Muty: Lock de exclusión mutua distribuido*

Federico C. Repond
frepond@unq.edu.ar

Esteban Dimitroff Hódi
esteban.dimitroff@unq.edu.ar

26 de mayo de 2016

Introducción

La tarea es implementar un lock de exclusión mutua distribuido. El bloqueo usará una estrategia multicast y trabajará en una red asíncrona en la que no tendremos acceso a un reloj sincronizado. Haremos la implementación en tres versiones: la propensa a deadlocks, la no equitativa y la sincronizada por Lamport. Antes de comenzar es recomendable tener conocimiento teórico del algoritmo multicast y de como funcionan los relojes de Lamport.

1. La arquitectura

El escenario es que un conjunto de workers necesitan sincronizarse y, ellos decidirán aleatoriamente tomar el lock y una vez tomado mantenerlo por un tiempo corto antes de liberarlo. Cada worker recolectará una estadística de cuánto tiempo le lleva adquirir el lock para presentar datos interesantes al final de cada test.

Implementemos primero el worker y luego refinemos el lock.

1.1. El worker

Cuando el worker es arrancado se le es dado el acceso a un lock. Además se le da un nombre para imprimir mejor en la consola y una seed para que cada worker tenga su propia secuencia aleatoria. También proveeremos información del tiempo promedio que el worker duerme y trabaja.

Tendremos cuatro workers compitiendo por el lock de manera que si duermen en promedio 1000 ms y trabajan por un promedio 2000 ms tendremos un lock con altas chances de congestión. Podremos fácilmente cambiar estos parámetros para simular más o menos congestión. La constante de deadlock indica cuánto tiempo (4000 ms) se esperará por el lock, antes de darse por vencido.

*Adaptado al español del material original de Johan Montelius (`https://people.kth.se/~johanmon/dse.html`)

La GUI será un proceso que nos dará algo de feedback en la pantalla sobre qué esta haciendo el worker- La GUI podría simplemente loggear cosas en la terminal, pero una interface gráfica es por supuesto preferible. Una GUI de ejemplo, basada en la librería WX, es provista en el apéndice.

```
-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).
```

Haremos algo de auditoría y guardaremos el tiempo que tarda obtener el lock. Al final imprimiremos algunas estadísticas.

Un worker duerme por un tiempo y luego decide moverse a la sección crítica. La llamada a `critical/4` retornará información acerca de si la sección crítica fue accedida y cuánto tiempo se demoró en obtener el 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.
```

La sección crítica es accedida solicitando el lock. Esperamos por una respuesta `taken` o por un timeout. Si el lock es conseguido, se devuelve el tiempo T al proceso llamador.

La GUI es informada al momento de enviar el pedido de lock, y si se consigue o si hay que abortar.

```
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.
```

El worker termina cuando recibe un mensaje de `stop`. Simplemente imprimirá las estadísticas.

```
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. Los locks

Ahora trabajaremos con tres locks implementados en tres módulos: `lock1`, `lock2` y `lock3`. El primer lock, `lock1`, será muy simple, y no cumplirá los requerimientos de un lock. Evitará que más de un worker entre a la sección crítica pero eso es todo.

Cuando el lock es iniciado, se le dará un identificador único y un conjunto de locks pares. El identificador no será utilizado por el primer lock pero queremos que la interface sea igual para todos los locks.

```
-module(lock1).

-export([start/2]).

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

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

El lock entra en el estado `open` y espera o bien por un comando `take` para obtener el lock, o un `request` desde otro lock. Si se le ordena tomar el lock, entonces enviara un request multicast a los demás locks y luego entrará al estado `waiting`. Un request de otro lock es inmediatamente respondido con un mensaje `ok`. Notar como la referencia es utilizada para conectar el request con la respuesta.

```
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).
```

En el estado waiting el lock está esperando mensajes `ok`. Todos los requests estarán etiquetados con referencias únicas de manera de poder llevar cuenta de qué locks han respondido y a cuáles aún se los está esperando. Podríamos haber hecho una solución mas simple donde solamente esperamos por n locks pero esta versión es más flexible si queremos extenderla.

```
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).
```

Mientras el lock está esperando podría también recibir un mensaje `request` desde locks que hayan decidido tomar el lock. En esta versión del lock, simplemente agregamos estos a un conjunto de locks que tienen que esperar. Cuando el lock es liberado les enviaremos mensajes de `ok`.

Como modo de escape de la situación de deadlock, también permitimos al worker enviar un mensaje de `release` aunque el lock no haya sido obtenido aún. Entonces enviaremos mensajes de `ok` a todos los locks en espera y entraremos en el estado `open`.

En el estado `held` vamos agregando requests de los locks a la lista de locks en espera hasta que recibamos un mensaje `release` desde el worker.

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

Para el hacker Erlang hay algunas cosas para pensar. En Erlang los mensajes son encolados en el mailbox de los procesos. Si matchean un patrón en la cláusula receive son procesados, pero sino se mantienen en la cola. En nuestra implementación aceptamos y procesamos alegremente todos los mensajes, aunque algunos, como el mensaje `request` cuando estamos en el estado `held`, son simplemente guardados para después. ¿Sería posible usar la cola de mensajes Erlang y dejar que los mensajes se encolen hasta que se libere el lock? La razón por la que lo implementamos de esta manera es para hacer explícito que los

mensajes son tratados aún en el estado `held`. ¿Por qué no estamos esperando mensajes `ok`?

1.3. Algo de testing

Ahora escribamos una función de test que cree cuatro locks y cuatro workers. Conectémoslos y corramos los 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.
```

Ahora estamos usando el nombre del módulo como parámetro para la función de `start`. Seremos fácilmente capaces de testear diferentes locks con diferentes parámetros Sleep y Work. ¿Funciona bien? ¿Qué ocurre cuando incrementamos el riesgo de un conflicto de lock? ¿Por qué?

2. Resolviendo el deadlock

El problema con esta primera solución puede ser resuelto si le damos a cada lock un identificador único 1, 2 ,3 y 4. El identificador dará prioridad al lock. Un lock en el estado de espera enviará un mensaje `ok` a un lock que lo solicite si el lock tiene una prioridad más alta (siendo 1 la prioridad máxima).

Implementemos esta solución en un módulo llamado `lock2`, y veamos que funciona aunque tengamos alta contención. ¿Funciona? Hay una situación que debemos tratar correctamente. Si no, corremos el riesgo de tener dos procesos en la sección crítica al mismo tiempo. ¿Podemos garantizar que tenemos un solo proceso en la sección crítica en todo momento? En curso debemos estar preparados para poder explicar porqué la solución funciona.

Corramos algunos tests y veamos qué tan bien funciona la solución. ¿Qué tan eficientemente lo hace y cuál es la desventaja? Reportar cuáles son los resultados.

3. Tiempo de Lamport

Una mejora es permitir a los locks ser tomados con prioridad dada por el orden temporal. El único problema es que no tenemos acceso a relojes sincronizados (asumiendo que estamos ejecutando en una red asíncrona). La solución es utilizar relojes lógicos como los relojes de Lamport.

Para implementar esto debemos agregar una variable de tiempo al lock. EL valor es inicializado a cero pero es actualizado cada vez que el lock recibe un mensaje de `request` desde otro lock. El reloj de Lamport entonces lleva cuenta del request mas alto que se ha visto hasta el momento. Cuando un request es enviado, deberá tener un timestamp una unidad mas alto del visto hasta el momento.

Ahora vemos una solución donde el timestamp de Lamport no es necesario en todos los mensajes del sistema, sino en aquellos que son importantes, por ejemplo, los mensajes de `request`.

Cuando un lock está en estado de espera debe determinar si el request fue enviado antes o después de haber enviado su propio mensaje. Si esto no puede determinarse, entonces se utilizará el identificador de lock para resolver el orden.

Notemos que los workers no están involucrados con el reloj de Lamport. ¿Puede darse la situación en que un worker no se le da prioridad al lock a pesar de que envió el request a su lock con un tiempo lógico anterior al worker que lo consiguió?

Implemetaremos ésta solución en `lock3` y corramos algunos experimentos.

4. El reporte

Finalmente escribir un reporte que especifique cómo se resolvieron lock2 y lock3. Describir también que problemas y conclusiones fueron encontrados. Además se debe informar que pros y contras se ven en cada una de las soluciones.

Apéndice

Aquí proveemos una GUI. El worker arrancará la GUI y enviará mensajes si está esperando un lock, cuando lo obtiene y cuando el lock es liberado (o si el intento de obtenerlo es abortado). La ventana de la GUI sera: azul si esta abierto, amarilla si está esperando o roja para indicar que el lock fue obtenido.

```
-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}),
            wxFrame:refresh(Frame),
            loop(Frame);
        enter ->
            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.
```