Seminar Report: Muty

Sistemes Distribuïts en Xarxa

Grupo:

Domínguez Purificación, Antonio
Álvarez Guerra, Adrián
Masriera Quevedo, Miquel

Profesor:

Guitart Fernandez , Jordi

6 de abril de 2015

Índice

1. Introducción

En esta práctica se implementa en Erlang algunos de los algoritmos vistos en clase de exclusión mútua. El objetivo de estos algoritmos es el de proporcionar un sistema con el cual se garantice que en una región crítica sólo entre un único proceso del sistema distribuido.

Para la implementación de dicho sistema disponemos de 4 procesos workers: Ringo, Paul, John y George. Cada uno de estos procesos tiene asociado un proceso lock que se encarga de enviar y recibir peticiones entre ellos para ponerse de acuerdo sobre quien entra en la región crítica. En esta sesión de laboratorio hay 3 versiones de lock:

- **Lock1**: Un lock da permiso a otro si no quiere entrar en la región crítica. En caso contrario no responde a la petición y la encola.
- **Lock2**: Un lock da permiso a otro si no quiere entrar en la región crítica. En caso contrario, compara las IDs: si la del otro proceso es menor le da permiso, si es mayor no responde y encola petición.
- **Lock3**: Un lock da permiso a otro si no quiere entrar en la región crítica. En caso contrario, compara los tiempos lógicos de Lamport: si el del otro proceso es menor le da permiso, si es mayor no responde y encola petición, si son iguales compara por ID.

2. Trabajo hecho

2.1. Lock 1

Tal y como se ha dicho anteriormente, en esta primera versión de Lock sólamente se responde con un OK a un request si estamos en estado open, si estamos en estado waiting para entrar en la región crítica no respondemos y encolamos la petición. Este comportamiento se puede observar en los siguientes fragmentos de código de *lock1.erl*

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

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

2.2. Lock 2

En esta versión añadimos una comparación de IDs entre procesos. Esta comparación tiene lugar cuando recibimos un request en estado waiting. Si el ID del proceso que envia el request es menor que el que lo recibe, este último responde con un OK al primero. Dicho comportamiento se puede observar en este fragmento de código.

```
wait(Nodes, Master, Refs, Waiting, MyId) ->
    receive
        {request, From, Ref, OtherId} ->
            if
                OtherId < MyId ->
                    From ! {ok, Ref},
                    NewRef = make_ref(),
                    From ! {request, self(), NewRef, MyId},
                    wait(Nodes, Master, [NewRef | Refs], Waiting, MyId);
                true ->
                    wait(Nodes, Master, Refs, [{From, Ref}|Waiting], MyId)
```

```
            end;
        {ok, Ref} ->
            NewRefs = lists:delete(Ref, Refs),
            wait(Nodes, Master, NewRefs, Waiting, MyId);
        release ->
            ok(Waiting),
            open(Nodes, MyId)
    end.
```

2.3. Lock 3

En este caso las prioridades de las peticiones no estarán definidas por IDs, sino que utilizaremos un clock lógico (*Lamport clock*) que nos permitirá ordenar las peticiones entre locks. Para ello, necesitamos almacenar el máximo lock visto en el sistema (variable *MyLock*) que iremos actualizando a medida que recibimos peticiones y por otro lado el timestamp asociado a los requests que realiza nuestro lock (variable *timestamp*) para comparar la prioridad de nuestras requests con la de los otros procesos.

```
-module(lock3).
-export([start/1]).

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

init(MyId) ->
    MyClock = 0,
    receive
        {peers, Nodes} ->
            open(Nodes,MyId,MyClock);
        stop ->
            ok
    end.

open(Nodes, MyId,MyClock) ->
    receive
        {take, Master} ->
            Refs = requests(Nodes, MyId,MyClock),
            wait(Nodes, Master, Refs, [], MyId, MyClock, MyClock);
        {request, From,  Ref, _, RemoteClock} ->
            NewClock = lists:max([MyClock,RemoteClock]) + 1,
            From ! {ok, Ref, NewClock},
            open(Nodes, MyId, NewClock);
        stop ->
            ok
    end.

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

```
      end,
      Nodes).

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

wait(Nodes, Master, Refs, Waiting, MyId, MyClock, TimeStamp) ->
    receive
        {request, From, Ref, OtherId, RemoteClock} ->
            NewClock = lists:max([MyClock,RemoteClock]) + 1,
                if
                RemoteClock < TimeStamp ->
                    From ! {ok, Ref, NewClock},
                    wait(Nodes, Master, Refs, Waiting, MyId, NewClock, TimeStamp);

                RemoteClock == TimeStamp ->
                    if
                        OtherId < MyId ->
                                  From ! {ok, Ref, NewClock},
                                wait(Nodes, Master, Refs, Waiting, MyId, NewClock, TimeStamp);
                            true ->
                                wait(Nodes, Master, Refs, [{From, Ref}|Waiting], MyId, NewClock, TimeStamp)
                    end;
                true ->
                    wait(Nodes, Master, Refs, [{From, Ref}|Waiting], MyId, NewClock, TimeStamp)
                end;

        {ok, Ref, RemoteClock} ->
            NewClock = lists:max([MyClock,RemoteClock]) + 1,
            NewRefs = lists:delete(Ref, Refs),
            wait(Nodes, Master, NewRefs, Waiting, MyId, NewClock, TimeStamp);

        release ->
            ok(Waiting, MyClock),
            open(Nodes, MyId, MyClock)
    end.

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

held(Nodes, Waiting, MyId, MyClock) ->
    receive
        {request, From, Ref, _, RemoteClock} ->
            NewClock = lists:max([MyClock,RemoteClock]) + 1,
            held(Nodes, [{From, Ref}|Waiting],MyId, NewClock);
        release ->
            ok(Waiting, MyClock),
            open(Nodes,MyId, MyClock)
    end.
```

3. Experimentos

3.1. Lock 1

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

En esta versión de *lock*, cuando un proceso en estado waiting recibe un request de otro proceso no responde a este y lo encola, de forma que podemos tener varios procesos sin recibir ningún ok y esperando indefinidamente. Por lo tanto, se producirán más deadlocks cuanto más frecuentes sean los requests y más tiempo dure la región crítica.

Los siguientes ejemplos están realizados con un tiempo de 30 segundos. En el primero podemos ver el comportamiento de este lock utilizando un tiempo de **sleep de 1000ms y de work de 50ms**. Tal y como se puede ver hay pocos deadlocks y los 4 procesos entran varias veces en la región crítica.

- Ringo: 26 locks taken, 287.88461538461536 ms (avg) for taking, 1 withdrawals
- Paul: 28 locks taken, 268.92857142857144 ms (avg) for taking, 1 withdrawals
- John: 27 locks taken, 6.37037037037037 ms (avg) for taking, 2 withdrawals
- George: 22 locks taken, 714.5 ms (avg) for taking, 0 withdrawals

En el segundo ejemplo hemos utilizado el caso inverso: en el que los procesos apenas entrarán en la región crítica y sufrirán muchos deadlocks durante los 30 segundos del experimento. Este caso se consigue con un **sleep de 50ms y un work de 1000ms**:

- Ringo: 0 locks taken, 0 ms (avg) for taking, 4 withdrawals
- Paul: 0 locks taken, 0 ms (avg) for taking, 4 withdrawals
- John: 1 locks taken, 8.0e3 ms (avg) for taking, 3 withdrawals
- George: 2 locks taken, 3945.5 ms (avg) for taking, 3 withdrawals

ii) Split the muty module and make the needed adaptations to enable each worker-lock pair to run in different machines (that is, john and l1 should run in a machine, ringo and l2 in another, and so on). Remember how names registered in remote nodes are referred and how Erlang runtime should be started to run distributed programs.

Tras intentar sin éxito registrar todos los procesos desde un solo código mediante el uso de spawns, decidimos separar el registro de procesos en diferentes ficheros de código que ejecutaran los diferentes PCs de forma independiente. Más concretamente l1 (John) y l2 (Ringo) los ejecutará laptop@192.168.101 y l3 (Paul) y l4 (George) los ejecutará

pc@192.168.1.102.

Código 'laptop@192.168.101':

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

start(Lock,Sleep, Work) ->

    register(l1, apply(Lock, start, [1])),
    register(l2, apply(Lock, start, [2])),


    timer:sleep(2000),

    l1 ! {peers, [{l3,'pc@192.168.0.102'}, {l4,'pc@192.168.0.102'}, l2]},
    l2 ! {peers, [{l3,'pc@192.168.0.102'}, {l4,'pc@192.168.0.102'}, l1]},

    register(w1, worker:start("John", l1, Sleep, Work)),
    register(w2, worker:start("Ringo", l2, Sleep, Work)),

    ok.

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

Código 'pc@192.168.102':

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

start(Lock,Sleep, Work) ->

    register(l3, apply(Lock, start, [3])),
    register(l4, apply(Lock, start, [4])),


    timer:sleep(2000),

    l3 ! {peers, [{l1,'laptop@192.168.0.101'}, {l2,'laptop@192.168.0.101'}, l4]},
    l4 ! {peers, [{l1,'laptop@192.168.0.101'}, {l2,'laptop@192.168.0.101'}, l3]},

    register(w3, worker:start("Paul", l3, Sleep, Work)),
    register(w4, worker:start("George", l4, Sleep, Work)),

    ok.

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

3.2. Lock 2

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

En esta versión de *lock*, cuando un proceso waiting recibe un request de otro proceso no lo ignora y lo encola como en la primera versión, sino que compara su ID con el propio y le responde con un OK en caso de que el otro proceso tenga mejor ID (y mayor prioridad). De esta forma se logra reducir drasticamente el numero de deadlocks.

De la misma forma que en el experimento anterior, estos se realizan en un intervalo de 30 segundos. El primero se realiza con **sleep de 1000ms y work de 50ms**. Tal y como se puede ver no ha habido ningun deadlock y los procesos han entrado en la región crítica más veces que en el experimento anterior.

- John: 59 locks taken, 2.1186440677966103 ms (avg) for taking, 0 withdrawals
- Ringo: 54 locks taken, 4.037037037037037 ms (avg) for taking, 0 withdrawals
- Paul: 48 locks taken, 9.729166666666666 ms (avg) for taking, 0 withdrawals
- George: 52 locks taken, 9.23076923076923 ms (avg) for taking, 0 withdrawals

En el siguiente ejemplo vemos el comportamiento descrito al inicio. Con un **sleep de 50ms y un work de 1000ms**, se puede observar como los procesos que menos withdrawals sufren y más entran en la región crítica son Ringo y John, los cuales tienen el ID más bajo y, por lo tanto, tienen prioridad frente a los demás. Con estos parámetros, se podría llegar a un caso extremo de *starving* en los procesos de Paul i George.

- Ringo: 33 locks taken, 418.06060606060606 ms (avg) for taking, 0 withdrawals
- John: 35 locks taken, 446.45714285714286 ms (avg) for taking, 0 withdrawals
- Paul: 2 locks taken, 3085.5 ms (avg) for taking, 3 withdrawals
- George: 1 locks taken, 7313.0 ms (avg) for taking, 3 withdrawals

3.3. Lock 3

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

A diferencia de en el caso anterior, ahora las prioridades no están marcadas por el identificador de cada proceso sino que se implementa un reloj lógico (*Lamport clock*) que marcará el orden de las peticiones. De modo que las peticiones con un *timestamp* asociado inferior tendrán preferencia sobre las peticiones con *timestamp* más grandes.

Siguiendo el patrón usado en los experimentos anteriores, realizamos una ejecución de 30 segundos y con tiempo de **sleep de 1000ms y work de 50ms**. Como sucedía con el lock2, seguimos sin tener deadlocks y todos los procesos entran en la región crítica un número de veces similar (pocos conflictos).

- John: 58 locks taken, 3.1724137931034484 ms (avg) for taking, 0 withdrawals
- Ringo: 58 locks taken, 7.448275862068965 ms (avg) for taking, 0 withdrawals
- Paul: 60 locks taken, 5.95 ms (avg) for taking, 0 withdrawals
- George: 58 locks taken, 6.948275862068965 ms (avg) for taking, 0 withdrawals

Por último, probamos con el lock3 un tiempo de **sleep de 50ms y un work de 1000ms** y podemos observar como, a diferencia del lock2, todos los procesos siguen accediendo a la zona crítica un número de veces similar, evitando el posible *starving* de los procesos con menor prioridad.

- John: 16 locks taken, 1406.25 ms (avg) for taking, 0 withdrawals
- Ringo: 16 locks taken, 1446.375 ms (avg) for taking, 0 withdrawals
- Paul: 16 locks taken, 1506.6875 ms (avg) for taking, 0 withdrawals
- George: 15 locks taken, 1479.0666666666666 ms (avg) for taking, 0 withdrawals

4. Preguntas

4.1. Lock 1

What is the behavior of the lock when you increase the risk of a conflict?

Cuando un proceso quiere entrar en la región crítica envía requests al resto de procesos y espera a recibir un OK de cada uno de ellos. Cuando un proceso recibe un request responde con un OK sólo si él tampoco quiere entrar en la región crítica, en caso contrario no responde lo encola. Dado este comportamiento se puede llegar a la situación de que todos los procesos estén en estado waiting y ninguno responda con OK al resto, asi que ninguno de los procesos entraría en la región crítica (*deadlock*).

4.2. Lock 2

i) Justify how you guarantee that only one process is in the critical section at any time.

El caso en el que más de un proceso puede entrar a la región crítica es consecuencia a un retraso en el recibimiento de un OK que ha sido enviado. Pongamos un ejemplo:

P1, P2 y P3 son 3 procesos con IDs 1, 2 y 3 respectivamente. Imaginemos que P2 quiere entrar en la región crítica y hace request a P1 y P3, ambos responden con un OK ya que no quieren entrar y P2 recibe el OK de P1, pero el de P3 se demora por algún motivo. Si en este intervalo en el que se está demorando el OK de P3, P1 quiere entrar en la región crítica y hace request a P2 y P3, ambos están en estado waiting, y ya que el ID de P1 es menor que el ID de P2 este le contestará con un OK aunque quiera entrar en la región crítica (sigue esperando el ok de P3) provocando que al final P1 y P2 (cuando reciba el OK retrasado de P3) entren simultaneamente en la región crítica.

Para evitar este caso extremo, cuando P2 le contesta el OK a P1 además vuelve a hacer un request a P1 y como este último estará en la región crítica, encolará la petición, haciendo que P2 siga en estado de wait en vez de entrar en la región crítica.

ii) What is the main drawback of lock2 implementation?

Los procesos con poca prioridad (IDs altos) apenas entran en la región critica cuando hay mucha demanda de ella y se podria llegar a un caso extremo de *starving*.

4.3. Lock 3

Note that the workers are not involved in the Lamport 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 before (as-

suming real-time order)?

Sí. El uso del Lamport clock nos permite distinguir el orden de los eventos ocurridos entre los locks pero este no tiene por qué ser el mismo orden que el ocurrido realmente en los eventos entre workers y locks.

5. Opinión personal

En esta práctica hemos podido comprobar de primera mano cómo funcionan los sistemas de exclusión a regiones críticas donde sólo puede haber un único proceso ejecutando el código. Además, también ha sido muy útil para comprender como están implementados los relojes lógicos y la metodología para programar con el lenguaje de programación Erlang.

En conclusión, una práctica bastante útil y fácil de interpretar gracias a la utilización de la librería gráfica de Erlang que, en nuestra opinión, debería seguir en el próximo curso.