Muty: a distributed mutual-exclusion lock

Johan Montelius

October 2, 2016

Introduction

Your task is to implement a distributed mutual-exclusion lock. The lock will use a multicast strategy and work in a asynchronous network where we do not have access to a synchronized clock. You will do the implementation in three versions: the dead-lock prone, the unfair and the Lamport clocked. Before you start you should have good theoretical knowledge of the multicast algorithm and how Lamport clocks work.

1 The architecture

The scenario is that a set of workers need to synchronize and, they will randomly decide to take a lock and when taken hold it for a short period before releasing it. Each worker will collect statistics on how long time it took them to acquire the lock so that it can present some interesting figures at the end of each test.

Let's first implement the worker and then do refinement of the lock.

1.1 the worker

When the worker is started it is given access to a lock. It is also given a name for nicer print-out and a seed so that each worker will have its own random sequence. We also provide information on for how long the worker in average is going to sleep and work.

We will have four workers competing for a lock so if they sleep for in average 1000 ms and work for in average 2000 ms we will have a lock with high chance of congestion. You can easily change these parameters to simulate more or less congestion. The deadlock constant is how long (4000 ms) we are going to wait for a lock before giving up.

The gui is a process that will give you some feedback on the screen on what the worker is actually doing. The gui could simply log things on the terminal but a graphical interface is of course to be prefered. An example gui, based on the the wx library, is given in the appendix.

```
-module(worker).

-export([start/5]).
```

```
-define(deadlock, 4000).

start(Name, Lock, Seed, Sleep, Work) ->
    spawn(fun() -> init(Name, Lock, Seed, Sleep, Work) end).

init(Name, Lock, Seed, Sleep, Work) ->
    Gui = spawn(gui, init, [Name]),
    random:seed(Seed, Seed, Seed),
    Taken = worker(Name, Lock, [], Sleep, Work, Gui),
    Gui ! stop,
    terminate(Name, Taken).
```

We will do some book-keeping and save the time it took to get the locks. In the end we will print some statistics.

A worker sleeps for a while and then decides to move into the critical section. The call to `critical/4` will return information on if the critical section was entered and how long it took to acquire the lock.

```
worker(Name, Lock, Taken, Sleep, Work, Gui) ->
    Wait = random:uniform(Sleep),
    receive
       stop ->
            Taken
    after Wait ->
            T = critical(Name, Lock, Work, Gui),
            worker(Name, Lock, [T|Taken], Sleep, Work, Gui)
    end.
```

The critical section is entered by requesting the lock. We wait for a reply `taken` or for a time-out. If the lock is taken the elapsed time `T` is returned to the caller.

The gui is informed as we send the request for the lock and if we acquire the lock or have to abort.

```
critical(Name, Lock, Work, Gui) ->
  T1 = erlang:system_time(micro_seconds),
  Gui ! waiting,
  Lock ! {take, self()},
  receive
      taken ->
          T2 = erlang:system_time(micro_seconds),
          T = T2 - T1,
          io:format("~w: lock taken in ~w ms~n",[Name, T div 1000]),
          Gui ! taken,
          timer:sleep(random:uniform(Work)),
          Gui ! leave,
          Lock ! release,
          {taken, T}
  after ?deadlock ->
          io:format("~w: giving up~n",[Name]),
          Lock ! release,
          Gui ! leave,
          no
  end.
```

The worker terminates when it receives a `stop` message. It will simply print out some statistics.

```
terminate(Name, Taken) ->
    {Locks, Time, Dead} =
       lists:foldl(
          fun(Entry,{L,T,D}) ->
            case Entry of
               {taken,I} ->
                   {L+1,T+I,D};
                _ ->
                   {L,T,D+1}
            end
          end,
          {0,0,0}, Taken),
    if
       Locks > 0 ->
           Average = Time / Locks;
       true ->
           Average = 0
    end,
    io:format("~s: ~w locks taken, average of ~w ms, ~w deadlock situations~n",
              [Name, Locks, (Average div 1000), Dead]).
```

1.2 the locks

You will now work with three locks implemented in three modules: `lock1`, `lock2` and `lock3`. The first lock, `lock1`, will be very simple and will not fulfill the requirements that we have on a lock. It will prevent several workers from entering the critical section but that is about it.

When the lock is started it is given a unique identifier and a set of peer locks. The identifier is not used by the first lock but we want the interface to be the same for all lock.

```
-module(lock1).

-export([start/2]).

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

init(_) ->
    receive
        {peers, Peers} ->
            open(Peers);
        stop ->
            ok
    end.
```

The lock enters the state `open` and waits for either a command to `take` the lock or a `request` from another lock. If it is ordered to take the lock it will multicast a request to all other locks and then enter a waiting state. A request from another lock is immediately replied with an `ok` message. Note how the reference is used to connect the request to the reply.

```
open(Nodes) ->
    receive
        {take, Master} ->
            Refs = requests(Nodes),
            wait(Nodes, Master, Refs, []);
        {request, From,  Ref} ->
            From ! {ok, Ref},
            open(Nodes);
        stop ->
            ok
    end.

requests(Nodes) ->
    lists:map(fun(P) -> R = make_ref(), P ! {request, self(), R}, R end, Nodes).
```

In the waiting state the lock is waiting for `ok` messages. All requests have been tagged with unique references so that it can keep track of which locks that have replied and which that it is still waiting for. We could have made simpler solution where we simple wait for n locks to reply but this version is more flexible if we want to extend it.

```
wait(Nodes, Master, [], Waiting) ->
    Master ! taken,
    held(Nodes, Waiting);
wait(Nodes, Master, Refs, Waiting) ->
    receive
        {request, From, Ref} ->
            wait(Nodes, Master, Refs, [{From, Ref}|Waiting]);
        {ok, Ref} ->
            Refs2 = lists:delete(Ref, Refs),
            wait(Nodes, Master, Refs2, Waiting);
        release ->
            ok(Waiting),
            open(Nodes)
    end.

ok(Waiting) ->
    lists:foreach(fun({F,R}) -> F ! {ok, R} end, Waiting).
```

While the lock is waiting it could also receive `request` messages from locks that have also decided to take the lock. In this version of the lock we simply add these to a set of locks that have to wait. When the lock is released we will send them `ok` messages.

As an escape from dead-lock, we also allow the worker to send a `release` message even though the lock is not yet held. We will then send `ok` messages to all waiting locks and enter the `open` state.

In the `held` state we keep adding requests from locks to the list of waiting locks until we receive a `release` message from the worker.

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

For the Erlang hacker there are some things to think about. In Erlang messages are queued in the mail-box of the processes. If they do not match a pattern in a receive statement they are handled but otherwise they are kept in the queue. In our implementation we happily accept and handle all messages even though some, such as the `request` messages when in the `held` state, are just stored for later. Would it be possible to use the Erlang message queue instead and let request messages be queued until we release the lock? The reason why I've implemented it they way I did was that I wanted to make it explicit that request messages are treated even if we're in the `held` state. - Why are we not looking for `ok` messages?

1.3 some testing

Now write some test procedures that create four locks and four workers. Connect them and run some tests.

```
-module(muty).

-export([start/3, stop/0]).


start(Lock, Sleep, Work) ->
L1 = apply(Lock, start, [1]),
L2 = apply(Lock, start, [2]),
L3 = apply(Lock, start, [3]),
L4 = apply(Lock, start, [4]),
        L1 ! {peers, [L2, L3, L4]},
        L2 ! {peers, [L1, L3, L4]},
        L3 ! {peers, [L1, L2, L4]},
        L4 ! {peers, [L1, L2, L3]},
register(w1, worker:start("John", L1,  34, Sleep, Work)),
register(w2, worker:start("Ringo", L2, 37, Sleep, Work)),
register(w3, worker:start("Paul", L3, 43, Sleep, Work)),
register(w4, worker:start("George", L4, 72, Sleep, Work)),
ok.

stop() ->
    stop(w1), stop(w2), stop(w3), stop(w4).

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

We're now using the name of the module as a parameter to the start procedure. We will easily be able to test different locks with different sleep and work parameters. Does it work ok? What is happening when you increase the risk of a lock conflict? Why?

2 Resolving dead-lock

The problem with the first solution can be handled if we give each lock a unique identifier 1, 2, 3 and 4. The identifier will give a priority to the lock.

A lock in the waiting state will send a `ok` message to a requesting lock if the requesting lock has a higher priority (`1` having highest priority).

Implement this solution in a module called `lock2`, and show that it works even if we have high contention. Does it work? There is a situation that you have to handle correctly. If not, you run the danger of having two processes in the critical section at the same time. Can you guarantee that we only have one process in the critical section at any time? At the seminar be prepared to explain why your solution does work.

Run some tests and see how well your solution works. How well does it perform and what is the drawback? At the seminar be prepared to show your results.

3 Lamport time

One improvement is to let locks be taken with priority given in time order. The only problem is that we do not (assuming we are running over a asynchronous network) have access to synchronized clocks. The solution is to use logical clocks such as Lamport clocks.

To implement this you must add a time variable to the lock. The value is initialized to zero but is updated every time the lock receives a `request` message from another lock. The Lamport clock thus keeps track of the highest request we have seen so far. When a request is sent, it should have a timestamp of one higher than what we have seen.

You now see a solution where the Lamport timestamp need not be added to all message in the system but only the ones that are important i.e. the request messages.

When a lock is in the waiting state it must determine if the request was sent before or after it sent it's own request message. If this can not be determined the lock identifier is used to resolve the order.

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 though it issued a request to it's lock logically before the worker that took the lock?

4 The seminar

At the start of the seminar you should hand in a two page report on how you solved the task of handling the dead-lock in lock2. You should also describe how you think the priority on Lamport time could be implemented. At the seminar you should be able to present and explain your solution to the above problems. You should also be able to discuss the pros and cons with each solution. During the seminar we will also implement the solution in a module called `lock3` and run some experiments.

Appendix

Here is a gui. The worker will start the gui and send messages when it is waiting for a lock, when it receives a lock and when the lock is released (or attempt to take the lock is aborted). The window of the gui will be: blue for open, yellow for waiting and red for held.

```
-module(gui).
-export([start/1, init/1]).
-include_lib("wx/include/wx.hrl").

start(Name) ->
        spawn(gui, init, [Name]).

init(Name) ->
        Width = 200,
        Height = 200,
        Server = wx:new(),  %Server will be the parent for the Frame
        Frame = wxFrame:new(Server, -1, Name, [{size,{Width, Height}}]),
        wxFrame:show(Frame),
        loop(Frame).


loop(Frame)->
        receive
                waiting ->
                        wxFrame:setBackgroundColour(Frame, {255, 255, 0}),
                        loop(Frame);
                enter ->
                        wxFrame:setBackgroundColour(Frame, ?wxRED),
                        loop(Frame);
                leave ->
                        wxFrame:setBackgroundColour(Frame, ?wxBLUE),
                        loop(Frame);
                stop ->
                        ok;
                Error ->
                        io:format("gui: strange message ~w ~n", [Error]),
                        loop(Frame)
        end.
```