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

```
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.

```
ballot(Name, Round, Proposal, Acceptors, PanelId) ->
  prepare(..., ...),
  Quorum = (length(...) div 2) + 1,
  MaxVoted = order:null(),
  case collect(..., ..., ..., ...) of
    {accepted, Value} ->
      io:format("[Proposer ~w] Phase 2: round ~w proposal ~w (was ~w)~n",
                [Name, Round, Value, Proposal]),
      % update gui
      PanelId ! {updateProp, "Round: " ++ io_lib:format("~p", [Round]), Value},
      accept(..., ..., ...),
      case vote(..., ...) of
        ok ->
          {ok, ...};
        abort ->
          abort
      end;
    abort ->
      abort
  end.
```

```
ballot(Name, Round, Proposal, Acceptors, PanelId) ->
  % Send prepare message with round information
  prepare(Round, Acceptors),
  % Necessary votes
  Quorum = (length(Acceptors) div 2) + 1,
  MaxVoted = order:null(),
  % Quorum vamos haciendole -1 hasta llegar a 0
  case collect(Quorum, Round, MaxVoted, Proposal) of
    {accepted, Value} ->
      io:format("[Proposer ~w] Phase 2: round ~w proposal ~w (was ~w)~n",
                [Name, Round, Value, Proposal]),
      % update gui
      PanelId ! {updateProp, "Round: " ++ io_lib:format("~p", [Round]), Value},
      % We got promised, lets ask for votes
      accept(Round, Value, Acceptors),
      case vote(Quorum, Round) of
        ok ->
          {ok, Value};
        abort ->
          abort
      end;
    abort ->
      abort
```

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. In the next step, we need is calculate the necessary votes, which are calculated as the number of Acceptors divided by 2, plus 1. The parameters of *collect* function must be *Quorum, Round, MaxVoted, Proposal*. In case of this function returns an accept we call the *accept* function with *Round, Value, Acceptors* values as parameters. Following that function, we call *vote* fuction inside of a case statement, in case we recieve an ok we 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)
  after ?timeout ->
    abort
  end.
```

```
collect(N, Round, MaxVoted, Proposal) ->
  receive
    % Promise received, no previous votes. Keep collecting 'support'
    {promise, Round, _, na} ->
      collect(N-1, Round, MaxVoted, Proposal);
    {promise, Round, Voted, Value} ->
      % We got the promise. Update the maximum Voted/Proposal
      case order:gr(Voted, MaxVoted) of
  % Learn value
        true ->
          collect(N-1, Round, Voted, Value);
  % Keep this proposal
        false ->
          collect(N-1, Round, MaxVoted, Proposal)
      end;
    % TODO: Old message, ignore and keep going?
    {promise, _, _,  _} ->
      collect(N, Round, MaxVoted, Proposal);
    % Rejected, just keep gathering support
    {sorry, {prepare, Round}} ->
      collect(N, Round, MaxVoted, Proposal);
    % TODO: Old message from message or whatever?
    {sorry, _} ->
      collect(N, Round, MaxVoted, Proposal)
  after ?timeout ->
    abort
```

Terminar cuando TODO esté hecho

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

```
vote(N, Round) ->
  receive
    {vote, Round} ->
      vote(N-1, Round); % voto ganado, uno menos
    {vote, _} -> % voto desactualizado?
      vote(N, Round);
    {sorry, {accept, Round}} ->
      vote(N, Round); % Rejected, keep going
    {sorry, _} ->
      vote(N, Round) % Rejected from other round or from the promise
  after ?timeout ->
    abort
  end.
```

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 vote necessary for consensus. 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.

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.