Loggy: Un logger de tiempo lógico*

Federico C. Repond
frepond@unq.edu.ar

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

14 de abril de 2016

Introducción

En este ejercicio aprenderemos a usar tiempo lógico en un ejemplo práctico. La tarea es implementar un mecanismo de logging que recibe eventos de log desde un conjunto de workers. Los eventos son etiquetados con el timestamp Lamport del worker y los eventos deben ser ordenados antes de ser escritos al stdout. Es un poco más complicado de lo que uno creería en principio.

1. Primer intento

Para tener algo por donde empezar primero construiremos un sistema que al menos haga algo.

1.1. El logger

El logger simplemente acepta eventos y los imprime en la pantalla. Estará preparado para recibir timestamps en los mensajes, pero no haremos mucho con ellos por ahora.

```
-module(logger).
-export([start/1, stop/1]).

start(Nodes) ->
    spawn_link(fun() ->init(Nodes) end).

stop(Logger) ->
    Logger ! stop.

init(_) ->
    loop().
```

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

El logger recibe una lista de nodos y les enviará mensajes pero por ahora ignoraremos esto. Lo usaremos luego cuando extendamos el logger.

```
loop() ->
    receive
        {log, From, Time, Msg} ->
            log(From, Time, Msg),
            loop();
         stop ->
            ok
    end.

log(From, Time, Msg) ->
    io:format("log: ~w ~w ~p~n", [Time, From, Msg]).
```

Erlang nos proveerá un orden FIFO para la entrega de mensajes, pero eso solo ordena mensajes entre dos procesos. Si un proceso A envía un mensaje al logger y luego envía un mensaje al proceso B, el proceso B puede reaccionar al mensaje de A y enviar un mensaje al logger. Por supuesto nos interesa que el mensaje de log de A se imprima antes que el de B, pero nada garantiza que esto ocurra así (Lamentablemente para este ejercicio tendríamos que correr un conjunto de tests extensivo antes de detectarlo en la vida real, pero sin embargo podemos introducir algún retardo en el sistema para aumentar la probabilidad).

2. El worker

El Worker va a ser muy sencillo, esperará un tiempo y luego enviará un mensaje a uno de sus pares. Mientras esté esperando, esta preparado para recibir mensajes de otros pares de manera que si corremos varios workers y los conectamos entre sí, tendremos mensajes enviados entre ellos aleatoriamente.

Para seguir lo que va sucediendo y en que orden ocurre enviaremos una entrada de log al logger cada vez que enviamos o recibimos un mensaje. Notemos que en la versión original del worker no llevamos cuenta del tiempo logico; simplemente enviará eventos al logger.

Al worker le será dado un nombre único y acceso al logger. Además proveeremos al worker un valor único como seed para el generador random. Si todos los los workers son iniciados con el mismo generador random, estarán mas en sincronismo, y serán mas predecibles y menos divertidos. Además proveeremos un valor sleep y un jitter; el primero determinará que tan activo es el worker enviando mensajes, y el segundo introducirá un retardo random entre el envío de un mensaje y el envío de la entrada de log.

```
-module(worker).
-export([start/5, stop/1, peers/2]).

start(Name, Logger, Seed, Sleep, Jitter) ->
```

```
    spawn_link(fun() -> init(Name, Logger, Seed, Sleep, Jitter) end).

stop(Worker) ->
    Worker ! stop.

init(Name, Log, Seed, Sleep, Jitter) ->
    random:seed(Seed, Seed, Seed),
    receive
        {peers, Peers} ->
            loop(Name, Log, Peers, Sleep, Jitter);
        stop ->
            ok
    end.
```

La fase de arranque en `init/5` se ve rara, pero esta ahí para que podamos iniciar todos los workers y despues informarles cuáles son sus pares. Si les diésemos la lista de pares al comienzo podríamos entrar en una race condition donde un worker envía mensajes a otros que aun no han sido creados. Por conveniencia proveemos una interface funcional.

```
peers(Wrk, Peers) ->
    Wrk ! {peers, Peers}.
```

El proceso del worker es bastante simple. El proceso esperará o bien por mensajes de alguno de sus pares o después de un lapso aleatorio seleccionará un worker al que le enviará un mensaje. El worker no sabe nada de tiempo, entonces simplemente crearemos un valor dummy, `na`, para tener algo que enviarle al logger. El mensaje podría por supuesto contener cualquier cosa, pero aquí incluiremos un valor aleatorio que esperamos único de manera de poder seguir el envío y la recepción de un mensajes.

```
loop(Name, Log, Peers, Sleep, Jitter)->
    Wait = random:uniform(Sleep),
    receive
        {msg, Time, Msg} ->
            Log ! {log, Name, Time, {received, Msg}},
            loop(Name, Log, Peers, Sleep, Jitter);
        stop ->
            ok;
        Error ->
            Log ! {log, Name, time, {error, Error}}
    after Wait ->
            Selected = select(Peers),
            Time = na,
            Message = {hello, random:uniform(100)},
            Selected ! {msg, Time, Message},
            jitter(Jitter),
```

```
            Log ! {log, Name, Time, {sending, Message}},
            loop(Name, Log, Peers, Sleep, Jitter)
    end.
```

La selección de a qué par enviar el mensaje es aleatoria, y el jitter introduce un pequeño retardo entre enviar el mensaje al par e informar al logger. Si no introducimos el retardo, difícilmente veríamos mensajes fuera de orden al correr en la misma máquina virtual.

```
select(Peers) ->
    lists:nth(random:uniform(length(Peers)), Peers).

jitter(0) -> ok;
jitter(Jitter) -> timer:sleep(random:uniform(Jitter)).
```

3. El test

Si tenemos el worker y el logger podemos armar un test para ver que todo funcione.

```
-module(test).
-export([run/2]).

%report on your initial observations
run(Sleep, Jitter) ->
    Log = logger:start([john, paul, ringo, george]),
    A = worker:start(john, Log, 13, Sleep, Jitter),
    B = worker:start(paul, Log, 23, Sleep, Jitter),
    C = worker:start(ringo, Log, 36, Sleep, Jitter),
    D = worker:start(george, Log, 49, Sleep, Jitter),
    worker:peers(A, [B, C, D]),
    worker:peers(B, [A, C, D]),
    worker:peers(C, [A, B, D]),
    worker:peers(D, [A, B, C]),
    timer:sleep(5000),
    logger:stop(Log),
    worker:stop(A),
    worker:stop(B),
    worker:stop(C),
    worker:stop(D).
```

Esto es solo una manera de disponer un caso de test. Como podemos observar, comenzamos creando el proceso de logging y cuatro workers. Cuando los workers han sido creados les eniamos un mensaje con sus respectivos pares.

Corramos algunos tests y tratemos de encontrar mensajes de log que sean mostrados fuera de orden. ¿Cómo sabemos que fueron impresos fuera de orden?

Experimentemos con el jitter y veamos si podemos incrementar o decrementar (¿eliminar?) el numero de entradas incorrectas.

4. Tiempo Lamport

Nuestra tarea es ahora introducir tiempo lógico en el proceso del worker. Cada worker debería llevar cuenta de su propio contador y pasarlo en cada mensaje que envía a los otros workers. Al recibir un mensaje el worker debe actualizar su timer al mayor entre su reloj interno y el timestamp del mensaje antes de incrementar su reloj.

Para comparar nuestras soluciones, y de paso cambiarlas de un modo interesante, deberemos implementar el manejo del tiempo Lamport en un módulo separado `time`. Este módulo debe contener las siguientes funciones:

- `zero()`: retorna un valor Lamport inicial (puede ser 0).
- `inc(Name, T)`: retorna el tiempo T incrementado en uno (probablemente ignoremos el Name, pero lo usaremos más adelante).
- `merge(Ti, Tj)`: unifica los dos timestamps Lamport (eso es, toma el mayor).
- `leq(Ti, Tj)`: retorna `true` si Ti es menor o igual a Tj.

Asegurémonos de que el módulo del worker utilice esta API y no contenga ningún conocimiento de como nosotros elegimos representar tiempos Lamport (que puede ser tan simple como 0,1,2,...). Podríamos querer cambiar la representación más adelante y solo necesitaríamos trabajar en el módulo `time`.

Hagamos algunos tests e identifiquemos situaciones donde las entradas de log sean impresas en orden incorrecto. ¿Cómo identificamos mensajes que estén en orden incorrecto? ¿Qué es siempre verdadero y qué es a veces verdadero? ¿Cómo lo hacemos seguro?

4.1. La parte delicada

Ahora la parte compleja. Si el logger solo recolecta todos los mensajes de log y los guarda para más adelante, uno podría esperar con el ordenamiento hasta que todos los mensajes se hayan recibido. Si queremos imprimir los mensaje en un archivo, o al stdout en el orden correcto pero *durante* la ejecución tenemos que tener cuidado de no imprimir nada demasiado temprano.

Debemos de alguna manera mantener una cola de retención de mensajes que no podemos entregar aun porque no sabemos si recibiremos un mensaje con un timestamp anterior. ¿Cómo sabemos si los mensajes son seguros de imprimir?

Suena fácil, y lo es por supuesto una vez que lo tengamos bien, pero habrá cosas que nos olvidaremos de cubrir antes de conseguirlo. Hay algunos modos de resolver esto pero debemos seguir estas pautas para luego hacer algunos cambios a la implementación.

El logger deberá llevar un *reloj* que lleve cuenta de los timestamps de los últimos mensajes de cada uno de los workers. Deberá mantener tambien una *cola de retención* donde contenga mensajes de log que aun no son seguros de imprimir. Cuando un nuevo mensaje de log llega deberá actualizar el reloj, agregar el mensaje a la cola de retención y recorrer la cola para encontrar los mensajes que son ahora seguros para imprimir.

Extendamos el módulo `time` para que también implemente las siguientes funciones:

- `clock(Nodes)`: retorna un *reloj* que pueda llevar cuenta de los nodos
- `update(Node, Time, Clock)`: retorna un reloj que haya sido actualizado dado que hemos recibido un mensaje de log de un nodo en determinado momento.
- `safe(Time, Clock)`: retorna `true` o `false` si es seguro enviar el mensaje de log de un evento que ocurrió en el tiempo Time dado.

El logger deberia solamente utilizar la API del módulo `time`, no debería tener conocimiento de cómo están representados los tiempos Lamport o el reloj. Si lo hacemos bien deberíamos ser capaces de luego cambiar la representación sin necesidad de cambiar el logger.

4.2. En el curso

Para el curso debemos primero tener los procesos de workers y logger implementados. Los workers deben llevar cuenta de los tiempos lógicos y actualizarlos a medida que envían y reciben mensajes. El logger debe tener una cola de retención con mensajes que aun no ha impreso. Cuando los mensajes son impresos, deben estar en el orden correcto.

También debemos escribir un reporte que describa el módulo `time` (por favor no escribir una página de código fuente; describirlo con sus propias palabras). Describir si encontraron entradas fuera de orden en la primera implementación y en caso afirmativo, cómo fueron detectadas. ¿Qué es lo que el log final nos muestra? ¿Los eventos ocurrieron en el mismo orden en que son presentados en el log? ¿Que tan larga será la cola de retención? Hacer algunos tests para tratar de encontrar el máximo número de entradas.

4.3. vectores de relojes

¿Qué diferencias habrían si se hubieran utilizado vectores de relojes? Es un tema muy interesante. Si la primera parte de este ejercicio resultó fácil, intentemos implementar vectores de relojes. Descubriremos que no es tan complicado y de hecho proveen algunos beneficios.