Muty: a distributed mutual-exclusion lock

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

February 6, 2013

Introduction

Your task is to implement a distributed mutual-exclusion lock. The lock will use a multicast strategy and work in an asynchronous network where we do not have access to a synchronized clock. You will do the implementation in three versions: the deadlock prone, the unfair, and the Lamport clocked. Before you start you should have good theoretical knowledge of the totally ordered 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. The lock is **distributed**, and each worker will operate with a given instance of the lock. Each worker will collect statistics on how long 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 its lock instance. 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 before trying to get the lock and work with the lock taken.

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 withdrawal constant is how long (8000 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 complete code of the gui is given in the appendix.

```
-module(worker).
-export([init/5]).
-define(withdrawal, 8000).

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 (if worker has not been stopped while sleeping). The call to `critical/4` will return information on if the critical section was entered and how long it took to acquire the lock. For each invocation of the `critical` function, the worker stores this information in the `Taken` list.

```
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 to the worker's lock instance. Notice that locks instances are implemented as independent processes. We wait for a reply `taken` or for a deadlock timeout. Note that we can get a timeout when we are really in a deadlock, or simply when the lock instance is taking too long to respond. The `elapsed/1` function should calculate the time in milliseconds from the time `T1`. The elapsed time `T` is then 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 =  now(),
  Gui ! waiting,
  Lock ! {take, self()},
  receive
```

```
      taken ->
          T = elapsed(T1),
          io:format("~s: lock taken in ~w ms~n",[Name, T]),
          Gui ! taken,
          timer:sleep(random:uniform(Work)),
          io:format("~s: lock released~n",[Name]),
          Gui ! leave,
          Lock ! release,
          {taken, T}
  after ?withdrawal ->
          io:format("~s: giving up~n",[Name]),
          Lock ! release,
          Gui ! leave,
          no
  end.

elapsed({_,S1,M1}) ->
    {_,S2,M2} = now(),
    (S2 - S1)*1000 + ((M2 - M1) div 1000).
```

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, ~w ms (avg) for taking, ~w withdrawals~n",
              [Name, Locks, Average, Dead]).
```

1.2 The locks

We will work with three versions of the lock 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 all about it.

When each lock instance is started, it is given a unique identifier and a list of peer-lock processes (i.e. the other lock instances). The identifier will not be used in the lock1 implementation, but we keep it there to make the interface to all locks the same.

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

```
-module(lock1).
-export([init/2]).

init(_, Nodes) ->
    open(Nodes).

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 instance is waiting for ok messages. All requests have been tagged with unique references (using `make_ref()` BIF) so that the lock instance can keep track of which lock instances have replied and which it is still waiting for (`Refs`). There is a simpler solution where we simply wait for n locks to reply, but this version is more flexible if we want to extend it.

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

As an escape from deadlock, 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 lock instances and enter the `open` state.

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

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

For the Erlang hacker there are some things to think about. In Erlang, messages are queued in the mailbox of the processes. If they do 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? Yes! The reason for not doing so was to make it explicit that `request` messages are treated even if we are in the `held` state.

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

1.3 Some testing

Next test procedure creates four locks instances and four workers. Note that we are using the name of the module (i.e. `lock1`) as a parameter to the start procedure. We will easily be able to test different locks.

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

start(Lock, Sleep, Work) ->
    register(l1, spawn(Lock, init,[1, [l2,l3,l4]])),
    register(l2, spawn(Lock, init,[2, [l1,l3,l4]])),
    register(l3, spawn(Lock, init,[3, [l1,l2,l4]])),
    register(l4, spawn(Lock, init,[4, [l1,l2,l3]])),
    register(john,   spawn(worker, init, ["John",  l1,34,Sleep,Work])),
    register(ringo,  spawn(worker, init, ["Ringo", l2,37,Sleep,Work])),
    register(paul,   spawn(worker, init, ["Paul",  l3,43,Sleep,Work])),
    register(george, spawn(worker, init, ["George",l4,72,Sleep,Work])),
    ok.

stop() ->
    john ! stop,
    ringo ! stop,
    paul ! stop,
    george ! stop,
    l1 ! stop,
    l2 ! stop,
    l3 ! stop,
    l4 ! stop.
```

Experiments. i) Make tests with different `Sleep` and `Work` parameters to analyze how this lock implementation responds to different contention degrees. ii) 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.

Open Questions. What is happening when you increase the risk of a lock conflict?

2 Resolving deadlock

The problem with the first solution can be handled if we give each lock instance a unique identifier `1`, `2`, `3` and `4`. The identifier will give a priority to the lock instance. A lock instance in the waiting state will send an `ok` message to a requesting lock instance if the requesting lock instance has a higher priority (`1` having the highest priority).

Implement this solution in a module called `lock2`, and show that it works even if we have high contention. There is a situation that you have to be careful with (i.e. a process wants to access the lock and it has already acknowledged another process with lower priority that it is still gathering `ok` messages). If you do not handle correctly this situation, you run the danger of having two processes in the critical section at the same time.

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

Open Questions. i) Justify how you guarantee that only one process is in the critical section at any time. ii) What is the main drawback of `lock2` implementation?

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 an 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 clock variable to the lock instance, which keeps track of the instance logical time. The value is initialized to zero but is updated every time the lock instance receives a `request` or `ok` message from another lock instance to one plus the greater of the own clock and the timestamp received in the message.

When a lock instance is in the waiting state and receives a `request` it must determine if this request was sent before or after it sent its own request message. To do this, each request will have an associated timestamp. Request timestamps have to be compared to determine which was raised first. If this cannot be determined (timestamps are equal), the lock instance identifier is used to resolve the order. Implement the solution in a module called `lock3`.

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

Open Questions. 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?

Appendix

Here is the gui. The worker will start the gui and send messages when it is waiting for a lock (the window of the gui will be YELLOW), when it takes the lock (the gui will be RED), and when the lock is released (or attempt to take the lock is aborted) (the gui will be BLUE).

```
-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 ->
            %wxYELLOW doesn't exist in "wx/include/wx.hrl"
            wxFrame:setBackgroundColour(Frame, {255, 255, 0}),
            wxFrame:refresh(Frame),
            loop(Frame);
        taken ->
            wxFrame:setBackgroundColour(Frame, ?wxRED),
            wxFrame:refresh(Frame),
            loop(Frame);
        leave ->
            wxFrame:setBackgroundColour(Frame, ?wxBLUE),
            wxFrame:refresh(Frame),
            loop(Frame);
        stop ->
            ok;
        Error ->
            io:format("gui: strange message ~w ~n", [Error]),
            loop(Frame)
    end.
```