Muty: a distributed mutual-exclusion lock

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

February 6, 2025

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 Ricart & Agrawala's mutual exclusion algorithm and how Lamport's clocks work.

This assignment includes a number of coding and experimental milestones. On accomplishing each of them, the team must show their outcome to the professor during the practical sessions, so that he can track the progress and evaluate the team accordingly. The team can validate their coding milestones beforehand through the review questionnaires at ATENEA. This assignment also includes a number of open questions. The team must answer each of them through the questionnaire at ATENEA before the deadline (one week after the last session of the seminar).

1 The architecture

The scenario is that a set of workers need to synchronize. 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 they can present some interesting figures at the end of each test.

1.1 The worker

When the worker is started, it is given its lock instance and a name for nicer print-out. We also provide information on for up to how long the worker 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 up to 1000 ms and work for up to 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 code of the gui is given in the appendix.

```
-module(worker).
-export([start/6]).
-define(withdrawal, 8000).

start(Name, Main, Module, Id, Sleep, Work) ->
    spawn(fun() -> init(Name, Main, Module, Id, Sleep, Work) end).

init(Name, Main, Module, Id, Sleep, Work) ->
    Lock = apply(Module, start, [Id]),
    Main ! {ready, Lock},
    Gui = gui:start(Name),
    Taken = worker(Name, Lock, [], Sleep, Work, Gui),
    Gui ! stop,
    Lock ! 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 `acquire/3` will return information on whether the critical section was entered and how long it took to acquire the lock. For each invocation of the `acquire` function, the worker stores this information in the `Taken` list.

```
worker(Name, Lock, Taken, Sleep, Work, Gui) ->
    Sleeptime = rand:uniform(Sleep),
    receive
        stop ->
            Taken
    after Sleeptime ->
            T = acquire(Name, Lock, Gui),
            case T of
                stopped ->
                    Taken;
                withdrawn ->
                    worker(Name, Lock, [T|Taken], Sleep, Work, Gui);
                _ ->
                    Worktime = rand:uniform(Work),
                    receive
                        stop ->
                            Gui ! leave,
                            Lock ! release,
                            Taken
                    after Worktime ->
                            io:format("~s: lock released~n", [Name]),
                            Gui ! leave,
                            Lock ! release,
                            worker(Name, Lock, [T|Taken], Sleep, Work, Gui)
                    end
            end
    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 withdrawal 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. We calculate the elapsed time `T` in milliseconds from the times `T1` and `T2` and return it 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.

```
acquire(Name, Lock, Gui) ->
  T1 = erlang:monotonic_time(),
  Gui ! waiting,
  Ref = make_ref(),
  Lock ! {take, self(), Ref},
  receive
      {taken, Ref} ->
          T2 = erlang:monotonic_time(),
          T = erlang:convert_time_unit(T2-T1, native, millisecond),
          io:format("~s: lock taken in ~w ms~n", [Name, T]),
          Gui ! taken,
          {taken, T};
      stop ->
          Gui ! leave,
          Lock ! release,
          stopped
  after ?withdrawal ->
          io:format("~s: giving up~n", [Name]),
          Gui ! leave,
          Lock ! release,
          withdrawn
  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, ~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 (`Ref`) is used to connect the request to the reply.

```
-module(lock1).
-export([start/1]).

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

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

open(Nodes) ->
    receive
        {take, Master, Ref} ->
            Refs = requests(Nodes),
            wait(Nodes, Master, Refs, [], Ref);
        {request, From,  Ref} ->
            From ! {ok, Ref},
            open(Nodes);
        stop ->
            ok;
        Error ->
            io:format("open: unsupported message: ~w~n", [Error])
    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/0` Erlang 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, TakeRef) ->
    Master ! {taken, TakeRef},
    held(Nodes, Waiting);
wait(Nodes, Master, Refs, Waiting, TakeRef) ->
    receive
        {request, From, Ref} ->
            wait(Nodes, Master, Refs, [{From, Ref}|Waiting], TakeRef);
        {ok, Ref} ->
            NewRefs = lists:delete(Ref, Refs),
            wait(Nodes, Master, NewRefs, Waiting, TakeRef);
        release ->
            ok(Waiting),
            open(Nodes);
        Error ->
            io:format("wait: unsupported message: ~w~n", [Error])
    end.

ok(Waiting) ->
    lists:foreach(
      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);
        Error ->
            io:format("held: unsupported message: ~w~n", [Error])
    end.
```

1.3 Some testing

Let's do some testing by using the next procedure, which creates four lock 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. We also provide the time (in milliseconds) for up to how long the worker is going to sleep before trying to get the lock (`Sleep`) and work with the lock taken (`Work`).

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

start(Lock, Sleep, Work) ->
    Main = self(),
    register(w1, worker:start("John", Main, Lock, 1, Sleep, Work)),
    register(w2, worker:start("Ringo", Main, Lock, 2, Sleep, Work)),
    register(w3, worker:start("Paul", Main, Lock, 3, Sleep, Work)),
    register(w4, worker:start("George", Main, Lock, 4, Sleep, Work)),
    collect(4, []).

collect(N, Locks) ->
    if
        N == 0 ->
            lists:foreach(fun(L) ->
                L ! {peers, lists:delete(L, Locks)}
            end, Locks);
        true ->
            receive
                {ready, L} ->
                    collect(N-1, [L|Locks])
            end
    end.

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

Note that the test procedure runs all the processes in the same Erlang instance. To run the tests in a distributed fashion, you must adapt the `muty` module to start (and stop) each worker in a different Erlang instance. Check the slides about Erlang to refresh how processes are created remotely, how names registered in remote nodes are referred, and how Erlang runtime should be started to run distributed programs.

CODE REVIEW (1 points)
Distributed execution: Erlang file `muty.erl`.

EXPERIMENTAL MILESTONES (1 points)	
MS1 (0.5 points)	Make tests with different `Sleep` and `Work` parameters to analyze how `lock1` responds to different degrees of contention.
MS2 (0.5 points)	Try the lock system while running each worker in a different Erlang instance.

OPEN QUESTIONS (0.5 points)
a) What unwanted halting situation can occur with this lock implementation when there is high contention?   b) Indicate two situations which can lead to a process *withdrawal* (i.e., a process gets tired of waiting to be granted access to the critical section). Assume that processes do not fail.

2 Lock priorities

The problem with the first solution can be handled by using the unique identifier (`1`, `2`, `3`, and `4`) that each lock instance is given (as the `MyId` parameter). 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). Otherwise, it will add it to the set of instances that have to wait, as we were doing before.

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). You must handle correctly this situation to avoid the danger of having two workers in the critical section at the same time (hint: you can send an additional `request` message (with a new reference) when sending an `ok` message to a higher-priority requesting lock instance).

CODE REVIEW (2 points)
Lock based on process priorities: Erlang file `lock2.erl`.

EXPERIMENTAL MILESTONES (0.5 points)	
MS3 (0.5 points)	Repeat the tests on `MS1` to compare the behavior of this lock implementation with respect to the previous one.

OPEN QUESTIONS (0.75 points)
c) Justify how your code guarantees that only one worker is in the critical section at any time, especially in the tricky situation described before.
d) Do the situations described in the previous question b) still lead to a process *withdrawal* in this lock implementation?
e) What is the main drawback of this lock implementation?

3 Lamport's 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's clocks.

You must **add a clock variable to the lock instance** to keep track of its logical time. It is initialized to zero and must be increased (only) every time the lock instance requests access to the critical section (before it sends the `request` messages to the other instances). In addition, it must be updated when the instance receives a `request` message from another lock instance to the greatest of the own clock and the timestamp received in the message (you can use `max/2` Erlang BIF). Thus, the clock will keep track of the highest request we have seen so far. Note that we do not need to add the Lamport's timestamp to all the messages but only to the `request` messages.

When a lock instance is in the waiting state and receives a `request`, it must determine whether this request was sent before or after it sent its own `request` message. To do this, it needs to **keep another variable with the timestamp corresponding to its own `request` message**, which is compared with the timestamp of the incoming `request` to determine which was raised first. If timestamps are equal, the lock instance identifier is used to resolve the order. Implement this solution in a module called `lock3`.

CODE REVIEW (3 points)
Lock based on Lamport's clocks: Erlang file `lock3.erl`.

EXPERIMENTAL MILESTONES (0.5 points)	
MS4 (0.5 points)	Repeat the tests on `MS1` to compare the behavior of this lock implementation with respect to the former ones.

OPEN QUESTIONS (0.75 points)

f) Justify if this lock implementation requires sending an additional `request` message (as well as the `ok` message) when a process in the `waiting` state receives a `request` message with the same logical time from another process with higher priority.

g) Do the situations described in the previous question b) still lead to a process *withdrawal* in this lock implementation?

h) The workers are not involved in the Lamport's clock. According to this, would it be possible that a worker is given access to a critical section prior to another worker that issued a request to its lock instance causally before (assuming happened-before order)? (Note that workers may send messages to one another independently of the mutual-exclusion protocol).

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]).
-include_lib("wx/include/wx.hrl").

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

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.
```