Detección de fallas en Erlang*

Federico C. Repond
frepond@unq.edu.ar

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

31 de marzo de 2016

Introducción

En esta asignación veremos cómo funcionan la detección de fallas en Erlang. Primero dispondremos de un sistema con dos procesos ejecutándose en un nodo Erlang, y luego dividiremos el sistema para que los procesos corran en dos nodos en máquinas distintas.

1. Enviar y recibir

Comencemos por dos procesos simples; uno que envía mensajes de ping cada segundo y uno que los consume alegremente. El `producer` será arrancado primero y esperará a que el `consumer` se conecte. Una vez que se han conectado, los mensajes de ping podrán enviarse, cada uno etiquetado con un número incremental. Debemos tener la posibilidad de detener el producer, y él deberá entonces enviar un mensaje final al consumer.

1.1. El producer

El producer puede verse así, comenzamos declarando el modulo y la interface.

```
-module(producer).
-export([start/1, stop/0, crash/0]).

start(Delay) ->
    Producer = spawn(fun() -> init(Delay) end),
    register(producer, Producer).

stop() ->
    producer ! stop.
```

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

```
crash() ->
    producer ! crash.
```

La función `start/1` creará un nuevo proceso y lo registrará bajo el nombre `producer`. Recibe un parámetro llamado `Delay`, que sera la cantidad de milisegundos entre mensajes. La función `stop/0` simplemente enviará un mensaje de stop al proceso registrado bajo el nombre `producer`.

En la inicialización del producer, esperaremos a que un consumer nos envíe un mensaje. Un consumer enviará un mensaje `{hello, Consumer}` donde `Consumer` es el identificador del proceso que debe recibir los mensajes.

```
init(Delay) ->
    receive
        {hello, Consumer} ->
            producer(Consumer, 0, Delay);
        stop ->
            ok
    end.
```

El proceso está luego implementado usando la construcción `after` que nos permite esperar por un mensaje un cierto tiempo antes de continuar. Si no es recibido ningún mensaje de `stop` enviamos un mensaje `ping` al consumer.

```
producer(Consumer, N, Delay) ->
    receive
        stop ->
            Consumer ! bye;
        crash ->
            42/0 %% this will give you a warning, but it is ok
    after Delay ->
        Consumer ! {ping, N},
        producer(Consumer, N+1, Delay)
    end.
```

Si recibimos un mensaje `stop` enviamos un mensaje `bye` al consumer y terminamos la ejecución. Esta es la forma controlada y de hacer saber al consumer que no debería esperar ver más mensajes. Si recibimos un mensaje `crash` simplemente terminamos sin informar al consumer. Veremos por supuesto que esto provoca problemas.

1.2. El consumer

El consumer es igualmente simple, debe tener las siguientes propiedades:

- Debe exportar dos funciones: `start/1` que toma un identificador de proceso o nombre registrado de un producer como argumento y `stop/0` que termina el proceso.

- Al inicializar deberá enviar un mensaje `hello` al producer antes de entrar en su estado recursivo con un *valor esperado* puesto en 0.
- Deberá recibir una secuencia de mensajes `ping` y revisar que el mensaje contenga el *valor esperado*. Si es correcto debe imprimir el número en la pantalla y continuar. Si recibe un número más alto debe imprimir un warning antes de continuar. En ambos casos el valor esperado siguiente es uno más que el valor recibido.
- Si el proceso recibe un mensaje `bye` (enviado por el producer) o un mensaje `stop` (enviado al llamar `stop/0` el proceso debe terminar.

2. Detectando un crash

Arranquemos el producer y después arranquemos el consumer en la misma shell de Erlang. El parámetro para el consumer es simplemente `producer` ya que es el nombre local en bajo el cual está registrado.

Deberíamos poder arrancar el producer y luego el consumer y ver como los mensajes son recibidos por el consumer. Si hacemos `stop` del producer, el consumer es informado mediante un mensaje `bye` y termina elegantemente.

El problema, es si simulamos un crash y el producer simplemente termina. El consumer ahora estará trabado, esperando un mensaje `ping` o `bye` que nunca llegará. Esto puede ser resuelto implementando el consumer usando una construcción `after`. Si no se ha recibido ningún mensaje en diez segundos, podemos suponer que el producer esta muerto y hacer otra cosa. Esta es una forma bastante cruda de resolver el problema y hay una manera mejor.

El sistema runtime de Erlang puede ayudar ya que sabe si el producer esta vivo o no. En Erlang tenemos una construcción llamada `monitor/2` que pregunta al sistema para darnos información sobre el estado de otro proceso. Para usar esta característica arrancamos un monitor para el producer y agregamos una cláusula adicional la definición recursiva.

El monitor es arrancado de la siguiente manera:

```
init(Producer) ->
    Monitor = monitor(process, Producer),
    Producer ! {hello, self()},
    consumer(0, Monitor).
```

El *Monitor* retornado de la llamada a `monitor/2` es simplemente una referencia única así que podemos determinar que monitor fue notificado.

Además de tener un parámetro adicional, un mensaje más es agregado a la definición recursiva.

```
{'DOWN', Monitor, process, Object, Info} ->
    io:format("~w died; ~w~n", [Object, Info]),
    consumer(N, Monitor);
```

Éste mensaje es lo que deberíamos recibir si hay un crash en el producer o termina. Sabemos que el mensaje corresponde al monitor que comenzamos ya que el segundo elemento de la tupla es idéntico a nuestro `Monitor`. El elemento `Object` es el identificador de proceso (o nombre registrado) del proceso muerto y el elemento `Info` nos da información de porqué fue terminado.

Podrán preguntarse porqué estamos haciendo la llamada recursiva ya que es por supuesto inútil. Si el producer murió no recibiremos más mensajes y quedaremos trabados en la misma situación que antes. Veremos más adelante que usaremos este detalle para mostrar un comportamiento particular.

Ejecutemos el producer y el consumer y luego provoquemos el crash en el producer para ver que está funcionando.

3. Un experimento en dos nodos

Erlang es, como sabemos, bastante transparente cuando se trata de distribución. Podemos fácilmente ejecutar nuestros producer y consumer en nodos Erlang distintos.

3.1. En el mismo host

Usemos dos nodos, gold y silver, ejecutando en la misma máquina. Arranquémoslos en modo distribuido y asegurémonos que los arrancamos con la misma cookie así pueden comunicarse.

```
> erl -sname gold -setcookie foo
```

Si el producer se arranca en el nodo `silver` el consumer deberá tener el argumento `{producer, 'silver@host'}` donde `host` es el nombre de la máquina donde estamos ejecutando.

Ahora no solo podemos simular un crash en el producer, sino ademas matar el nodo Erlang donde está corriendo. ¿Qué mensaje se da como razón cuando el nodo es terminado? ¿Porqué?

3.2. Un experimento distribuido

Ahora las cosas se están poniendo interesantes; ejecutemos cada uno de los nodos en diferentes computadoras. En principio todo debería ser normal y deberíamos ser capaces de provocar un crash en el proceso. ¿Qué sucede si matamos el nodo Erlang en el producer?

Ahora probemos desconectar el cable de red de la máquina corriendo el producer y volvamos a enchufarlo despues de unos segundos. ¿Qué pasa? Desconectemos el cable por períodos mas largos. ¿Qué pasa ahora?

¿Qué significa haber recibido un mensaje `'DOWN'`? ¿Cuándo debemos confiar en el?

¿Se recibieron mensajes fuera de orden, aun sin haber recibido un mensaje `'DOWN'`? ¿Qué dice el manual acerca de las garantías de envíos de mensajes?

Éstas preguntas refieren a los problemas principales al implementar sistemas distribuidos.