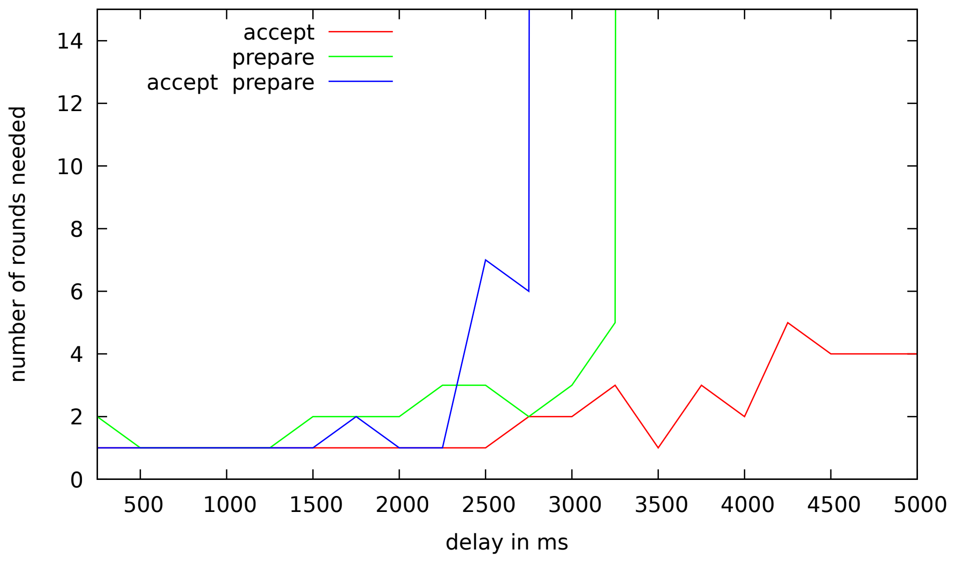


Figure 1: Number of rounds for different delays

whereas the delay at the prepare messages and at both message types made the algorithm not terminate in a reasonable time frame (1 minute) for a delay of more than 3250ms resp. 2750ms.

Experiment ii) This experiment is about testing, whether Paxos still terminates, even if no sorry messages are sent. For this the sending of sorry messages was commented out. The result was, that the algorithm still terminates. This was expected, because sorry messages are only an optimization in Paxos. The description of Paxos by Lamport in Paxos made easy does not use sorry messages and still works.

Experiment iii) For this exercise we should try, whether the Paxos algorithm still terminates, if some percentage of the vote and/or promise messages are dropped. Figure 2 shows how many rounds were needed for consensus, if the specified percentage of messages was lost. Figure 2 shows, that if only promise messages are lost, up to 70% message loss can be tolerated, before Paxos takes long to terminate, 60% for vote and 50%, if both promise and vote messages can be lost. It makes sense, that it takes a lot of time for Paxos to terminate, if more than 50% of the messages are lost, because it is hard to receive messages from a majority of acceptors, if more than half of the messages get lost.

Experiment iv) This experiment is about trying to increase the number of acceptors and proposers. A screenshot of the program running with more

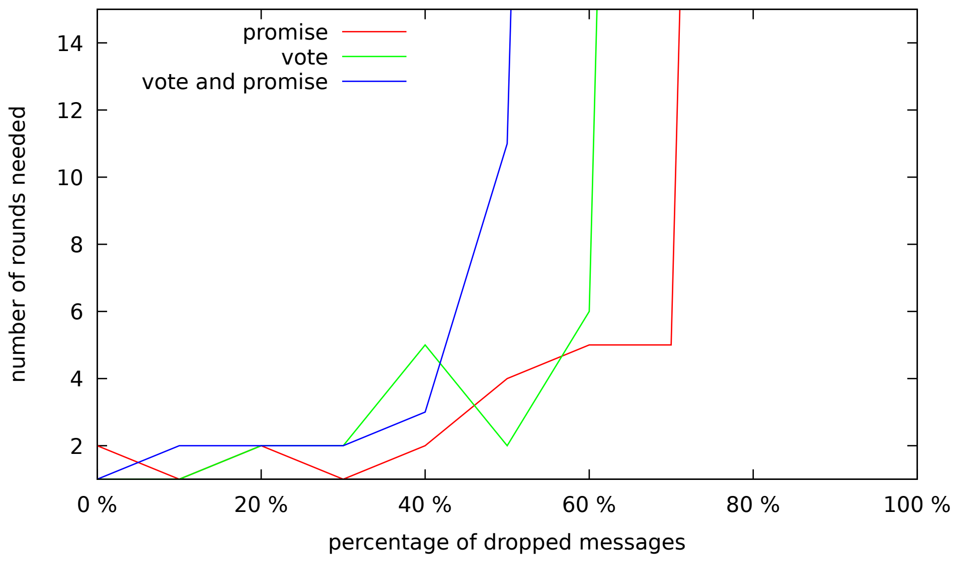


Figure 2: Number of rounds for different drop rates

acceptors and proposers can be seen in figure 3. There was no noticeable difference in the number of rounds, which were needed to reach a decision, if more acceptors and proposers were used.

Experiment v) For this experiment the paxy module was adapted to create the acceptors and proposers in a remote instance. The adapted code can be found in the remote subdirectory. To execute the code, the following steps have to be taken:

1. Start the proposer node by executing

   ```
   erl -sname paxy-pro -setcookie secret
   ```

   and then loading the proposer in it by executing `c(proposer).`

2. Start the acceptor node by executing

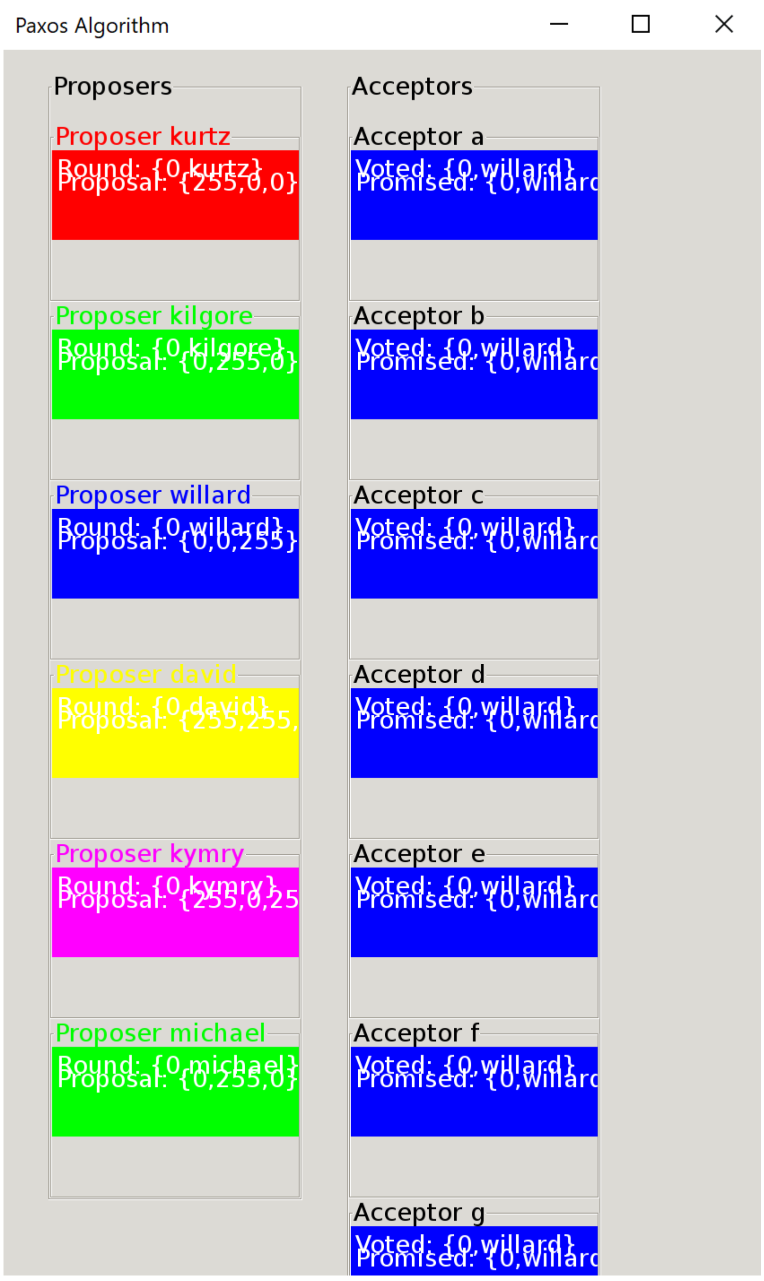
   ```
   erl -sname paxy-acc -setcookie secret
   ```

   and then loading the acceptor in it by executing `c(acceptor).`

3. Start the main node by executing

   ```
   erl -sname master -setcookie secret
   ```

   and then loading paxy with `c(paxy).` and starting it with `Pids = paxy:start([10, 10, 10]).`. Pids can be used, to stop the acceptors by calling `paxy:stop(Pids)`,


Paxos Algorithm

Proposers

Proposer kurtz
Round: {0,kurtz}
Proposal: {255,0,0}

Proposer kilgore
Round: {0,kilgore}
Proposal: {0,255,0}

Proposer willard
Round: {0,willard}
Proposal: {0,0,255}

Proposer david
Round: {0,david}
Proposal: {255,255,

Proposer kymry
Round: {0,kymry}
Proposal: {255,0,25

Proposer michael
Round: {0,michael}
Proposal: {0,255,0}

Acceptors

Acceptor a
Voted: {0,willard}
Promised: {0,willar

Acceptor b
Voted: {0,willard}
Promised: {0,willar

Acceptor c
Voted: {0,willard}
Promised: {0,willar

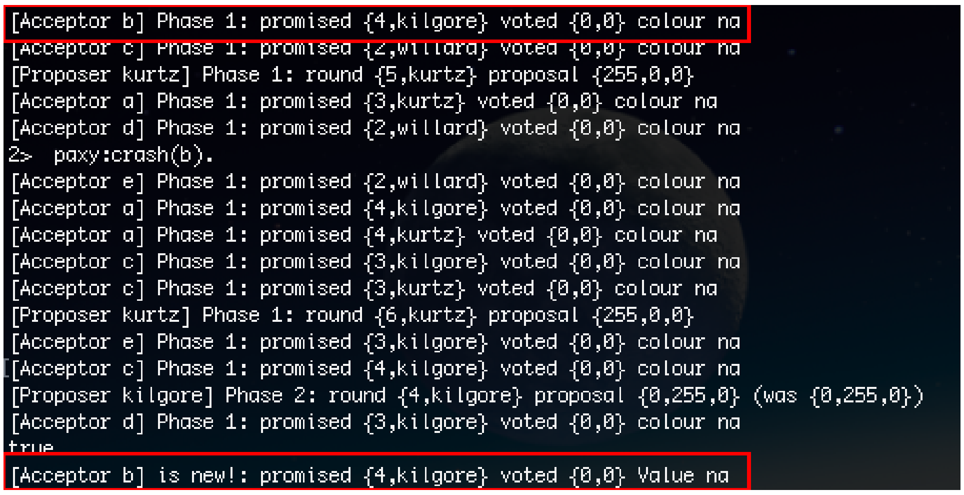
Acceptor d
Voted: {0,willard}
Promised: {0,willar

Acceptor e
Voted: {0,willard}
Promised: {0,willar

Acceptor f
Voted: {0,willard}
Promised: {0,willar

Acceptor g
Voted: {0,willard}
Promised: {0,willar

Figure 3: Screenshot of the Paxos algorithm with more acceptors and proposers



```
[Acceptor b] Phase 1: promised {4,kilgore} voted {0,0} colour na
[Acceptor c] Phase 1: promised {2,willard} voted {0,0} colour na
[Proposer kurtz] Phase 1: round {5,kurtz} proposal {255,0,0}
[Acceptor a] Phase 1: promised {3,kurtz} voted {0,0} colour na
[Acceptor d] Phase 1: promised {2,willard} voted {0,0} colour na
2> paxy:crash(b).
[Acceptor e] Phase 1: promised {2,willard} voted {0,0} colour na
[Acceptor a] Phase 1: promised {4,kilgore} voted {0,0} colour na
[Acceptor a] Phase 1: promised {4,kurtz} voted {0,0} colour na
[Acceptor c] Phase 1: promised {3,kilgore} voted {0,0} colour na
[Acceptor c] Phase 1: promised {3,kurtz} voted {0,0} colour na
[Proposer kurtz] Phase 1: round {6,kurtz} proposal {255,0,0}
[Acceptor e] Phase 1: promised {3,kilgore} voted {0,0} colour na
[Acceptor c] Phase 1: promised {4,kilgore} voted {0,0} colour na
[Proposer kilgore] Phase 2: round {4,kilgore} proposal {0,255,0} (was {0,255,0})
[Acceptor d] Phase 1: promised {3,kilgore} voted {0,0} colour na
true
[Acceptor b] is new!: promised {4,kilgore} voted {0,0} Value na
```

Figure 4: Server crash and restart

Fault Tolerance For this experiment we simulated a crash and restart of a server. In order to check that the server restarted with the same state that it crashed with, we amended the code to have the server print out it's state upon restart. Figure 4 above shows a crash of 'acceptor b' and it's subsequent restart. We can see that it's state at time of crash and restart were the same.

Improvement based on sorry messages For this section we implemented the changes proposed on the statement by extending the procedures *collect*/4 and *vote*/2 to abort the ballot when we have received $\lceil \frac{n}{2} \rceil$ sorry messages. Afterwards we performed experimentations to see what was the improvement over the time and rounds it took to reach consensus, if any. For this we computed ten repetitions of the possible four combinations, this combinations being having the improvement for the procedure *collect*/4 only, for the procedure *vote*/2 only, for both, or for neither (like in the original version). This was done on the basic case of 3 proposers and 5 acceptors, all proposers starting out at the same time. In table 1 we can see the results of these experiments:

Version	**Rounds**	**Time(s)**
Neither	1.5	3.588
Collect/4	1.181	2.565
Vote/2	1.8	3.022
Collect/4 & Vote/2	1.72	0.592

Table 1: Mean for rounds and time taken for all versions to reach consensus for ten repetitions

As we can see, there is a drastic improvement in the time taken to reach consensus when implementing fully this new version, as it reaches execution times of up to 6 times faster on average than the previous version. We can also observe than only applying one of the two changes also results as a decent improvement over the original version. In addition, we can note some differences between the rounds taken from all versions, but there isn't any profound change on them, and seems to depend more on the specific execution performed.

4 Open questions

The open questions were answered in the Experiments section.

5 Personal opinion

We all agree, that the assignment was a useful exercise. It was interesting to learn about the internals of this algorithm through implementation. The practical application helped a lot to understand what was taught in theory before. This seminar should be included in the curriculum of the next year as well.