# Tolerancia a fallos

**Eduardo Gimeno Soriano**

**Sergio Álvarez Peiro**

**22 de noviembre de 2019**

# Introducción

La práctica consiste en el tratamiento de errores, mediante la implementación del algoritmo de detección de fallos cliente-servidor visto en clase, de la arquitectura de la práctica 1, el escenario 3: un master con un pool de workers.

## Análisis y clasificación de los fallos que pueden aparecer

En esta práctica el proceso worker puede generar 3 tipos de fallos, además de un fallo en la propia ejecución del código. Los fallos que se generan en el worker tienen una probabilidad cada uno que a su vez depende de si el worker va a generar un nuevo tipo de operación, que depende de un contador que se va aumentando en la recursividad.

Un error puede ser que el proceso cuando elige una operación tiene la probabilidad de no usar ninguna implementación y ejecutar un halt, por lo que la máquina virtual de Erlang se para y el worker se va a caer, no pudiendo atender a más peticiones. La probabilidad de este fallo es 1 entre 10 cuando se elige una operación.

El resto de operaciones son distintas implementaciones de fibonacci, que se diferencian en el tiempo que tardan en completarse. Como se quiere asegurar un QoS en el que las tareas se completen en 2,5 segundos, si se elige una implementación lenta se considera como un fallo. Las 3 implementaciones tienen una probabilidad de 2 entre 10 cuando se genera una operación.

El último fallo que genera el worker tiene una probabilidad de 3 entre 10 y es que el parámetro omission tenga el valor a true, haciendo que la respuesta al cliente no se mande nunca.

El fallo que es un error de ejecución que puede ocurrir no es un fallo que genere el worker, sino un error de que el parámetro a calcular enviado por el cliente sea demasiado grande. Como la implementación of usa dos operaciones que elevan un número al número que envía el cliente. Si este número es igual a 1500 no se puede calcular. Este error se da en la última petición de genera_workload.

## Estrategias de detección para cada uno de los fallos

Para la detección del fallo por caída de la máquina de Erlang, violación de la QoS y omisión de respuesta se utiliza un timeout en el receive que ejecuta el proxy esperando la respuesta del worker que se le haya asignado.

Para el fallo que proviene de la imposibilidad de realizar el cálculo de la lista de 1500 números de Fibonacci se detecta en el master. En cuanto recibe el mensaje del proxy con el valor a calcular comprueba si vale 1500.

## Estrategias de corrección para cada uno de los fallos

Para los fallos detectados mediante timeout, una vez vence se realiza un nuevo intento, teniendo un máximo de 10 intentos.

El sistema inicialmente contaba con dos workers, existiendo la posibilidad de caída de un worker se han añadido más, esto sumado a que la carga de trabajo es mayor que en la práctica 1. Sin fallos con cuatro workers se puede garantizar la QoS, pero debido a la existencia del fallo por caída se ha duplicado el número a ocho. No subsana este error pero el objetivo es postergar que el sistema se quede sin workers el mayor tiempo posible.

Para el fallo detectado en el master, este envía al proxy un mensaje de que dicha operación no está permitida, el proxy una vez ha recibido este mensaje, se lo comunica al cliente.

## Modificaciones arquitecturales respecto del escenario 3 de la práctica 1

En la práctica 1 el sistema estaba formado por un master, un pool de recursos y dos workers. Ahora el sistema cuenta con ocho workers y un proxy entre el cliente y el master.

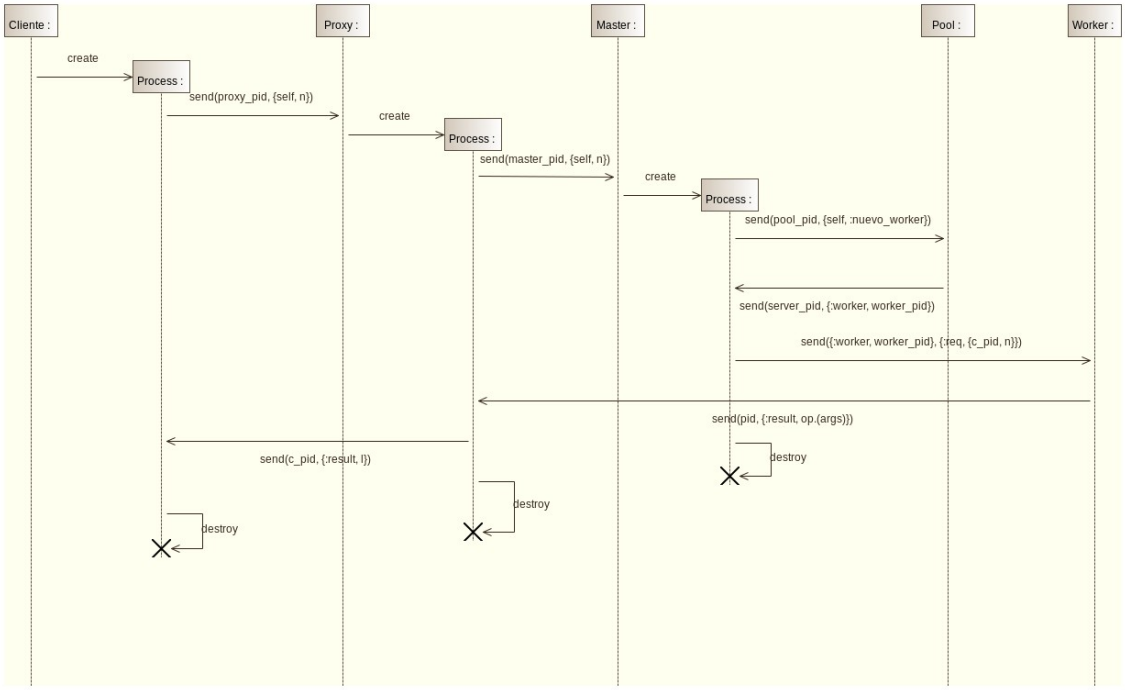


Figura 1: Intercambio de mensajes en el caso normal

# Validación Experimental

Para comprobar el comportamiento del sistema, se ha probado a ejecutar el cliente, master, pool y proxy en una máquina del laboratorio y 8 workers repartidos en otras dos máquinas. También se ha probado a ejecutar con 4 workers solo.

El objetivo que se quiere conseguir con estas pruebas es ver cómo se recupera el sistema de los fallos. Para eso el proxy muestra por pantalla cada vez que se está reintentando cuando el worker no ha enviado el resultado a tiempo. Esto solo pasará cuando se dé el fallo de que no se cumple el QoS o el worker no envía respuesta. El

cliente también muestra por pantalla cada vez que recibe un resultado o cuando a fallado 10 veces la misma operación.

Al final se observa cómo empiezan a fallar todos los workers (haciendo el halt) y no se pueden completar más tareas.

# Conclusiones

La detección y tratamiento de errores es una parte crucial a tener en cuenta a la hora de diseñar un sistema distribuido. Muchos algoritmos, como el Ricart-Agrawala, dependen de que no se de ningún fallo cuando el sistema está en ejecución. Por desgracia, en la práctica esto no ocurre así.

Implementar un buen tratamiento de fallos ofrece una cierta garantía de que el sistema puede seguir funcionando durante un tiempo mayor que si no se gestionasen.

# Bibliografía

[1] Material de la asignatura de Sistemas Distribuidos del grado de Ingeniería Informática de UNIZAR.

[2] Documentación del lenguaje Elixir https://elixir-lang.org/docs.html