Práctica 3 - Tolerancia a fallos

Sistemas distribuidos

Alberto Calvo Rubió, Óscar Baselga Lahoz
Universidad de Zaragoza

24 de noviembre de 2019

Índice

1. Introducción

En esta práctica se parte del escenario 3 de la práctica 1 con la arquitectura Master-Pool-Worker con ciertas modificaciones. En esta ocasión los workers pueden tener fallos y el QoS se mide en el tiempo que se tarda desde que el cliente envía la petición hasta que se devuelve el resultado correcto, se exige no mas de 2,5 segundos. Para resolver estos problemas se van a diseñar diferentes técnicas de tolerancia a fallos y el algoritmo del Matón.

2. Análisis y clasificación de los fallos

Los workers en esta ocasión pueden fallar con un determinado porcentaje de probabilidad y según el número de servicios que hayan realizado. Estos errores se han identificado como:

- **Halt**: el worker ejecuta System.halt matando al proceso y a la máquina virtual de erlang (matando a los demás procesos que corrían sobre ella).
  Fallo tipo **Crash**, ya que tras el fallo, deja de funcionar.
- **Fibonnaci lento**: calcula el fibonnaci del número proporcionado pero de una forma más lenta, que en algunos casos puede tardar más que el QoS establecido.
  Fallo tipo **Timming**, ya que la respuesta tarda más de lo esperado.
- **Operación errónea**: esta operación devuelve un número decimal y además puede provocar una excepción en el worker, provocando no enviar ningún resultado.
  Fallo tipo **Response**, ya que devuelve una respuesta incorrecta, y además de tipo **Crash** pudiendo saltar la excepción.
- **Omisión del resultado**: el worker decide directamente no enviar ningún resultado.
  Fallo tipo **Omission**, ya que la conexión ha sido establecida, pero no se ha recibido respuesta tras una petición.

3. Estrategias de detección y corrección de los fallos

Para detectar y corregir los fallos explicados anteriormente se han diseñado una serie de estrategias para corregir cada uno de ellos

3.1. Replicación

El problema que encontramos es que cuando realizamos una petición al master pueden suceder diferentes errores: el máster cae, el worker no contesta, se cae, devuelve un resultado erróneo, etc. Cualquiera de estos puede suceder, pero con una pequeña posibilidad. Una forma sencilla de solucionar cualquiera de ellos es **replicando las peticiones**. Para conseguir esto hay que hacer una serie de cambios respecto a la practica 1.

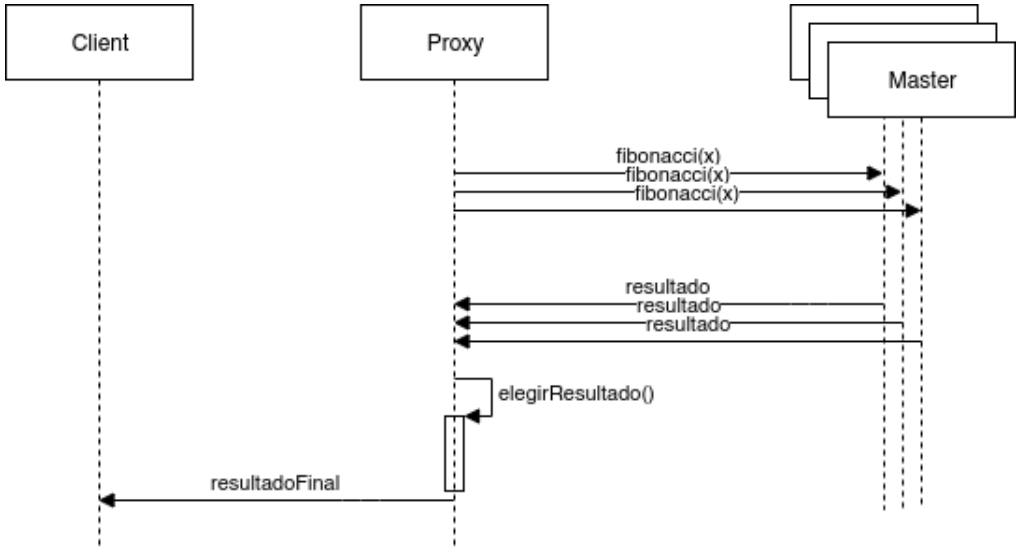

Figura 1: Proxy y replicación Máster

En primer lugar, es necesario crear un **proxy**, este elemento servirá para que todo el sistema replicado pueda tener lugar, ya que habrá varias soluciones y este se encargará realizar la **comparación de resultados** y elegir cuál es el correcto para devolver al cliente. Además sirve de **abstracción del sistema respecto del cliente**, en el que este solo se comunicará con el proxy sin saber que hay detrás de él.

Una vez el proxy permite recibir distintos resultados, lo importante es **replicar el máster**. Esto va a permitir que el proxy envíe a cada uno de ellos la petición del cliente. Estos pedirán al Pool un worker que ejecute la operación, y estos devolverán el resultado al master. Conforme más master se repliquen, más se consigue reducir la probabilidad de que no se obtenga ningún resultado correcto.

Con estos elementos conseguimos solucionar todos los tipos de errores explicados anteriormente, ya que con mucha probabilidad se obtendrá un worker que realice correctamente su trabajo y se lo envíe al proxy. Aunque los problemas están resueltos aún es necesario cumplir el QoS, por ello ha sido necesario incluir más elementos en la estrategia que los complemente.

3.2. Timeout

Un elemento fundamental para complementar la replicación y mantener el QoS se trata de los timeout. Al replicar las peticiones, se debe establecer una serie de timeouts para que tanto el proxy como el master no se queden esperando todos los resultados. Si se esperase demasiado tiempo y el proxy ya tuviese algún resultado, se debe enviar antes de que no se cumpla el QoS.

El primero de ellos es el **timeout del proxy**. Este salta tras darle un determinado tiempo a los máster para enviar sus resultados, sino se asume que han habido problemas durante la búsqueda

del resultado, ya que las operaciones solicitadas no superan en tiempo de ejecución el timeout propuesto.

El segundo es el **timeout del master**. Si el worker al que le han solicitado la operación esta tardando más que el timeout, se considera que ha fallado, ya sea por omisión del mensaje, timing, halt, etc. Siguiendo una de las filosofías de elixir "***Fail fast***"se procede a **matar al worker** que esta tardando demasiado en responder o posiblemente ya haya muerto.

3.3. Supervisión y despliegue automático

Para poder asegurar el QoS y seguir la estrategia de matar a los worker que están tardando demasiado, es necesario **disponer de una gran cantidad de workers a la espera** y una forma de **detectar que han sido eliminados y crearlos** lo antes posible.

El elemento que se ha añadido respecto a la practica 1 ha sido un proceso denominado "*vigilante*", algo similar a los supervisores de elixir. Este proceso se ha diseñado de tal forma que en vez de que el Pool cree los workers, el Pool creará tantos procesos vigilante como workers se quieran crear. Los vigilantes **crearán el proceso Worker mediante un link**. De esta forma, serán notificados en cuanto se muera el proceso hijo y se procederá a crear uno nuevo. Sin embargo, puede que al caer el worker, caiga todo el nodo, por ello se han tomado dos decisiones:

- Cada **nodo contendrá un solo worker** evitando una caída múltiple de workers
- Cada vigilante realizará **pings al nodo** donde se haya detectado que se ha muerto el proceso worker para conocer si ha caído también y **reiniciarlo vía SSH**

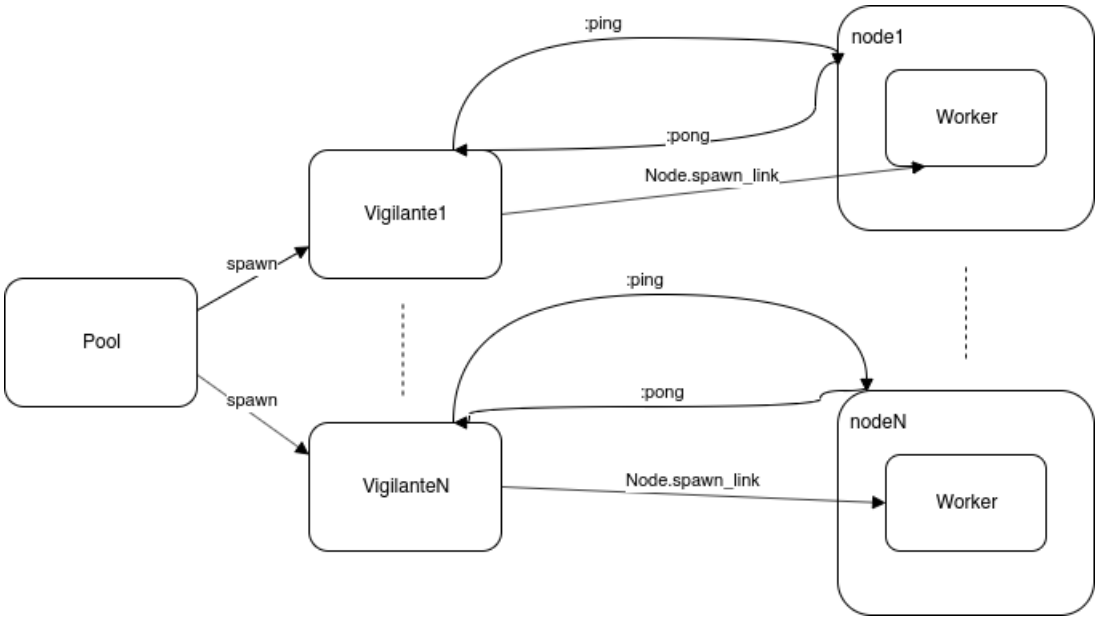

Figura 2: Estrategia de supervisión

Además, según este sistema de despliegue automático se pueden crear tantos workers como se indiquen al Pool y este creará sus correspondientes vigilantes. Los vigilantes crearán los nodos en las direcciones que les ha proporcionado el Pool y seguido se iniciarán los workers.

4. Algoritmo del matón

Se ha implementado el algoritmo del matón en los master. Para reflejar mejor este comportamiento se ha diseñado un diagrama de estados. Cada estado está implementado como una función en elixir. Las transiciones se representan como:

entrada que produce transición / salida

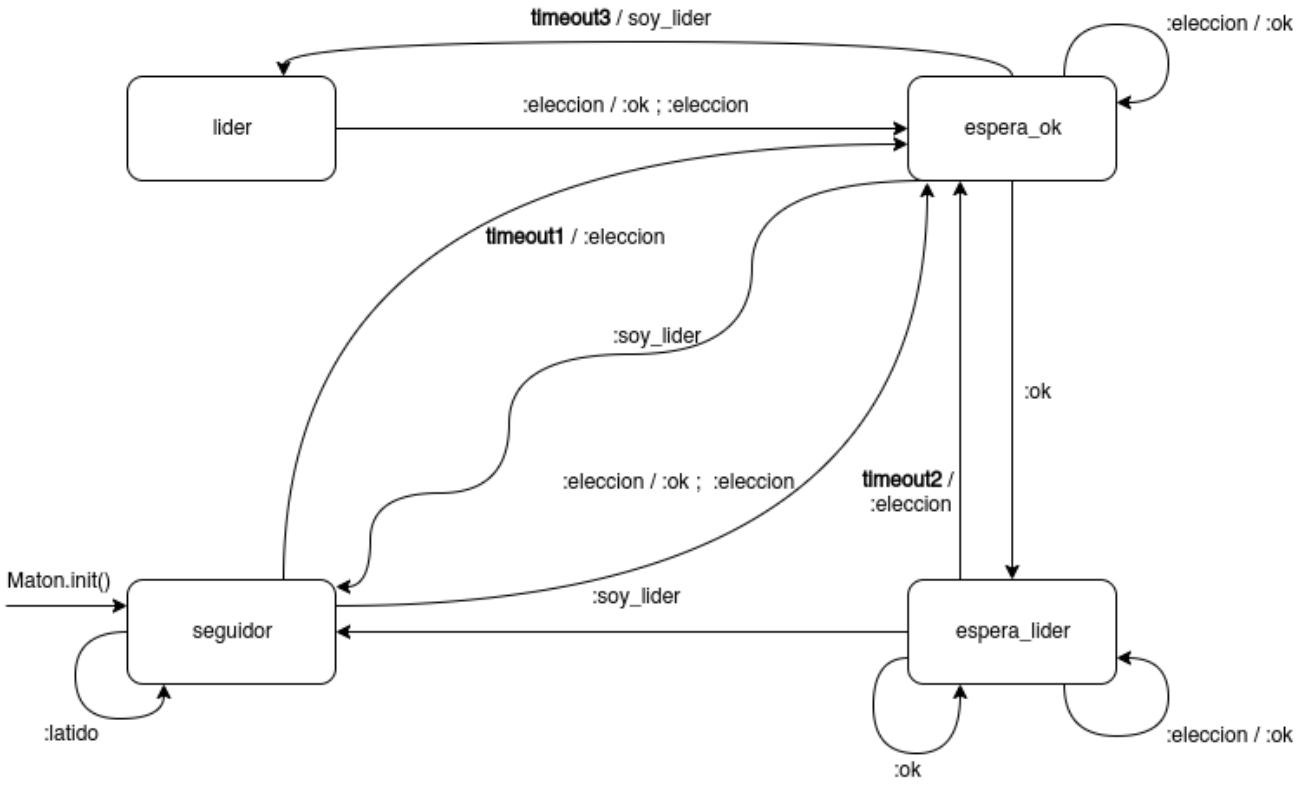

Figura 3: Diagrama de estados del algoritmo del Matón

La función del líder consiste en controlar si los demás master están vivos, de forma que si uno de ellos cae, se reiniciará de forma similar a como los vigilantes reinician los workers.