Seminar Report: Paxy

Ignacio Encinas Rubio, Adrián Jimenez González

September 22, 2022

1 Introduction

The Paxy seminar consists in understanding the Paxos protocol through an Erlang implementation. First, we will develop a basic understanding of the underlying algorithm when filling the gaps in the template implementation. Later on we will build on this understanding and see the effects of different modifications such as introducing message delays, message drops, removing sorry messages and so on. Some of these modifications are specially important because they are responsible for making it impossible to guarantee a consensus in an asynchronous system.

Paxos is extensively used in production systems, so getting practical experience with it and examining how it behaves under different circumstances will prove very useful.

2 Code modifications

In this section we will briefly comment the code added to the template version in order to make the algorithm work. We will show the minimum number of lines of code possible to follow the reasoning.

2.1 Proposer.erl

```
case ballot(Name, ..., ..., ..., PanelId
    ) of
{ok, Value} ->
  {Value, Round};
abort ->
  timer:sleep(rand:uniform(Backoff)),
  Next = order:inc(...),
  round(Name, (2*Backoff), ..., Proposal
   , Acceptors, PanelId)
end.
```

Listing 1: Template

```
case ballot(Name, Round, Proposal,
   Acceptors, PanelId) of
{ok, Value} ->
  {Value, Round};
abort ->
  timer:sleep(rand:uniform(Backoff)),
  Next = order:inc(Round),
  round(Name, (2*Backoff), Next,
   Proposal, Acceptors, PanelId)
end.
```

Listing 2: Filled version

The first incomplete function is **round**. The ballot function is responsible for creating the *prepare* and *accept* messages from the proposer. It will need the Round in order to generate the prepare message, the Proposal to generate the accept message and the list of Acceptors for being able to send the actual messages.

If we're not able to get our Proposal accepted we'll try again with a higher Round number after backing off for a while in order to facilitate consensus.

```
prepare(..., ...),
Quorum = (length(...) div 2) + 1,
MaxVoted = order:null(),
case collect(..., ..., ..., ...) of
  {accepted, Value} ->
    accept(..., ..., ...),
    case vote(..., ...) of
      ok ->
        {ok, ...};
      abort ->
        abort
```

Listing 3: Template

```
prepare(Round, Acceptors),
Quorum = (length(Acceptors) div 2) + 1,
MaxVoted = order:null(),
case collect(Quorum, Round, MaxVoted,
   Proposal) of
  {accepted, Value} ->
    accept(Round, Value, Acceptors),
    case vote(Quorum, Round) of
      ok ->
        {ok, Value};
      abort ->
        abort
```

Listing 4: Filled version

For the **ballot** function we need to add *Round and Acceptors* as parameters for the *prepare* function. With this function we send the prepare message with the Round information to the list of Accpeptors.

In the next step, we need is calculate the necessary votes to achieve consensus, which is calculated as the number of Acceptors divided by 2, plus 1.

The parameters of *collect* function must be Quorum in order to know if we achieve the consensus, and Round, MaxVoted to know the maximum Round the Acceptor has promised and Proposal. In case of this function returns an accepted (also called *promise* on the diagram) we call the *accept* function with *Round, Value, Acceptors* values as parameters to send the accept message.

Following that function, we call *vote* fuction inside of a case statement with the Quorum and the Round, in case we recieve an ok the function will return *{ok, Value}*.

```
collect(N, Round, MaxVoted, Proposal) ->
  receive
    {promise, Round, _, na} ->
      collect(..., ..., ..., ...);
    {promise, Round, Voted, Value} ->
      case order:gr(..., ...) of
        true ->
          collect(..., ..., ..., ...);
        false ->
          collect(..., ..., ..., ...)
      end;
    {promise, _, _, _} ->
      collect(N, Round, MaxVoted, Proposal
     );
    {sorry, {prepare, Round}} ->
      collect(..., ..., ..., ...);
    {sorry, _} ->
      collect(N, Round, MaxVoted, Proposal
     )
```

Listing 5: Template

```
collect(N, Round, MaxVoted, Proposal) ->
  receive
    {promise, Round, _, na} ->
      collect(N-1, Round, MaxVoted,
     Proposal);
    {promise, Round, Voted, Value} ->
      case order:gr(Voted, MaxVoted) of
        true ->
          collect(N-1, Round, Voted, Value
     );
        false ->
          collect(N-1, Round, MaxVoted,
     Proposal)
      end;
    {promise, _, _, _} ->
      collect(N, Round, MaxVoted, Proposal
     );
    {sorry, {prepare, Round}} ->
      collect(N, Round, MaxVoted, Proposal
     );
    {sorry, _} ->
      collect(N, Round, MaxVoted, Proposal
     )
```

Listing 6: Filled version

Terminar cuando TODO esté hecho. Preguntar al profesor en clase

For the **collect** function, we expect to receive the reply from the acceptor to our accept. Depending on the message received our function will do:

- If the message is *{promise, Round, _, na}*, we do recursion substracting 1 to N. Keep

collecting 'support' after getting one promise.

- If the message is *{promise, Round, Voted, Value}*, we are getting the promise but we need to update the maximum Voted/Proposal. That is what we do in the next case statement. We use the maximum between voted and MaxVoted for the recursion.
- If the message is *{sorry, {prepare, Round}}*, we keep trying to get support of other Acceptors.

```
vote(N, Round) ->
  receive
    {vote, Round} ->
      vote(..., ...);
    {vote, _} ->
      vote(N, Round);
    {sorry, {accept, Round}} ->
      vote(..., ...);
    {sorry, _} ->
      vote(N, Round)
  after ?timeout ->
    abort
  end.
```

Listing 7: Template

```
vote(N, Round) ->
  receive
    {vote, Round} ->
      vote(N-1, Round);
    {vote, _} ->
      vote(N, Round);
    {sorry, {accept, Round}} ->
      vote(N, Round);
    {sorry, _} ->
      vote(N, Round)
  after ?timeout ->
    abort
  end.
```

Listing 8: Filled version

In the **vote** function we expect to recieve a vote or sorry message from the Acceptors. In this function we define the behavior depending what we receive:

- In case of receive a vote like *{vote, Round}*, we subsctact 1 to the votes necessary for consensus (N) to the next vote function with the same Round.
- If we receive *{sorry, {accept, Round}}* that means that the vote was rejected and we keep trying, so we do not substract nothing to the number of votes remaining for achieve consensus on the next vote function, passing N and Round again as parameters.

2.2 Acceptor.erl

```
receive
  {prepare, Proposer, Round} ->
    case order:gr(..., ...) of
      true ->
        ... ! {promise, ..., ..., ...},
        acceptor(Name, ..., Voted, Value
    , PanelId);
      false ->
        ... ! {sorry, {prepare, ...}},
        acceptor(Name, ..., Voted, Value
    , PanelId)
    end;
```

Listing 9: Template

```
receive
  {prepare, Proposer, Round} ->
    case order:gr(Round, Promised) of
      true ->
        Proposer ! {promise, Round,
    Voted, Value},
        acceptor(Name, Round, Voted,
    Value, PanelId);
      false ->
        Proposer ! {sorry, {prepare,
    Round}},
        acceptor(Name, Promised, Voted,
    Value, PanelId)
    end;
```

Listing 10: Filled version

The Acceptor code only has **acceptor** function to fullfil. With this function we specify the desided behavior of Acceptors. The program receives the messages of the proposers.

In case we receive a prepare message, we compare the Round that comes inside the message with our highest promised.

- If the Round variable is higher we send a *promise* message with the new Round, the previous *Voted* round and *value*. Then we update our Promised value with the Round.
- Else if the value recieved in Round is lower, we send a *sorry* message to the acceptor with the Round we are refusing.

```
{accept, Proposer, Round, Proposal} ->
  case order:goe(..., ...) of
    true ->
      ... ! {vote, ...},
      case order:goe(..., ...) of
        true ->
          acceptor(Name, Promised,
... , ... , PanelId);
        false ->
          acceptor(Name, Promised,
... , ... , PanelId)
      end;
    false ->
      ... ! {sorry, {accept, ...}},
      acceptor(Name, Promised, Voted,
Value, PanelId)
```

Listing 11: Template

```
{accept, Proposer, Round, Proposal} ->
  case order:goe(Round, Promised) of
    true ->
      Proposer ! {vote, Round},
      case order:goe(Round, Voted) of
        true ->
          acceptor(Name, Promised,
Round, Proposal, PanelId);
        false ->
          acceptor(Name, Promised,
Round, Voted, PanelId)
      end;
    false ->
      Proposer ! {sorry, {accept,
Round}},
      acceptor(Name, Promised, Voted,
Value, PanelId)
  end;
```

Listing 12: Filled version

The second type of message acceptor can receive is *accept* message. In this case, we need to compare the *Round* of the message with our *Promised* value.

- If the *Round* is bigger or equal to the *Promised*, we vote for the *Proposal*. Then, in case the *Round* is bigger or equal to *Voted*, we update the highest numbered proposal, passing *Round* value as *Voted*. Else if it is lower, we update the the *Voted* and *Value* with *Round* and *Proposal*.
- If the *Round* is lower than the *Promised*, we send a *sorry* message with the *Round* value we are refusing.

3 Experiments

Provide evidence of the experiments you did (e.g., use screenshots) and discuss the results you got. In addition, you may provide figures or tables with experimental results of the system evaluation. For each seminar, we will provide you with some guidance on which kind of evaluation you should do.

3.1 Introducing delays in the promise and vote messages

We have tried adding delay both to the promise and vote messages. The expected result is that adding delay will increase the time needed to reach consensus. As we're dealing with random delays there is a noticeable oscillation and the runtime increase is highly volatile.

The increase in runtime is important but relatively controlled for small delays when compared to the proposer timeout. The problem arises once our delay becomes so big that we start timing out in our proposers and we can't get work done, thus our algorithm starts taking too much time. It is possible that it eventually achieves consensus but further

examination is not feasable, and it wouldn't really matter as we need a somewhat reliable algorithm.

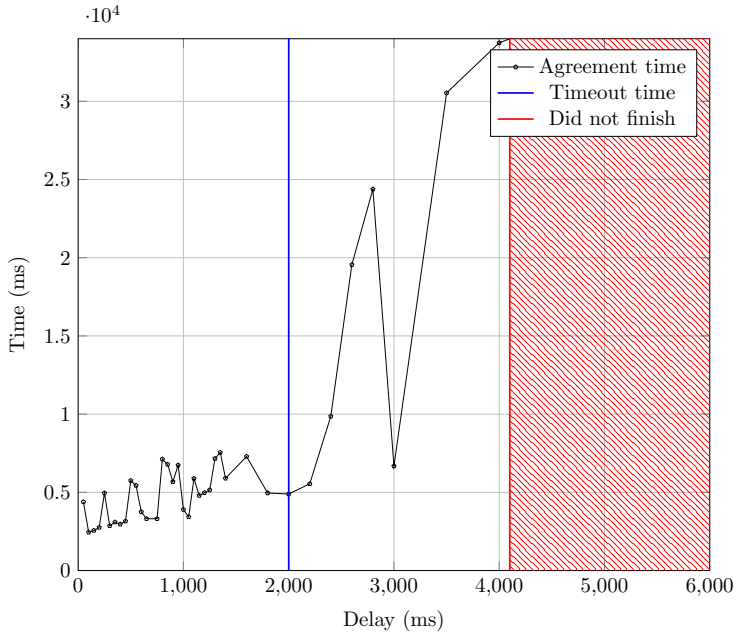

Figure 1: Results with proposer timeout = 2000

The number of rounds is 2 or less for almost every delay. It only increases along with the runtime spikes we see starting at $delay = 2800$, where the decision starts taking 6, 9 and up to 10 rounds.

3.2 Avoid sending sorry messages

3.2.1 Could you come to an agreement when sorry messages are not sent?

3.3 Try randomly dropping promise and vote messages in the acceptor

3.3.1 How does the drop ratio affect the number of rounds required to reach consensus?

3.4 Try increasing the number of acceptors and proposers

3.4.1 What is the impact of having more acceptors while keeping the number of proposers?

3.5 Adapt the paxy module to create the proposers and acceptors in a remote Erlang instance

4 Fault tolerance

4.1 Experiments

5 Improvement based on sorry messages

5.1 Experiments

6 Open questions

Try to answer all the open questions in the documentation. When possible, do experiments to support your answers.

7 Personal opinion

Provide your personal opinion of the seminar, indicating whether it should be included in next year's course or not.