Seminar Report: Paxy

Ignacio Encinas Rubio, Adrián Jimenez González

September 21, 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

```erlang
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

```erlang
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

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

3.1.1 Does the algorithm still terminate? Does it require more rounds? How does the impact of the message delays depend on the value of the timout at the proposer?

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.