Seminar Report: Muty

Pedro Garcia Mota, Pau Espin Pedrol

March 21, 2013

1 Introduction

Introduce in a couple of sentences the seminar and the main topic related to distributed systems it covers.

Implementation of a distributed mutual-exclusion lock in an asyncronous network using a multicast strategy. Implement and use a distributed logical clock system.

2 Work done

Present the new code that you have added or how you have implemented a required functionality by using small Erlang code snippets (you do not need to copy&paste all the code).

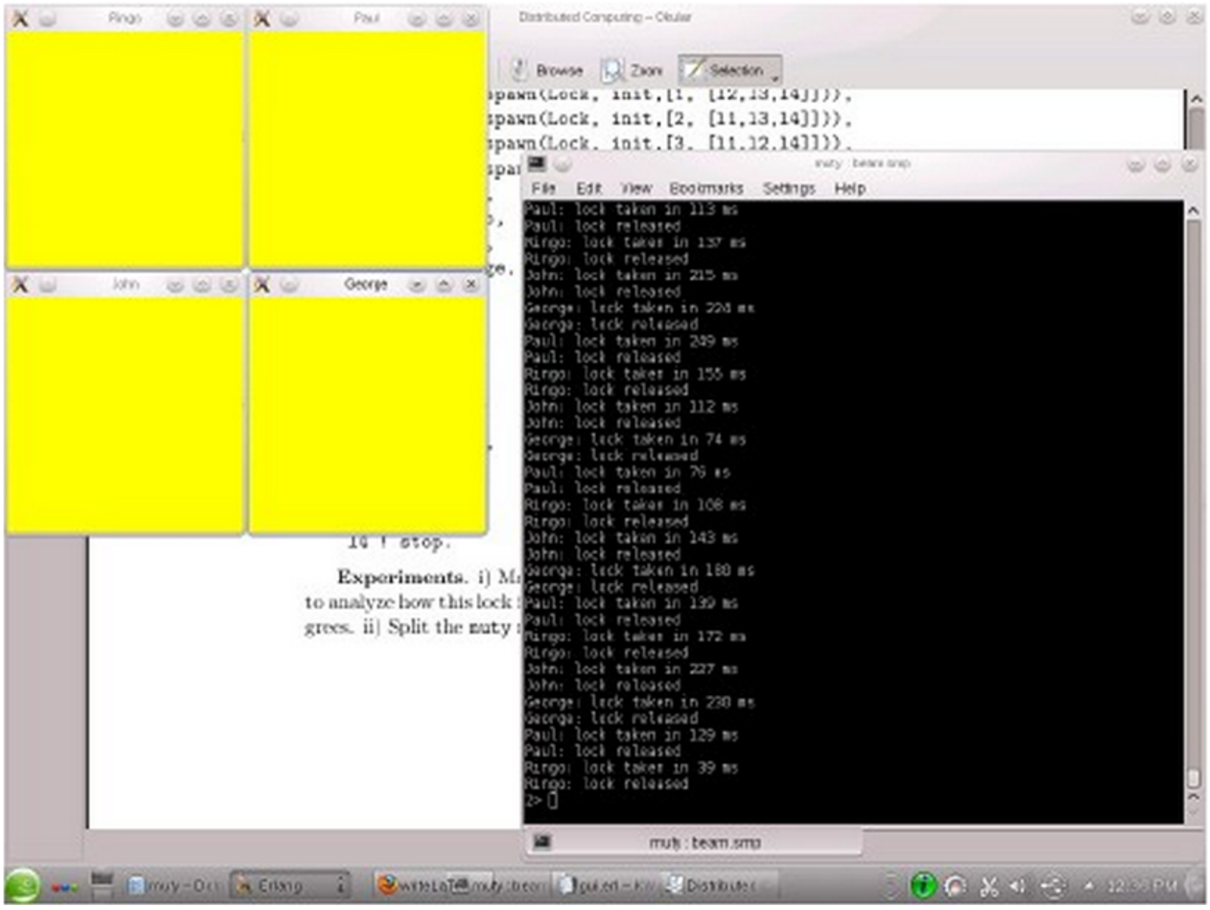
```
———lock2.erl———

wait(Id, Nodes, Master, Refs, Waiting) ->
    receive
        {request, From, Ref, IdOther} ->
            if
                IdOther < Id ->
                    From ! {ok, Ref},
                    R = make_ref(),
                    From ! {request, self(), R, Id},
                    wait(Id, Nodes, Master, [R | Refs], Waiting);
                true ->
                    wait(Id, Nodes, Master, Refs, [{From, Ref}|
                        Waiting])
            end;
        {ok, Ref} ->
            Refs2 = lists:delete(Ref, Refs),
            wait(Id, Nodes, Master, Refs2, Waiting);
        release ->
            ok(Waiting),
            open(Id, Nodes)
    end.


held(Id, Nodes, Waiting) ->
```

```
    receive
        {request, From, Ref, _} ->
            held(Id, Nodes, [{From, Ref}|Waiting]);
        release ->
            ok(Waiting),
            open(Id, Nodes)
    end.


----------lock 3.erl----------

wait(Id, Nodes, Master, Refs, Waiting, Clock, MyTimestamp) ->
    receive
        {request, From, Ref, IdOther, Timestamp} ->
      if
        Timestamp < MyTimestamp ->
          Newclock = Clock + 1,
          From ! {ok, Ref, Newclock},
          R = make_ref(),
          From ! {request, self(), R, Id, MyTimestamp},
          wait(Id, Nodes, Master, [R | Refs], Waiting, Newclock,
              MyTimestamp);

        Timestamp > MyTimestamp ->
          Newclock = Timestamp + 1,
          wait(Id, Nodes, Master, Refs, [{From, Ref}|Waiting],
              Newclock, MyTimestamp);

        Timestamp == MyTimestamp ->
          Newclock = Clock + 1,
          if
            IdOther < Id ->
              From ! {ok, Ref, Newclock},
              R = make_ref(),
              From ! {request, self(), R, Id, Newclock, MyTimestamp}
                  ,
              wait(Id, Nodes, Master, [R | Refs], Waiting, Newclock,
                  MyTimestamp);
            true ->
              wait(Id, Nodes, Master, Refs, [{From, Ref}|Waiting],
                  Newclock, MyTimestamp)
          end
      end;
        {ok, Ref, Timestamp} ->
      Refs2 = lists:delete(Ref, Refs),
            wait(Id, Nodes, Master, Refs2, Waiting, max(Clock,
                Timestamp) + 1, MyTimestamp);
        release ->
      ok(Waiting, Clock),
            open(Id, Nodes, Clock)
    end.

held(Id, Nodes, Waiting, Clock) ->
receive
    {request, From, Ref, _, Timestamp} ->
        held(Id, Nodes, [{From, Ref}|Waiting], max(Clock,Timestamp)
           + 1);
    release ->
        ok(Waiting, Clock),
        open(Id, Nodes, Clock)
end.
```

3 Experiments

Make tests with different Sleep and Work parameters to analyze how this lock implementation responds to different contention degrees.

If we set small Sleep times and bigger Work times (ie. 10 and 100 respectively), we can get a deadlock in few seconds, as there's a high degree of contention and with the lock1 algorithm it is easy to become deadlocked if more than one process is waiting for the lock (sent a request message), as each process is waiting for the other to finish (send the ok message).



```
spawn(Lock, init,[1, [12,13,14]]),
spawn(Lock, init,[2, [11,13,14]]),
spawn(Lock, init,[3, [11,12,14]]),
```

```
Paul: lock taken in 113 ms
Paul: lock released
Ringo: lock taken in 137 ms
Ringo: lock released
John: lock taken in 215 ms
John: lock released
George: lock taken in 224 ms
George: lock released
Paul: lock taken in 249 ms
Paul: lock released
Ringo: lock taken in 155 ms
Ringo: lock released
John: lock taken in 112 ms
John: lock released
George: lock taken in 74 ms
George: lock released
Paul: lock taken in 76 ms
Paul: lock released
Ringo: lock taken in 108 ms
Ringo: lock released
John: lock taken in 143 ms
John: lock released
George: lock taken in 180 ms
George: lock released
Paul: lock taken in 139 ms
Paul: lock released
Ringo: lock taken in 172 ms
Ringo: lock released
John: lock taken in 227 ms
John: lock released
George: lock taken in 230 ms
George: lock released
Paul: lock taken in 129 ms
Paul: lock released
Ringo: lock taken in 39 ms
Ringo: lock released
2> 
```

Split the muty module and make the needed adaptations to enable each worker-lock pair to run in different machines. Remember how names registered in remote nodes are referred and how Erlang runtime should be started to run distributed programs.

```
-export([start1/3, start2/3, start3/3, start4/3, stop/0]).

start1(Lock, Sleep, Work) ->
    register(l1, spawn(Lock, init, [1, [l2,l3,l4]])),
    register(john,   spawn(worker, init, ["John",  l1,34,Sleep,Work])),
    ok.

start2(Lock, Sleep, Work) ->
    register(l2, spawn(Lock, init,[2, [l1,l3,l4]])),
    register(ringo,  spawn(worker, init, ["Ringo", l2,37,Sleep,Work])),
    ok.

start3(Lock, Sleep, Work) ->
    register(l3, spawn(Lock, init,[3, [l1,l2,l4]])),
    register(paul,   spawn(worker, init, ["Paul",  l3,43,Sleep,Work])),
    ok.

start4(Lock, Sleep, Work) ->
    register(l4, spawn(Lock, init,[4, [l1,l2,l3]])),
    register(george, spawn(worker, init, ["George",l4,72,Sleep,Work])),
    ok.
```

Repeat the previous tests to compare the behavior of this lock with respect to the previous one.

The Lamport implementation does not suffer from deadlock or starvation problems like the lock2 implementation. It is more equitative with the acquisitors (althrough not completelly equal, as it still uses ID's to give priorities.)

Repeat the previous tests to compare this version with the former ones.

```
13> muty:stop().
stop
George: lock released
George: 12 locks taken, 190.41666666666666 ms (avg) for taking, 0 withdrawals
John: lock taken in 262 ms
John: lock released
John: 12 locks taken, 228.08333333333334 ms (avg) for taking, 0 withdrawals
Ringo: lock taken in 138 ms
Ringo: lock released
Ringo: 12 locks taken, 236.41666666666666 ms (avg) for taking, 0 withdrawals
Paul: lock taken in 232 ms
Paul: lock released
Paul: 12 locks taken, 243.75 ms (avg) for taking, 0 withdrawals
```

4 Open questions

- *What is happening when you increase the risk of a lock conflict?*

  A deadlock appears.

- *Justify how you guarantee that only one process is in the critical section at any time.*

  When a process receives a request and it has lower priority than the sender's priority the receiver responds with an Ok to the sender and makes a new request to all the processes to enter the critical region.

- *What is the main drawback of lock2 implementation?*

  Some process may suffer from starvation. It neither offers equalness among different processes.

- *Note that the workers are not involved in the Lamport clock. Could we have a situation where a worker is not given the priority to a lock even if it issued a request to its instance before (assuming a reference global time) the worker that took the lock?*

  Yes, it could happen, as the global time is not the same as the time given by the logic clock. The logic clock permits the system to order events one before/after the other, but the order doesn't not have to be the same as if we could order them using a global time.

5 Personal Opinion

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

This seminar is really useful to understand how mutual exclusion and logic clocks are implemented and used in distributed systems. It also helps on getting used to reading, understanding and writing erlang code.

Despite we had some problems with Wx libraries on last erlang revisions on our own computers (and so we could only work nice on the lab), it's been an interesting seminar to work on.