Seminar Report: Muty

Peter Grman, Leonardo Tonetto

October 19, 2012

1 Introduction

In this seminar session we implemented a distributed mutual-exclusion lock system, using three different approaches:

- `lock1`: which was the simplest version, where only the first lock acquire attempt could be considered fair. All the others would only be granted after some clients started to give up waiting for the locks;
- `lock2`: implemented an unfair priority design where each lock instance was given a fixed and different ID, and based on this ID the decision about granting locks were taken. Obviously it was unfair because each client was always requesting the lock to it's own lock instance, which could have a higher or lower priority that would be the same through the whole execution;
- `lock3`: finally this third implementation using Lamport Clocks[1], and this one presented itself as the fairest.

This time, `lock1` was already done and all we had to do was just copy&paste it from [2] and execute it to see the results. All the others, we implemented as part of this assignment.

2 System overview

This mutual-exclusion was designed to use four different modules per execution: **gui**, **muty**, **worker** and **lock**. Each of these will be briefly described next. Note that the **lock**s will use a multicast strategy and work in an asynchronous network where we don't have access to a synchronized clock.

2.1 gui

Including the *wx* external library, defines a simple frame that will show different colors depending on the message that it receives. *Yellow* for a `waiting` message, *red* for a `taken` message and *blue* for a `leave` message.

Each `worker` instance will have its own `gui` frame to show in which state it is.

2.2 muty

It's simply the module that we are going to use to execute the whole system. It spawns 4 `lock` and 4 `worker` instances. Each of these are initialized using different argument values for its `init` function that will be described next on this document. The `lock` and `worker` instances are organized like present on Figure 1 on page 2.

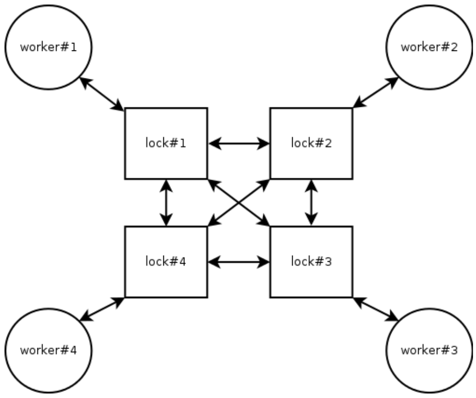

Figure 1: muty *worker* and *lock* instances

2.3 worker

The `worker` instances are the actual consumers of this system. The `init` function, called by the `muty` module, needs the `Name` of the instance that will be showed on correspondent the `gui` frame, the `lock` instance that it should request the lock (unique to each one in this implementation), a `Seed` number used for random number generation (also unique) and the maximum values that the random number generator should use to deliver the time that the instance must wait to send a new request and to release an acquired lock, `Sleep` and `Work` respectively.

The `worker`, after issuing a `take` message to its `lock` instance, will wait for a `taken` message back from the `lock` for a `deadlock` time, defined globally as 5000 ms. In case of success it will wait for a random time before releasing the lock, sending a `release` message to its `lock` instance, otherwise to avoid a deadlock condition it will also send the same message and get back to its initial state. For each of these conditions it's updating its corresponding `gui`.

Throughout the whole execution the time waiting for a lock, the time carrying a lock and the number of withdrawals are gathered and this data is presented afterwards. Some interesting conclusions can be inferred from these numbers, that will be discussed later on this document.

2.4 lock

As mentioned previously, there are three different implementations of this module, each one described next.

2.4.1 lock1 - The deadlock prone

This is the base implementation, the simplest one but most of its structure is kept for the next implementations. Three states are defined for an instance of this module:

- ***Open***: in this state, the `lock` instance will wait for a message that can be:
  - A `take` message from it's corresponding `worker`; the `lock` will broadcast a `request` message to all the other locks instances available bringing it to the ***wait*** state. While sending the `request` messages, it sends along a unique reference number that will be store in the `Refs` list so that later, on the ***wait*** state, it will be verified;
  - A `request` message from any other `lock` instance, which will automatically trigger a message `ok` to be sent to the requester, bringing it back to the ***open*** state;
  - A `stop` message sent by the `muty` instance, issued by the `stop()` function, ending the execution of this instance;
- ***Wait***: in this state, the `lock` instance will wait for the following messages:
  - A `request` message from any other `lock` instance. On this implementation, the requests are queued in the `Waiting` as a list of tuples containing the requester PID and a reference unique number.
  - An `ok` message that will contain that unique reference number sent on the ***open*** state after a `take` message, this number is then removed from the `Refs` list. If the `Refs` list is still not empty, it returns to the ***wait*** state, otherwise it's no longer waiting for an acknowledgement from the other `lock` instances, and then a `taken` message is sent to its corresponding `worker`, that now will have the lock as described before, bringing the `lock` to the ***held*** state;

  - A `release` message from its `worker`, meaning that it gave up waiting after a deadlock timeout;

- ***Held***: in this state the `worker` has the lock and the `lock` can receive the following messages:

  - A `request` message from the other lock instances, that will simply be queued;
  - A `release` message from the `worker` that owns the lock, meaning it doesn't need it anymore and it can be freed, then an `ok` message is sent back to all the `lock` instances that were on the `Waiting` list and the `lock` returns to the ***open*** state.

2.4.2 lock2 - The unfair

Here the actual coding begins. Now each `lock` instance will have its own and unique `LockID` that will be used to sort out the potential deadlock issue present on `lock1` (this will be discussed later). Based on the ID of a `request` message, the `lock` instance in the ***wait*** state will now respond with an immediate `ok` message if the requester's ID is lower that its own (note that here it was defined that lower ID means higher priority), otherwise the request will be queued in the `Waiting` list. The new code for the `wait()` function could be done this way:

```
wait(Nodes, Master, Refs, Waiting, MyLockID) ->
    receive
        {request, From, Ref, ReqID} ->
            if
                    ReqID < MyLockID ->
                R = make_ref(),
                Refs2 = [R|Refs],
                From ! {request, self(), R, MyLockID},
                        From ! {ok, Ref},
                wait(Nodes, Master, Refs2, Waiting, MyLockID);
                    true ->
                        wait(Nodes, Master, Refs, [{From, Ref}|Waiting], MyLockID)
            end;
        (...)
```

In this proposed implementation whenever a request from a higher priority `lock` instance is received, a new `request` is issued back to guarantee that the lower priority `lock` will receive an `ok` from the first one. It's not the smartest and most complex one, but except for some duplicate messages over the network this solution works without problems.

2.4.3 lock3 - Lamport clocked

This third version adds the concept of clock synchronization to make the locking decisions to be more fair. But since we don't have access to a central synchronized clock, the Lamport Clock is used for this purpose.

In it, each `lock` instance will have a logical clock that will be updated whenever an interaction between different instances occur on the system, such as `request` and `ok` messages. These messages should also now carry a time value to be used by the receiving instance.

When on the ***wait*** state, the `lock` instances will now decide if an `ok` message should be sent based on the comparison between the time when the `take` message was received by that `lock` instance and the logical time received on the message, that also in this case, should be the logical time when the other instance received the `take` message.

For this, if a `lock` instance sends a `request` message with a logical time smaller than the instance that is receiving it, this second one should respond with an `ok` message, otherwise if the time is higher, the receiver will just queue the request. In case the two values are the same, the concept of a unique ID for each instance used in **lock2** should define what will happen. Note that, for this last case, resending multiple `request` messages won't be a good idea because now the Lamport clocks will be updated for every new interaction between the instances. A proposed implementation for **lock3** is presented next:

```
init(MyLockID, Nodes) ->
    LamportClock = 0,
    open(MyLockID, Nodes, LamportClock).

open(MyLockID, Nodes, LamportClock) ->
    receive
        {take, Master} ->
            TakeTime = LamportClock,
            Refs = requests(MyLockID, Nodes, TakeTime),
            wait(Nodes, Master, Refs, [], MyLockID,
                    TakeTime, LamportClock);
        {request, From, Ref, _, ExtClock} ->
            LamportClock2 = updateClock(LamportClock, ExtClock),
            From ! {ok, Ref, LamportClock2},
            open(MyLockID, Nodes, LamportClock2);
        stop ->
            ok
    end.

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

wait(Nodes, Master, [], Waiting, MyLockID, _, LamportClock) ->
    Master ! taken,
    held(Nodes, Waiting, MyLockID, LamportClock);

wait(Nodes, Master, Refs, Waiting, MyLockID, TakeTime, LamportClock) ->
    receive
        {request, From, Ref, ReqID, ExtClock} ->
            LamportClock2 = updateClock(LamportClock, ExtClock),
```

```
            if
                TakeTime < ExtClock ->
                    wait(Nodes, Master, Refs, [{From, Ref}|Waiting],
                            MyLockID, TakeTime, LamportClock2);
                TakeTime > ExtClock ->
                    Refs2 = resendRequest(From, Refs, MyLockID, LamportClock2),
                    From ! {ok, Ref, LamportClock2},
                    wait(Nodes, Master, Refs2, Waiting, MyLockID, TakeTime, LamportClock2);
                TakeTime == ExtClock ->
                    if
                        ReqID < MyLockID ->
                            Refs2 = resendRequest(From, Refs, MyLockID, LamportClock2),
                            From ! {ok, Ref, LamportClock2},
                            wait(Nodes, Master, Refs2, Waiting, MyLockID, TakeTime,
                                    LamportClock2);
                        true ->
                            wait(Nodes, Master, Refs, [{From, Ref}|Waiting],
                                    MyLockID, TakeTime, LamportClock2)
                    end
            end;
        {ok, Ref, ExtClock} ->
            LamportClock2 = updateClock(LamportClock, ExtClock),
            Refs2 = lists:keydelete(Ref, 2, Refs),
            wait(Nodes, Master, Refs2, Waiting, MyLockID, TakeTime, LamportClock2);
        release ->
            ok(Waiting, LamportClock),
            open(MyLockID, Nodes, LamportClock)
    end.

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

held(Nodes, Waiting, MyLockID, LamportClock) ->
    receive
        {request, From, Ref, _, ExtClock} ->
            LamportClock2 = updateClock(LamportClock, ExtClock),
            held(Nodes, [{From, Ref}|Waiting], MyLockID, LamportClock2);
        release ->
            ok(Waiting, LamportClock),
            open(MyLockID, Nodes, LamportClock)
    end.

resendRequest(From, Refs, MyLockID, LamportClock) ->
    IsPidPresent  = lists:keymember(From,1,Refs),
    IsNamePresent = lists:keymember(process_info(From),1,Refs),
    if
        IsPidPresent or IsNamePresent->
            R = make_ref(),
            Refs2 = [R|Refs],
            From ! {request, self(), R, MyLockID, LamportClock},
            Refs2;
        true ->
            Refs
    end.

updateClock(MyClock, ExtClock) ->
    if
        MyClock >= ExtClock ->
```

```
        NewClock = MyClock + 1;
    MyClock < ExtClock ->
        NewClock = ExtClock + 1
end,
NewClock.
```

To solve the potential problem with multiple `request` messages being sent without need, that in this case would add a new value to the logical counter of the receiver, the `resendRequest` function was created that just simply test if the PID or the registered name of a `lock` instance is present on the `Refs` lists. If it is present nothing should else should be done, otherwise, a new `request` should be issued.

3 Evaluation

3.1 lock1

As expected the ***lock1*** works. Basically, the closer the value of `Work` time was to the deadlock time higher were the possibilities of a withdrawal, though it could be balanced with a bigger `Sleep` time. Since all the decisions were taken based on the queue of each `lock` instance, the `worker` instances were only acquiring a lock when the others were either releasing or either giving up to wait for the lock. The Table 1 on page 7 presents the average values for time to take a lock, the number of locks acquired and number of withdrawals.

[***Sleep***, ***Work***, ***Deadlock***]	*# locks taken*	*avg. time (ms)*	*# withdrawal*
[1000,2000,5000]	13,5	2277.44	0
[3000,2000,5000]	11	1325.79	0
[6000,2000,5000]	15	595.68	0
[1000,1000,5000]	30.5	1041.86	0
[1000,6000,5000]	5	3359.9	3.5
[6000,7000,5000]	8.25	2411.43	4
[7000,6000,5000]	6.25	2132.37	1.25
[1000,4000,5000]	9	3285.11	3.75
[1000,3500,5000]	8.75	2833.22	2.5

Table 1: ***lock1*** execution (avg. value for all the *worker* instances)

3.2 lock2

For ***lock2***, if we look at the *muty.erl* code, we see that statically the `lock` instances were distributed among the `worker` instances in the following way:

John with *lock#1*, ***Ringo*** with *lock#2*, ***Paul*** with *lock#3* and ***George*** with *lock#4*. The Figure 2 on page 8 shows the execution of ***lock2***.

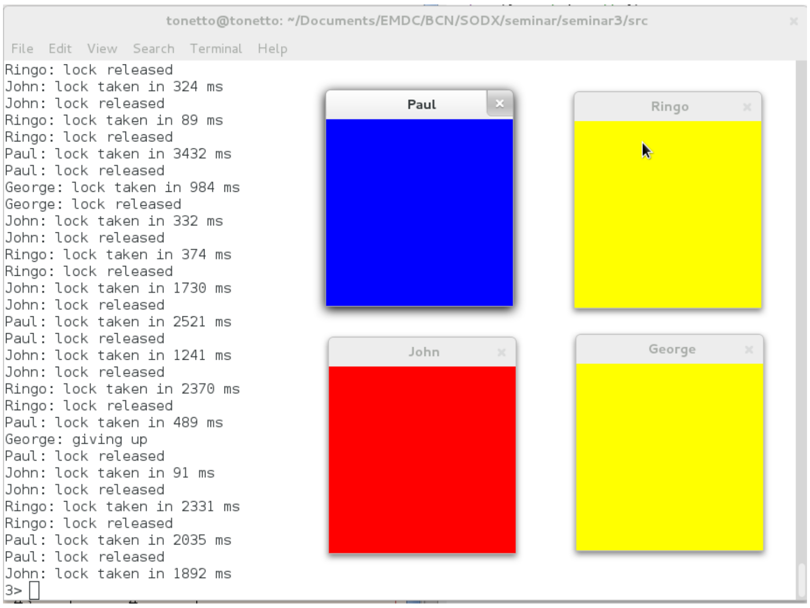

Figure 2: ***lock2*** executing

The Table 2 on page 9 presents the execution with the same values used on the previous table, for ***lock1***. We can see that for higher values of `Work` time and lower values of `Sleep` time (compared to `Deadlock` time), the system is highly unfair, leading to cases where ***George*** was withdrawing most of his lock requests. This could be compensated with higher values of `Sleep`, but in a real system this would probably not be possible.

[*Sleep*, *Work*, *Deadlock*]	# *locks taken*	*avg. time (ms)*	# *withdrawal*
[1000,2000,5000]	36	823,55	0
	33	829,36	0
	10	2045,9	8
	3	1319,0	13
[3000,2000,5000]	26	583,73	0
	22	847,14	0
	16	1989,13	0
	12	1518,67	4
[6000,2000,5000]	16	505,75	0
	14	675,43	0
	13	469,54	0
	14	501,00	1
[1000,1000,5000]	53	313,49	0
	43	501,49	0
	27	1212,96	1
	17	2100,29	3
[1000,6000,5000]	12	2551,17	0
	10	2255,0	2
	3	3938,67	9
	1	0,0	12
[6000,7000,5000]	10	1535,3	1
	10	1369,2	1
	4	1500	7
	4	1299,5	7
[7000,6000,5000]	15	1712,73	0
	11	1659,72	3
	9	1381,67	5
	4	2099,0	9
[1000,4000,5000]	26	1821,11	0
	21	1565,05	3
	14	2256,36	7
	6	1574,33	16
[1000,3500,5000]	42	1544,02	0
	33	2178,64	0
	13	2533,61	16
	2	1429,0	27

Table 2: ***lock2*** execution (***John***, ***Ringo***, ***Paul***, ***George*** values separately)

The ***lock2*** had to deal with the possible deadlock condition that could be caused when a lower priority `lock` instance, when still on the ***wait*** state, received a `request` message from a higher priority instance after having received an `ok` from that same instance. The Figure 3 on page 10 shows this situation. ***lock#2*** receives a `take` from its `work` instance and broadcast a `request` message, ***lock#1*** responds quickly with an `ok` message, but right after it receives a `take` from its own `work` instance, broadcasting a `request` too. This time, ***lock#2*** is still waiting for the others to respond, but as it received a lock request from an instance with higher priority, it's supposed to answer with an `ok`, and to not starve forever waiting it should send another `request` back to ***lock#1***. The implementation proposed (and already described) solved this problem.

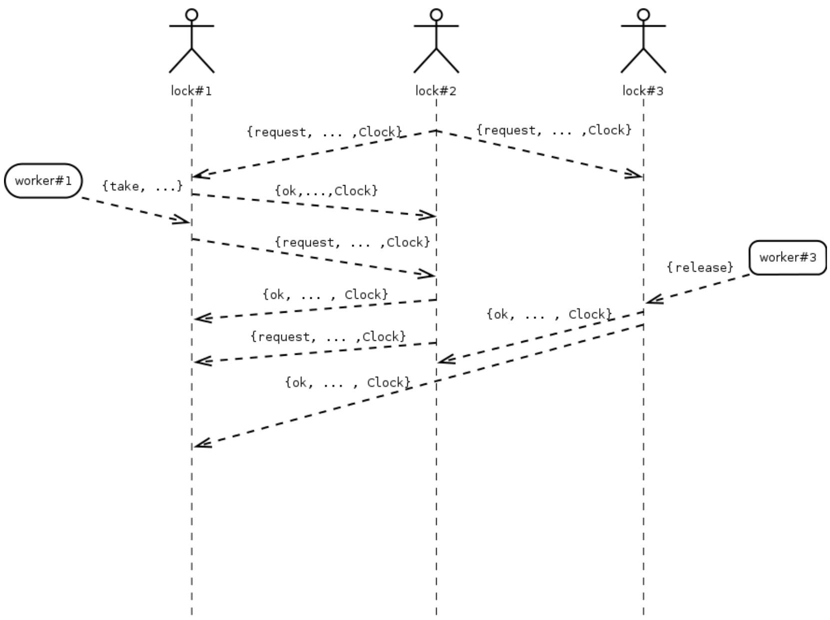

Figure 3: ***lock2*** possible deadlock condition

3.3 lock3

The ***lock3*** code implemented a fairest locking system, using Lamport logical clocks. The Figure 4 on page 11 shows a print-screen of this system working.

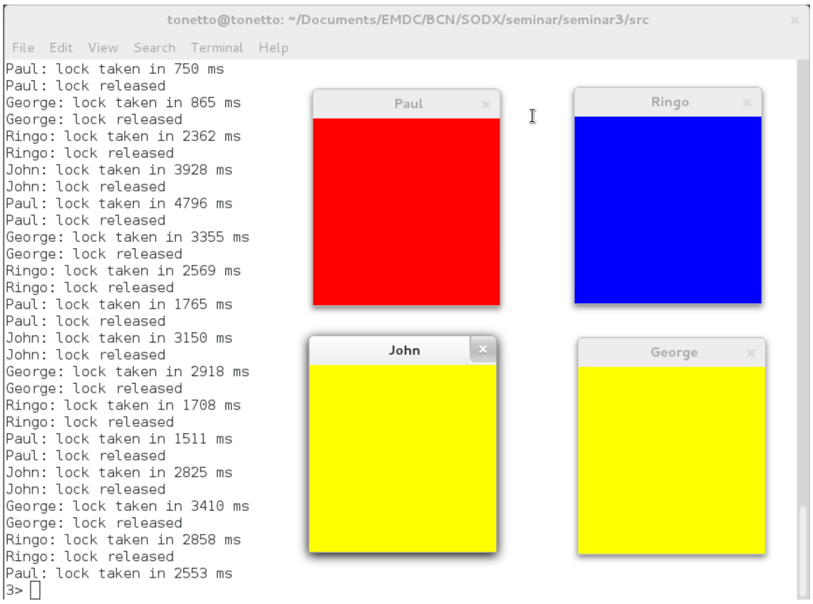

Figure 4: ***lock3*** executing

From Table 3 on page 12 we can see that this implementation was really the fairest of all. Even for high `Work` time, the `work` instances were giving up waiting for the lock in a more homogeneous way.

[***Sleep***, ***Work***, ***Deadlock***]	*# locks taken*	*avg. time (ms)*	*# withdrawal*
[1000,2000,5000]	18	2471,05	0
	18	2172,66	1
	19	2229,37	0
	20	2245,3	0
[3000,2000,5000]	23	1653.08	0
	22	1678.77	0
	21	1687,71	0
	24	1471,88	0
[6000,2000,5000]	14	461,42	0
	13	691,69	0
	12	485,25	0
	13	511,38	0
[1000,1000,5000]	39	997,67	0
	38	997,63	0
	37	1042,27	0
	39	1017,46	0
[1000,6000,5000]	9	3396,67	9
	10	3605,10	6
	8	2907,00	10
	9	2767,00	10
[6000,7000,5000]	14	1980,00	9
	13	2429,77	8
	14	2843,86	6
	13	2443,38	10
[7000,6000,5000]	8	1412,13	6
	9	2277,00	4
	12	2483,75	1
	10	2548,00	2
[1000,4000,5000]	12	3255,33	5
	12	3199,83	4
	10	3371,10	7
	14	3267,28	3
[1000,3500,5000]	13	2678,23	3
	12	2800,23	4
	11	3139,73	4
	14	2789,43	2

Table 3: ***lock3*** execution (***John***, ***Ringo***, ***Paul***, ***George*** values separately)

For a small number of `work` and `lock` instances it wasn't possible to notice a big difference between ***lock1*** and ***lock3*** implementations. Even modifying ***muty*** (calling it ***mutybis***) to create a big number (20) of both `work` and `lock` instances, the actual results didn't differ much from each implementation. This is probably caused by the fact that we are executing these instances in our own machines and using the internal loopback network interface.

Running this application on a real environment, where nodes are geographically separated and network overhead will play an important role, should give different results in the sense that ***lock3*** using logical clocks will present the fairest results. The source code for *mutybis.erl*, along with the test results for all the tests performed for this report are part of the tarball file that included this document.

4 Open questions

4.1 lock1

- *How does this lock implementation perform? What is happening when you increase the risk of a lock conflict? Why? (make tests with different Sleep and Work parameters to answer this).*

As previously described this lock implementation performs well for the predefined environment. When the `Work` time increases the `work` instances give up more frequently waiting for the lock. On the other hand, for a high `Work` time value, when the `Sleep` time increases the withdrawals decrease.

4.2 lock2

- *Justify how you guarantee that only one process is in the critical section at any time. Run tests and see how well your solution works with respect to the previous lock. What is the main drawback?*

From the Figure 3 on page 10 we explain how this is done. Lower priority instances, that had already received an `ok` from a higher priority instance, but are still waiting, once they receive a `request` from that lower ID instance (higher priority) it should re-send a `request` message along with the `ok` message. This way only one instance will be in the critical section at any time.

The main drawback is that, since each `work` instance is statically binded to one `lock` instance, that has a unique and statical ID (thus a statical priority), `work` instances that request a lock for a low priority `lock` instance will hardly acquire the lock for any execution time observed.

4.3 lock3

- *Run tests and compare this version with the former ones. 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 logically before the worker that took the lock?*

The tests were done and compared on the previous sessions of this document.

And yes, we could have a situation where a `work` instance is not given priority to a lock even if it issued a request to its `lock` instance logically before the one that actually took the lock.

It can be explained by the simple fact that it's not guaranteed that a message issued in the time $t_1$by a `work` instance will arrive to the lock system before a request sent in a time t_2 (where $t_1 < t_2$). A solution for this would be to use a synchronized clock from where all the messages sent inside the system would have a timestamp based on this clock, then these timestamps would be used to fairly decide which message was sent first and to who the lock actually should be granted.

5 Conclusions

In this seminar we implemented three different versions of a distributed mutual-exclusion lock system. We could see that by simply using statically assigned priority could lead to a highly unfair set and that using Lamport logical clocks we can easily implement a fair distributed lock system, even though these implementations bring also some complexity and potential deadlock conditions if not properly designed.

Even being a good proposal, logical clocks don't solve all the problems in a distributed system, and therefore a synchronized clock could be really helpful to solve some issues that can't be solved with the first solution. The problem is that it's really hard to deploy and maintain distributed synchronized clocks, specially when you have geographically distributed systems where the network latency is a fixed barrier impossible to leave behind.

References

[1] R. Stockton Gaines and Leslie Lamport. Time, clocks, and the ordering of events in a distributed system, 1978.

[2] Jordi Guitart. Mutty: a distributed mutual-exclusion lock. September 2012. Adapted with permission from Johan Montelius (KTH).