Paxy: the paxos protocol

Johan Montelius

June 13, 2012

Introduction

This exercise will give you the opportunity to learn the Paxos algorithm for gaining consensus in a distributed system. You should know the basic operations of the algorithm but you do not have to know all the details, that is the purpose of this exercise.

The code given is not complete, we use ... etc to indicate that you have to fill in the missing pieces.

1 Paxos

The Paxos algorithm has three different processes: proposers, acceptors and learners. The functionality of all three is often included in one process but it will be easier to implement the proposer and acceptor as two separate processes. The learner process will not be implemented since it is not needed to reach a consensus. In a real system it is of course important to also know the outcome of the algorithm but we will do without learners.

1.1 sequence numbers

We will need some basic support to handle sequence numbers. Since proposers need unique sequence numbers we need a way to generate and compare sequence numbers. One way of guarantee uniqueness is to use a tuple and let the first element be an, per proposer, increasing integer and the second an identifier unique for the proposer. We build a small `order` module that can be found in the appendix to this description. It will be quite easy to see what we mean when we use the exported functions.

1.2 the acceptor

Let's start with the acceptor. The acceptor has a state consisting of:

- **Name**: the Name of the acceptor,
- **Promise**: promised not to accept any ballot below this number
- **Voted**: the highest ballot number accepted
- **Accepted**: the value that has been accepted

Note that an acceptor can accept many values during the execution but we must remember the value with the highest ballot number.

When we start an acceptor we have not promised anything nor accepted a value so the `Promise` and `Voted` parameters are instantiated to null sequence numbers that are lower than any other sequence number. The `Accepted` parameter is initialized to `na` to indicate that it is *not applicable*.

The initialization of the acceptor will look as follows (we will add things later on but this is ok for now).

```
-module(acceptor).

-export([start/1]).

start(Name) ->
    spawn(fun() -> init(Name) end).

init(Name) ->
    Promise = order:null(),
    Voted = order:null(),
    Accepted = na,
    acceptor(Name, Promise, Voted, Accepted).
```

The acceptor is a process that is waiting for two types of messages: a prepare requests and accept requests. A prepare request, {`prepare, Proposer, Round`} will result in a promise, if we have not made any promises that prevents us to make such a promise. The round number of the prepare request must be compared with the `Promise` already given. If the round number is higher we return a promise, {`promise, Round, Voted, Accepted`}. It is of course very important that we in this message return the current accepted value and in which round we voted for this value.

If we can not give a promise we do not have to do anything but it could be polite to send an `sorry` message. If we really want to make life hard for the proposer we could even send back a promise. If we have promised not to vote in round lower than round 17, we could of course promise not to vote in a round lower than 12. The proposer will of course take our promise as an indication that it is possible for us to vote for a value in round 12 but that will of course not happen. To help the proposer we should inform it that we have have promised not to vote in the round requested by the proposer (we could even inform the proposer what we have promised but let's keep thing simple).

```
    {prepare, Proposer, Round} ->
        case order:gr(..., ...) of
            true ->
```

```
                ... ! {promise, ..., ..., ...},
                acceptor(Name, ..., Voted, Accepted);
            false ->
                ... ! {sorry, ...},
                acceptor(Name, ..., Voted, Accepted)
        end;
```

The accept request, sent by a proposer when it has received accept messages from a majority, also have two outcomes; either we can accept the request and then cast our vote in the ballot or we have a promise that prevents us from accepting the request. Note that we do not change our promise just because we vote for a new value.

Again, if we cannot accept the request we could simply ignore the message but it is polite to inform the proposer.

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

Nothing prevents an acceptor to accept a value in round 17 and then accept another value if ask to do so in round 12 (provided of course that it has not promised not to do so). This is a very strange situation but it is allowed. If we accept a value in a lower round we should of course still remember the value of the highest ballot number.

We also include a message to terminate the acceptor. You can also add messages for status information, a catch all clause etc. Also add print out statements so that you can track what the acceptor has done.

```
stop ->
   ok;
```

1.3 the proposer

The proposer work in rounds, in each round it will try to get acceptance of a proposed value or at least make the acceptors agree on any value. If this does not work it will try again and again but each time with a higher round number.

```
-module(proposer).

-export([start/4]).

-define(timeout, 2000).
-define(backoff, 10).
-define(delay, 20).

start(Name, Proposal, Acceptors, Seed) ->
    spawn(fun() -> init(Name, Proposal, Acceptors, Seed) end).

init(Name, Proposal, Acceptors, Seed) ->
    random:seed(Seed, Seed, Seed),
    Round = order:null(Name),
    round(Name, ?backoff, Round, Proposal, Acceptors).
```

In a round the proposer will wait for accept and vote messages for up to *timeout* milliseconds. If it has not received the necessary number of replies it will abort the round. It will then back-off an increasing number of milliseconds before starting the next round. It will try its best to get the acceptors to vote for a proposal but as you will see it will be happy if they can agree on anything. The `delay` will be used to introduce a slight delay in the system to make simulations more interesting.

Each round consist of one ballot attempt. The ballot either succeeds or aborts, in which case a new round is initiated.

```
round(Name, Backoff, Round, Proposal, Acceptors) ->
    case ballot(..., ..., ...) of
        {ok, Decision} ->
            io:format("~w decided ~w in round ~w~n", [..., ..., ...]),
            {ok, ...};
        abort ->
            timer:sleep(random:uniform(...)),
            Next = order:inc(...),
            round(..., (2*...), ..., ..., ...)
    end.
```

A ballot is initialized by multi-casting a prepare message to all acceptors. The process then collects all promises and also the accepted value with the highest sequence number so far. If we receive promises from a quorum (a majority) we start the voting process by multi-casting an accept message to all acceptors in the quorum. In the accept message we include the value with the highest sequence number accepted by a member on the quorum.

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

The collect procedure will simply receive promises and, if no acceptor has any objections, learn the so far accepted value with the highest ballot number. Note that we need a time out since acceptors could take forever or simply refuse to reply. Also note that we have tagged the sent request with the sequence number an only accept replies with the same sequence number, also that we need a catch all alternative since there might be delayed messages out there that otherwise would just stack up.

```
collect(0, _, _, Proposal) ->
    ...;
collect(N, Round, Max, Proposal) ->
    receive
        {promise, Round, _, na} ->
            collect(..., ..., ..., ...);
        {promise, Round, Voted, Value} ->
            case order:gr(..., ...) of
                true ->
                    collect(..., ..., ..., ...);
                false ->
                    collect(..., ..., ..., ...)
            end;
        {promise, _, _,  _} ->
            collect(..., ..., ..., ...);
        {sorry, Round} ->
```

```
            collect(..., ..., ..., ...),
        {sorry, _} ->
            collect(..., ..., ..., ...)
    after ?timeout ->
            abort
    end.
```

Collecting votes is almost the same procedure. We are only waiting for votes and need only count them until we have received them all. If we're unsuccessful we abort and hope for better luck next round.

```
vote(0, _) ->
    ...;
vote(N, Round) ->
    receive
        {vote, Round} ->
            vote(..., ...);
        {vote, _} ->
            vote(..., ...);
        {sorry, Round} ->
            vote(.., ...),
        {sorry, _} ->
            vote(..., ...)
    after ?timeout ->
            abort
    end.
```

The only things that is left is to implement the sending of requests. The prepare request will send the name of the acceptor as part of the message. This name is returned by the acceptor and can then be collected to identify the acceptors in the quorum.

```
prepare(Round, Acceptors) ->
    Fun = fun(Acceptor) -> send(Acceptor, {prepare, self(), Round}) end,
    lists:map(Fun, Acceptors).

accept(Round, Proposal, Acceptors) ->
    Fun = fun(Acceptor) -> send(Acceptor, {accept, self(), Round, Proposal}) end,
    lists:map(Fun, Acceptors).
```

Sending a message is of course trivial but we will, for reasons described later, implement it in a separate procedure.

```
send(Name, Message) ->
    Name ! Message.
```

2 Experiment

Let's set up a test and see if a set of acceptors can agree on something. We start five acceptors and have three proposers. The proposers try to make the acceptors vote for their suggestion. The proposers will hopefully find a quorum and then learn the agreed value. A test module will help us set up the experiments.

```
start(Seed) ->
    register(a, acceptor:start(a)),
    register(b, acceptor:start(b)),
    register(c, acceptor:start(c)),
    register(d, acceptor:start(d)),
    register(e, acceptor:start(e)),
    Acceptors = [a,b,c,d,e],
    proposer:start(kurtz, green, Acceptors, Seed+1),
    proposer:start(willard, red, Acceptors, Seed+2),
    proposer:start(kilgore, blue, Acceptors, Seed+3),
    true.
```

Since the acceptors stay alive even if a decision has been made we need to terminate them explicitly. The code below becomes useful during debugging since a crashed acceptor will be de-registered (and sending a message to an unregistered name will cause an exception).

```
stop() ->
    stop(a),
    stop(b),
    stop(c),
    stop(d),
    stop(e).

stop(Name) ->
    case whereis(Name) of
        undefined ->
            ok;
        Pid ->
            Pid ! stop
    end.
```

Add code to trace each state transition in the acceptor and proposer. Try to follow the execution and the progress of the algorithm.

Do some experiments and try to introduce delays in the acceptor. Insert larger delays and see if the algorithm still terminates.

Could you even come to an agreement when you ignore messages? Try ignoring to send sorry messages or simply randomly drop a vote. If you drop too many messages a quorum will of course never be found but we could probably loose quite many. Does the algorithm ever report conflicting answers?

What happens of we increase the number of acceptors to say 9 or 17? Will we reach a decision? What if we have also have 10 proposers?

3 Fault tolerant

In order to make the implementation fault tolerant we need to remember what we promise and what we vote for. If we use the module pers given in the appendix we can initialize our state to the state we had when we crashed and store state changes as we make promises. Where in the acceptor should we add this?

We also have to be careful when we send a message to an acceptor. We should first check that the acceptor is actually registered, if not it means that the acceptor is down. If we knew that the acceptor was registered on a remote node we could ignore this procedure since sending a message to a remote process always succeeds. If the acceptor is a locally registered process the send operation could throw an exception, something that we want to avoid.

```
send(Name, Message) ->
    case whereis(Name) of
        undefined ->
            down;
        Pid ->
            Pid ! Message,
            timer:sleep(random:uniform(?delay))
    end.
```

Simulate a crash and restart using the procedure below (in the test module and see if the protocol still comes to a consensus. You might have to increase the sleep period in the acceptor to make the execution run slower.

```
crash(Name) ->
    case whereis(Name) of
        undefined ->
            ok;
        Pid ->
            unregister(Name),
            exit(Pid, "crash"),
            register(Name, acceptor:start(Name))
    end.
```

4 Carrying on

There are some improvements that could be made in the implementation of the proposer. If we need three promises for a quorum and we have received three sorry messages from the in total five acceptors then we can abort the ballot. Change the code of of the collect/4 and vote/2 procedures to also keep track of how many messages in total there are still out there.

As you have probably noticed the decision process is fast if we only have one active proposer. Can we have an election in the beginning and decide who is to be the active proposer?

In the above implementation there is only one decision being made. A more practical system would of course have a series of decisions to make. Could we implement an acceptor that is willing to accept sequences of values.

A proposer would then be sent a value that it should try to add to the sequence. I t will of course still play by the rules and accept that other values could be added before its own value.

Appendix: order

```
-module(order).

-export([null/0, null/1, gr/2, goe/2, inc/1]).

null() ->
    {0,0}.

null(Id) ->
    {0, Id}.

gr({N1,I1}, {N2,I2}) ->
    if
        N1 > N2 ->
            true;
        ((N1 == N2) and (I1 > I2)) ->
            true;
        true ->
            false
    end.

goe({N1,I1}, {N2,I2}) ->
    if
        N1 > N2 ->
            true;
        ((N1 == N2) and  (I1 >= I2)) ->
            true;
        true ->
            false
    end.

inc({N, Id}) ->
    {N+1, Id}.
```

Appendix: pers

```
-module(pers).

-export([read/1, store/4, delete/1]).

read(Id) ->
    {ok, Id} = dets:open_file(Id, []),
    case dets:lookup(Id, perm) of
        [{perm, Bn, An, Av}] ->
            {Bn, An, Av};
        [] ->
            {order:null(), order:null(), na}
    end.

store(Id, Bn, An, Av)->
    dets:insert(Id, {perm, Bn, An, Av}).

delete(Id) ->
    dets:delete(Id, perm),
    dets:close(Id).
```