Práctica 3 - Tolerancia a Fallos Sistemas Distribuidos

Pedro Tamargo Allué - 758267
Juan José Tambo Tambo - 755742

Análisis y clasificación de los fallos que pueden aparecer

En el código proporcionado, se puede observar que el comportamiento de los workers viene dado de manera aleatoria ya que al generarse (Worker.init) se le asigna un número aleatorio entre 1 y 10. Este número se corresponde a cada cuantas tareas cambia el comportamiento del *Worker*.
Los fallos detectados que pueden aparecer están clasificados en 3 grupos:

- Fallos Tipo 1:
    - Fallos de cálculo. El nodo no realiza el cálculo. Pueden existir diversas razones para que se de este fallo:
        - El cálculo no termina en un tiempo razonable
        - El cálculo no termina (el proceso *Worker* ha terminado)
- Fallos Tipo 2:
    - Fallos de caída del nodo. Cuando el nodo cambia su comportamiento, hay una probabilidad de un 10% de que se de.
- Fallos Tipo 3:
    - Fallos de cálculo erróneo. El nodo realiza el cálculo en un tiempo razonable pero no este no devuelve el dato en las unidades correctas.

Estrategias de detección para cada uno de los fallos

Para la detección de los fallos anteriormente explicados, se ha utilizado la herramienta *Node.ping* que devuelve información acerca del nodo por el cual se está preguntando.
Si dicho nodo está activo (ejecutando el intérprete *"iex"*), la operación devolverá un átomo *:pong*, en el caso de que el nodo haya caído (no esté ejecutando el intérprete *"iex"*), devolverá un átomo *:pang*. Por lo tanto, podemos clasificar los fallos de la siguiente manera:

Devolución Node.ping	Fallo tipo 1	Fallo tipo 2	Fallo tipo 3
:pong	x		x
:pang		x	

Podemos observar que, con la operación *Node.ping* tanto los fallos de tipo 1 como los de tipo 3 devuelven el átomo *:pong*, esto indica que el nodo sigue ejecutando el intérprete *"iex"*. Por lo tanto, los fallos de tipo 3 no son detectados con la utilidad *Node.ping*. Pero por la propia definición de fallo de tipo 3, podemos observar que este fallo devuelve un número en un tiempo de cálculo razonable, por ello, la detección de este fallo ocurre si el tipo del dato devuelto no coincide con el tipo del dato que debería haber devuelto. En este caso, cuando el tipo de dato que devuelve no coincide con un entero, se convierte a una variable de tipo entero mediante la función *trunc(int).*

Estrategias de corrección para cada uno de los fallos

En caso de fallo de tipo 1, como se ha mencionado anteriormente, los nodos siguen ejecutando el intérprete *"iex"*, pero al haber expirado el *timeout* que se le ha otorgado a la ejecución, se considera que esos nodos no van a poder responder en un tiempo razonable. Por ello, en caso de detectar este tipo de fallo, el proxy indica al pool que el worker asignado no está respondiendo. Cuando el pool recibe este aviso, elimina dicho worker de la lista de ocupados, pero no lo introduce en ninguna de las listas de disponibles, por lo que nunca será asignado a otro proxy que realice una petición de worker.
En el caso de que hayamos detectado un fallo de tipo 2, es decir, que el nodo ha caído, se le indica al pool de la misma manera que en el caso anterior, haciendo así que ese nodo worker no esté disponible en futuras peticiones.
Por último, en el caso de fallo de tipo 3, se corrige en el momento que el proxy recibe el resultado del nodo worker. Se comprueba si el valor recibido es de tipo entero (utilizando la función is_integer) y en caso contrario, se realiza la operación *trunc* sobre ese dato, convirtiéndolo así en un dato de tipo entero.

Modificaciones arquitecturales respecto del escenario 3 de la práctica 1

En el escenario 3 de la práctica 1, mientras que el proceso principal del servidor se encargaba de recibir las peticiones del cliente, a su vez se ejecutaba unos procesos *"comunicar*", encargados de comunicarse con el *pool* para obtener un worker sobre el que ejecutar la petición del cliente. Sin embargo, en el escenario de esta práctica, éste proceso *"comunicar"* desaparece, siendo reemplazado por un proceso *proxy*.

Este proxy, en lugar de ser un proceso lanzado en la propia máquina donde se encuentra el *máster*, es un proceso ejecutado en otra máquina (Proxy Machine, la cual ejecuta un intérprete *"iex"*, para crear procesos de forma remota sobre ella), para minimizar así la carga de trabajo sobre el *máster*. Además, para cada tarea enviada por el cliente se replica en dos procesos *proxy*, garantizando así el *QoS*. Se ha decidido utilizar replicación para garantizar el *QoS* debido a que se pueden dar diversos fallos que retrasarían la respuesta del nodo si se tratase de manera secuencial.

La ejecución de los proxys comienza con un *pre-protocol*, mediante el cual ambos obtienen el PID del otro proxy de su pareja. Seguidamente solicitan al proceso *pool* la asignación de un *worker*.

Cuando uno de los dos proxys recibe la terminación de su worker asignado, se lo comunica a su pareja en el *post-protocol*, enviando una tupla con el átomo "*:fin_proxy*". Seguidamente, procede a recibir la confirmación por parte de su pareja de que ha recibido la orden, o bien la indicación de que el su pareja ha terminado también, lo que puede suceder si ambos terminan en un tiempo similar (ambos están ejecutando el *post-protocol*). En el primer caso, se envía resultado al cliente. En el segundo caso, el proxy con el mayor PID se comunica con el cliente. Por último, en ambos casos, los proxys le indican al *pool* que el *worker* asignado está disponible y terminan su ejecución.

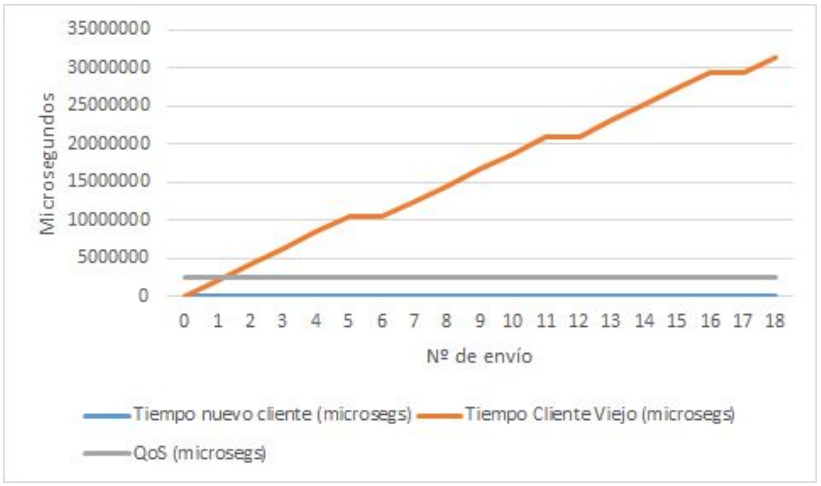
A la hora de que un *proxy* quiera devolver su *worker* asignado al *pool*, se evaluará el funcionamiento del mismo en la última tarea de cara a intentar averiguar el funcionamiento en la próxima. Gracias a este método de clasificación, se asignan tareas específicas a *workers* que hayan demostrado cierto comportamiento. Por ejemplo, si un cliente hace una petición de cálculo de 1500, dicha tarea se le asignará solamente a los *workers* cuyo comportamiento haya sido evaluado $[\&Fib.fibonacci\_tr\backslash 1, false]$ en la tarea anterior.

Resultados de ejecución

En el guión se pide que se debe mantener el *QoS* en todos los envíos, por lo que se han realizado mediciones para comprobar que esto se cumplía. Los resultados, utilizando el código de cliente que se proporcionaba en la práctica han sido los siguientes:

Como se puede observar, en ninguna ocasión se cumple el *QoS* necesario, el cual es 2,5 segundos. Esto se debe a que el cliente proporcionado está diseñado de tal manera que genera un máximo de 8 peticiones de cálculo cada dos segundos, pero las va recibiendo de manera secuencial en el mismo proceso cliente. Para solucionarlo, se ha diseñado una nueva versión de cliente, el cual lanza un proceso encargado de recibir los resultados,

mientras el hilo principal genera la carga de trabajo. De nuevo, se han tomado mediciones y los resultados han sido los siguientes:



Gráfica comparativa de los tiempos de los clientes y el QoS.

Para el modelado del sistema, se han utilizado diagramas de secuencia con los mensajes intercambiados entre cada uno de los componentes del mismo.

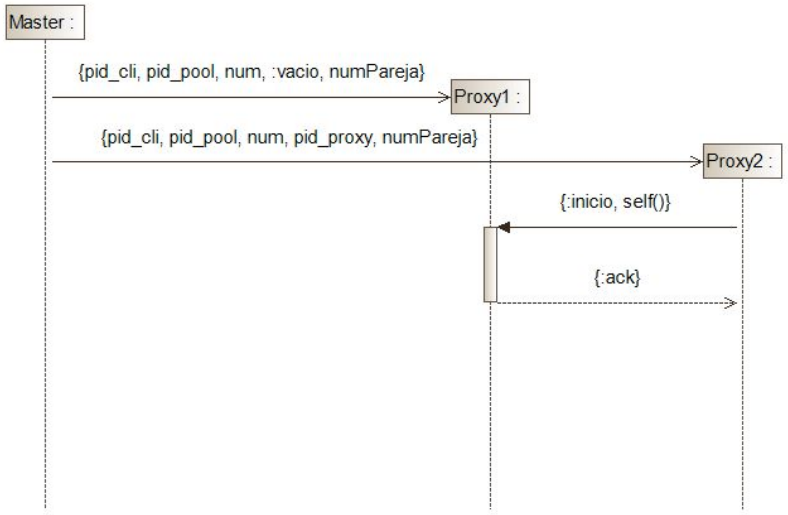


Diagrama de secuencia del *pre-protocolo* de los proxys

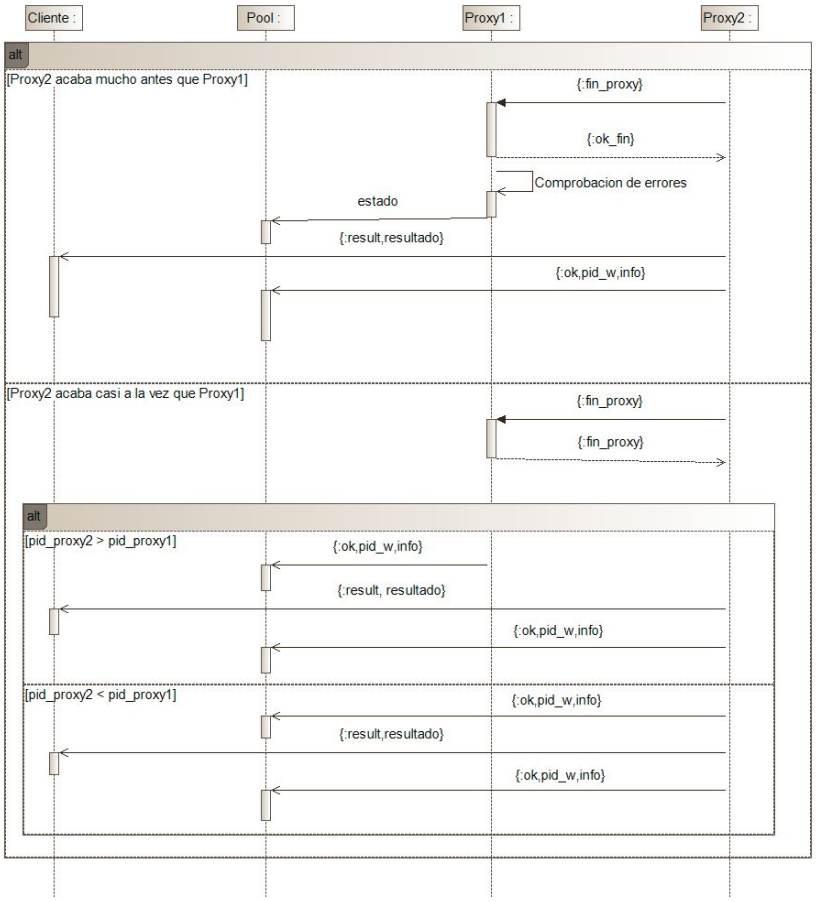


Diagrama de secuencia del *post-protocolo* de los proxys.

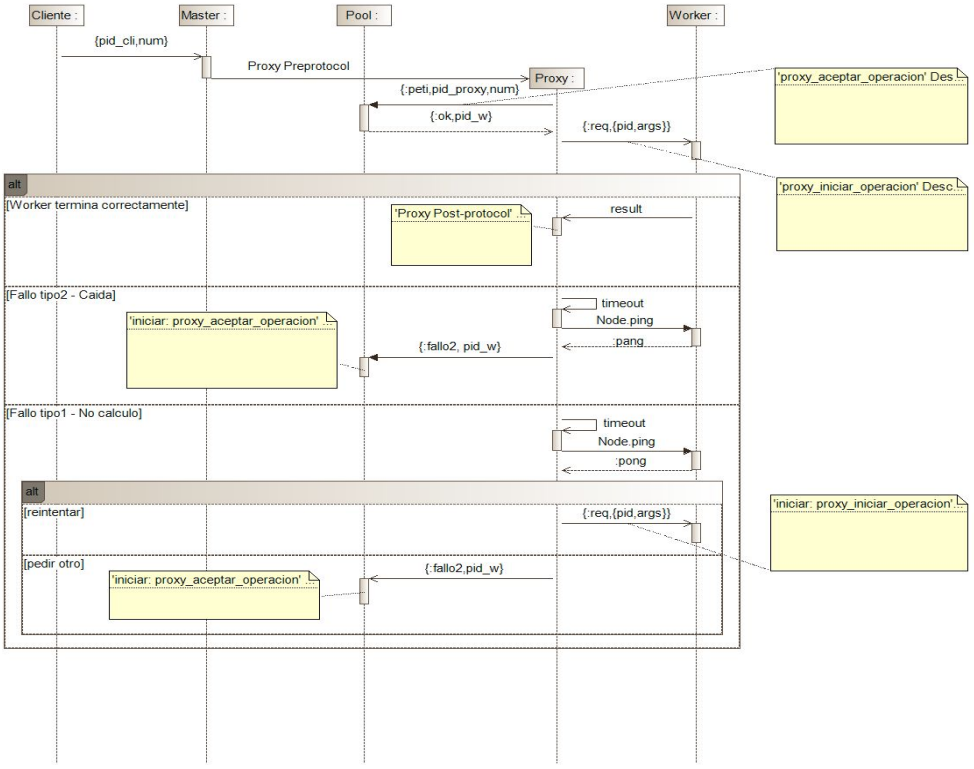


Diagrama de secuencia de los mensajes intercambiados entre los distintos componentes del sistema para la realización de una petición de cálculo emitida por el cliente.