Capítulo 11: Escalabilidad

Víctor Iranzo

2 de junio de 2018

1. Introducción

Podemos hacer todo lo posible para evitar que un sistema falle, pero llego un punto en la escala, la probabilidad de fallo es inevitable. Muchas organizaciones invierten mucho esfuerzo en evitar que un fallo se produzca, pero muy poco en mecanismos para recuperar el sistema una vez se ha producido.

Los mecanismos de escalabilidad de nuestro sistema han de ser conformes a sus necesidades: tener un sistema capaz de reaccionar automáticamente al incremento de carga o a un fallo es fantástico pero exagerado para ciertos servicios. También es necesario cuantos fallos pueden tolerar o cuánto tiempo puede estar el sistema caído para nuestros usuarios. Estos requisitos se pueden definir así:

- Latencia o tiempo de respuesta: cuánto tiempo puede una operación tardar. La carga que sufre un sistema puede influir en su tiempo de respuesta.
- Disponibilidad: cuánto tiempo está disponible un servicio respecto del tiempo que se deseaba que estuviera en funcionamiento. ¿Puede estar el servicio caído o tiene que estar disponible 24/7?
- Durabilidad de los datos: cuánto tiempo se deben almacenar ciertos datos y cuánta pérdida de datos se considera aceptable.
- Resilencia: es la capacidad de un sistema para tolerar fallos y continuar trabajando. Un sistema que por culpa de un servicio caído deja de funcionar es menos resilente que un sistema que puede continuar ofreciendo el resto de sus funcionalidades.

Existen varios modos de fallo que se pueden dar cuando un sistema está caído. Responder de forma lenta es uno de los peores porque aumenta la latencia de todas las peticiones para responder que el servicio está caído. Esto puede producir fallos en cascada dentro de la cadena de invocaciones.

2. Gestión de fallos en el sistema

En el libro "Antifragile"se explica como una organización como Netflix basada en la infraestructura de AWS prueba la tolerancia a fallos de sus servicios incitando al fallo de estos. Para ello, emplea un conjunto de programas que componen el "ejército de simios".

Chaos Monkey se encarga durante ciertas horas al día de apagar máquinas de forma aleatoria. Chaos Gorilla hace una función similar pero con los centros de datos. Latency Monkey simula redes de bajo rendimiento. Todos estos fallos pueden darse en producción y esta librería es una buena manera para comprobar si se está preparado para tolerarlos. Sin embargo, ¿cómo podemos gestionarlos una vez se producen?

2.1. Timeouts

Una manera de detectar los fallos es con una cuenta atrás. Esperar demasiado tiempo puede ralentizar al sistema entero. Esperar poco tiempo puede suponer considerar como fallidas tareas exitosas que han tardado más de lo normal. Una buena táctica para establecerlos es poner el mismo valor por defecto a todas las llamadas del sistema y ajustar los timeouts en función de los registros que obtengamos.

2.2. Cortocircuitos

Esta esta técnica se aplica cuando un servicio está caído o pasa un límite fijado por devolver timeouts o errores. Cuando esto ocurre, se dejan de mandar peticiones al servicio para que este devuelva un fallo rápido y no ralentice al resto del sistema. De esta forma, después de un período de tiempo podemos comprobar automáticamente si el servicio se ha recuperado y volverle a redirigir tráfico.

Como en el caso anterior, podemos establecer un cortocircuito con una condición por defecto y ajustarla según los casos específicos. Mientras la conexión está cerrada, las peticiones recibidas se pueden encolar para aten-

darlas más tarde. Existen diferentes librerías que implementan esto, como son la libería de Polly para .NET y la librería de Netflix Hystrix para Java.

Los cortocircuitos también se pueden emplear para tareas de mantenimiento programadas: cuando se tiene que modificar un servicio, se puede cortar la conexión de este y restablecerla una vez realizado el mantenimiento.

2.3. Bulkhead

Son una aproximación para aislar al sistema de un fallo en uno de sus componentes. Si tenemos una conexión distinta para cada servicio con el que interaccionemos, el que una de las conexiones vaya despacio no afectará al resto.

La separación en microservicios facilita el uso de bulkheads. A diferencia de los timeouts y los cortocircuitos que se emplean para actuar cuando un fallo se detecta, los bulkheads se pueden emplear para evitar que el fallo ocurra. Hystrix, por ejemplo, permite rechazar peticiones bajo ciertas condiciones para evitar que un servicio se sature, lo que se llama load schedding. Esto puede ser una buena medida para no convertir un servicio en un cuello de botella y evitar su fallo.

2.4. Aislamiento

Cuanto más dependa un servicio de otro para funcionar, más impacta la "‘salud"de uno sobre el otro. Esto se percibe sobretodo en las llamadas síncronas: un servicio puede tardar en responder no porque él no esté funcionando correctamente sino porque otro a quién ha invocado no contesta, produciendo un fallo en cascada.

3. Idempotencia

Si una operación se idempotente, podemos repetirla más de una vez sin un impacto adverso. Este tipo de operaciones es útil cuando no estamos seguros de si una operación se ha realizado y reenviamos la misma petición. En situaciones en las que tenemos varias instancias de un mismo servicio suscritas a los mismos eventos, aunque dos instancias procesen la misma petición no se producirá ningún error.

El sistema entero no tiene porque ser idempotente, solo las funcionalidades de negocio que se deseen. Algunos verbos HTTP como el GET o el PUT están definidos en su especificación como idempotentes. Si estos verbos no son idempotentes y los clientes piensan que sí, pueden surgir problemas a la hora de consumir el servicio.

4. Técnicas de escalabilidad

Generalmente un sistema se escala para lidiar con posibles fallos o mejorar su rendimiento. Vamos a ver diferentes técnicas para este propósito.

Rediseñar un sistema para que sea escalable no debe considerarse un fallo sino un éxito. En las fases iniciales de un proyecto no tiene sentido invertir demasiado tiempo en hacer el sistema escalable porque puedes estar invirtiendo esfuerzo para soportar una carga de trabajo que posiblemente igual no se dé nunca.

La mejor manera de hacer un sistema resilente es no situar todos los servicios en una misma infraestructura, ya sea una máquina física, red, etc. Algunos proveedores de infraestructura ofrecen facilidades para este propósito. Con esto se consigue que no haya un único punto de fallo.

4.1. Escalabilidad vertical

Consiste en aumentar las capacidades de la máquina donde se ejecuta parte del sistema, como mejorar su RAM o CPU. Esta aproximación puede resultar cara: un servidor grande puede costar más que dos pequeños que juntos sumen las mismas capacidades. Además, muchas veces el software no está preparado para beneficiarse de estas mejoras.

Aunque puede ser una manera sencilla para mejorar el rendimiento, problemas como la resilencia no se solucionan ya que si solo hay un servidor hay un único punto de fallo del sistema.

4.2. Dividir la carga de trabajo de un servicio

Si un servicio ofrece funcionalidades con diferente nivel de uso o criticidad, el servicio se puede dividir en otros más pequeños. De esta forma, se puede crear un servicio con la funcionalidad más crítica que se escale para aumentar su resilencia y de forma diferente al otro menos crítico.

4.3. Balanceadores de carga

Una forma de aumentar la resilencia de un servicio es evitar que este sea el único punto de fallo. Esto se puede conseguir teniendo varias instancias del mismo a las que se accede a través de un balanceador de carga. Un balanceador de carga se basa en un algoritmo para distribuir las llamadas que recibe entre las diferentes instancias de un servicio y además elimina instancias cuando no están accesibles y las vuelve a añadir cuando se recuperan. El balanceador de carga es transparente a los consumidores de un servicio.

Los balanceadores de carga pueden implementarse a nivel del hardware o el software. Los primeros son más difíciles de automatizar mientras que los segundos son más flexibles (mod_proxy).

4.4. Sistema basado en trabajadores

Emplear un balanceador de carga no es la única manera de tener diferentes instancias de un servicio. Un aproximación basada en trabajadores consiste en tener diferentes instancias que trabajan sobre un mismo backlog de peticiones. El número de trabajadores se puede adaptar a la demanda de cada momento (peaky load).

Los trabajadores en sí no necesitan ser resilentes: aunque un trabajador falle mientras atienda una petición, esta se terminará atendiendo por otra instancia más tarde. Una herramienta para implementar este tipo de sistemas es Zookeper.

5. Escalabilidad en bases de datos

Aunque se haya aumentado la resilencia del servicio ejecutando varias instancias de este en diferentes máquinas, todas ellas siguen accediendo a una misma infraestructura: la base de datos. La base de datos sigue siendo un único punto de fallo y puede convertirse en un cuello de botella. Además, si una misma base de datos contiene esquemas de diferentes servicios, su fallo puede ser catastrófico.

En esta sección veremos diferentes soluciones para escalar la base de datos según su tipo. Sin embargo, la solución más empleada suele ser aumentar las capacidades de la máquina donde se hospeda la base de datos, aunque esto puede emplearse hasta cierto límite.

5.1. Escalabilidad en la lectura

En bases de datos relacionales, los datos pueden ser copiados del nodo principal a varias replicas para distribuir las lecturas. Así, la escritura debe hacerse en el nodo principal pero la lectura se puede hacer en cualquier nodo.

Las modificaciones que se hagan en el nodo principal se propagan a las replicas en algún momento posterior a cada escritura. Esto significa que algunos datos que se leen en una replica pueden ser inconsistentes porque el modelo de consistencia es eventual.

Esta aproximación se emplea comúnmente y es fácil de emplear, pero pueden obtenerse beneficios similares a través de memorias cachés.

5.2. Escalabilidad en la escritura

La escalabilidad en la escritura es mucho más compleja. Una aproximación para llevarla a cabo es el sharding, donde se tiene la base de datos distribuida en diferentes nodos. Cuando se realiza una operación, a los datos (para su lectura o modificación) se les aplica una función hash y según el resultado obtenido se manda la consulta a un nodo u otro.

La complejidad del sharding proviene de consultas que requieren datos de múltiples nodos porque requiere consultar cada uno de los shards y juntar los resultados en memoria o tener un nodo con todos los datos de los shards para la lectura. Algunas herramientas suelen emplear cachés para las consultas que involucran a más de un shard.

Entre las herramientas existentes se pueden mencionar Mongo y Cassandra. Entre las funcionalidades de Cassandra encontramos por ejemplo que permite añadir nuevos shards en tiempo de ejecución y recolocar los contenidos del resto una vez hecho esto y que ayuda a mejorar la resilencia mediante la replicación de un mismo dato en más de un nodo.

5.3. CQRS

La segregación de responsabilidades de consultas y comandos (CQRS) consiste en separar el sistema de base de datos en dos partes: uno que atienda los comandos para modificarla y otro para atender las consultas. Así, cada una puede escalar de forma diferente.

Estas dos partes pueden hospedarse en servicios diferentes, aunque es-

to choca con servicios que ofrecen operaciones CRUD sobre una entidad. Además, la base de datos de lectura debe sincronizarse con los cambios que se hacen sobre la base de datos de escritura, lo que significa que puede dar datos obsoletos.

6. Caching

El uso de una caché consiste en almacenar el resultado de una operación para poder emplearlo en una petición posterior en lugar de gastar tiempo recalculando la misma operación.

Otros posibles usos de una caché son los siguientes: en la escritura, para mejorar el rendimiento de tal forma que el cambio se añade a un buffer que más adelante será escrito, o para mejorar la resilencia, ya en algunos sistemas si un servicio está caído podemos servirnos de datos almacenados en la caché (aunque estos puedan estar obsoletos) en lugar de devolver un fallo.

6.1. Tipos de caching

- Caching en la parte cliente: el cliente es quien almacena los datos cacheados. Con la ayuda del servidor decide durante cuánto tiempo son válidos los datos y cuándo debe solicitarlos de nuevo. Con esta aproximación se pueden reducir drásticamente las llamadas a través de la red. Sin embargo, la invalidación de los datos es compleja porque se debe trasladar a todos los clientes.
- Caching en el proxy: con esta aproximación se sitúa un proxy entre el cliente y el servidor que almacena los datos. Esta aproximación es opaca al cliente y al servidor y se puede emplear para cachear los resultados de más de un servicio a la vez. Los datos que se pueden almacenar son los del tráfico HTTP que redirige un proxy inverso a través de herramientas como Squid o Varnish.
- Caching en la parte servidora: el servidor es quien gestiona este sistema a través de cachés en memoria o herramientas como Redis, Memcache. Si la caché está dentro de los límites del servicio puede ser más fácil razonar sobre ella. Para el cliente la caché es transparente.

Según nuestras necesidades de carga o con que frecuencia se modifican los datos elegiremos una aproximación u otra, pudiendo emplearse diferentes en el conjunto del sistema. Hay que tener en cuenta que cuantas más cachés

haya entre la fuente de datos y el cliente, más probabilidad hay de que los datos estén obsoletos y más difícil será determinar si estos son todavía válidos.

6.2. Caching en el protocolo HTTP

El protocolo HTTP es ampliamente usado y existen un gran número de herramientas que gestionan de esta forma la caché de datos como son Squid o Varnish. En las cabeceras HTTP se pueden especificar directivas como cache-control para cachear respuestas durante un período de tiempo especificado en el valor de la clave Expires.

Otra aproximación es emplear etiquetas (ETags) para determinar si un recurso ha cambiado. Mediante el verbo GET se puede solicitar un recurso de tal forma que el servidor solo nos enviará este si se cumple cierta condición. Si hemos obtenido un recurso con un ETag y su tiempo de vida (TTL) ha expirado, podemos enviar al servidor una petición en forma de un GET condicional con la cláusula If-None-Match y la etiqueta del recurso. El servidor nos podrá responder de 2 posibles formas: con un "304 Not modifiedïndicando que el recurso que tenemos cacheado no ha sido modificado o con un "200 OK"que contenga la última versión del recurso y su nuevo ETag.

6.3. Fallos en la caché

Si se producen un gran número de fallos en la caché, las peticiones a la fuente de datos aumentarán drásticamente ralentizando el sistema entero.

Una manera de solucionar esto es haciendo que las llamadas entre la caché y la fuente de datos se hagan de forma asíncrona. De esta forma, cuando se produzca un fallo en la caché se devolverá un fallo al cliente inmediatamente y una invocación al origen de los datos, que eventualmente actualizará la caché. Obviamente, esta aproximación no tiene sentido para muchas situaciones.

7. Autoescalado

- Escalado reactivo:
- Escalado predictivo:

8. El teorema de CAP

8.1. Sistemas AP

8.2. Sistemas CP

8.3. Balance entre sistemas AP y sistemas CP

9. Descubrimiento de servicios

9.1. DNS

9.2. Herramientas para el descubrimiento de servicios

- Zookeeper
- Consul
- Eureka

10. Documentación de servicios

- Swagger
- HAL

11. Descripción de un servicio