Tema 6 - Escalabilidad

Guía de estudio

1. Introducción

Los sistemas distribuidos deben ser escalables. Según [1], "un sistema se dice que es escalable si puede manejar la adición de usuarios y recursos sin sufrir una pérdida apreciable de rendimiento o un incremento de su complejidad administrativa". Es decir, un sistema escalable debe admitir la adición tanto de usuarios que soliciten servicio al sistema como de los recursos necesarios para proporcionar estos servicios. En ambos casos, las tareas relacionadas con esas adiciones no deben generar ningún trastorno en la prestación de servicios (no debe incrementarse el tiempo de respuesta en aquellos servicios que sean interactivos, ni disminuir el rendimiento en aquellos que no lo sean).

Esta definición incluyó además la identificación de tres dimensiones en el estudio de la escalabilidad:

- *Escalabilidad de tamaño*: Un sistema presentará escalabilidad de tamaño cuando el número de usuarios atendidos por el sistema llegue a crecer linealmente con el número de sus nodos.
  En este tipo de escalabilidad se persigue atender a un número elevado de usuarios y para ello se recurre a añadir ordenadores al sistema. Si el sistema no fuera escalable, se llegaría a un punto en que la adición de nuevos ordenadores no tendría ningún efecto positivo sobre la capacidad de servicio, pudiendo llegar incluso a decrecer el rendimiento global.
- *Escalabilidad de distancia*: Un sistema presentará escalabilidad de distancia cuando tolere los problemas que comporta una mayor distancia entre sus nodos. Estos problemas son un mayor retardo de propagación, un menor ancho de banda y, generalmente, una menor fiabilidad en la transmisión.
  Por ejemplo, los proveedores de infraestructuras *cloud* suelen disponer de múltiples centros de datos en los que mantienen sus ordenadores. Si un usuario solicita máquinas virtuales en múltiples centros de datos, la escalabilidad de distancia será relevante para garantizar un buen comportamiento de las aplicaciones desplegadas.
- *Escalabilidad administrativa*: Este tipo de escalabilidad se dará cuando el sistema admita que múltiples organizaciones colaboren en la administración de sus nodos.
  La mayor parte de los middleware utilizados en los sistemas *grid* de cómputo intensivo permiten que múltiples organizaciones dejen disponibles grupos de ordenadores propios para que sean utilizados por las aplicaciones desplegadas sobre esos sistemas. En estos casos, algunas tareas de administración para cada grupo de ordenadores corren a cargo de la organización que los adquirió y los ha dejado disponibles en el sistema.

Además de estas tres dimensiones, recientemente también se ha pasado a distinguir dos clases diferentes de escalabilidad [2]:

- *Escalabilidad vertical* ("Scale-up"): Consiste en incrementar la capacidad de un nodo determinado reemplazando alguno de sus componentes *hardware* (por ejemplo: añadiendo procesadores adicionales si la placa base lo permite, instalando módulos adicionales de memoria RAM, reemplazando su disco duro por otro con mayor capacidad y ancho de banda, etc.). Un caso extremo es la sustitución del nodo entero

por otro con mejores prestaciones. A este caso puede llegarse bien por avería del nodo antiguo o por su obsolescencia.

- *Escalabilidad horizontal* ("Scale-out"): Consiste en añadir nodos al sistema. Es un nuevo nombre para lo que tradicionalmente se había llamado escalabilidad de tamaño.

Los conceptos de escalabilidad vertical y horizontal surgieron en el ámbito de la computación paralela. Tradicionalmente las aplicaciones de cómputo intensivo habían utilizado equipos multiprocesadores. Para reducir las tareas de administración del sistema se solía recurrir a la utilización de un único equipo para ejecutar una aplicación determinada (siempre y cuando su ejecución fuera abordable en un tiempo razonable). Por ello, la forma habitual de escalar un sistema de este tipo consistía en recurrir a la escalabilidad vertical. En la década de los '90 se popularizaron los "*clusters*" de ordenadores. En lugar de diseñar la aplicación para un solo equipo multiprocesador, se optó por utilizar un conjunto de ordenadores ubicados en un mismo *rack* e interconectados mediante una red local rápida. Así, se pasó de un modelo basado en escalabilidad vertical a otro basado en escalabilidad horizontal.

Estos conceptos se han aplicado durante la última década también al campo de la computación distribuida y ahora son comunes en ambas áreas (sistemas paralelos y sistemas distribuidos).

Hay que resaltar también que la escalabilidad vertical sufre algunas limitaciones importantes. Por una parte, la sustitución de alguno de los componentes del equipo suele exigir que el sistema deba ser parado para realizar ese reemplazo. Una vez instalado el nuevo componente, el sistema deberá ser iniciado de nuevo, modificando en algunos casos parte de su configuración. Con ello, ese ordenador no está disponible durante un intervalo prolongado. Además de la disponibilidad, otro factor a considerar es el coste económico de estas sustituciones. Si se prevé que el sistema deba utilizar escalabilidad vertical deberán seleccionarse componentes duraderos y fácilmente ampliables. Eso incrementa su coste de adquisición y a ese coste inicial habrá que sumar el de los componentes que se vayan adquiriendo posteriormente. En cualquier caso, con el transcurso del tiempo siempre se llegará a un estado en el que ya no interesará realizar nuevos cambios sobre un equipo con cierta edad. Las arquitecturas van evolucionando y abaratándose. Así, se llega a un punto en que un equipo antiguo, por mucho que se amplíe, no puede competir frente a máquinas más modernas y con mejor relación prestaciones/coste.

A su vez, la escalabilidad horizontal resulta ser más económica (puede basarse en equipos de gama media o incluso baja), no compromete la disponibilidad del sistema (siempre y cuando se realice una reconfiguración cuidadosa) y, en principio, no parece tan limitada (siempre se pueden añadir nuevos equipos al sistema, empleando los mecanismos de intercomunicación necesarios).

La dimensión más importante para obtener un servicio escalable es la escalabilidad de tamaño. Neuman [1] identificó cuatro mecanismos básicos para obtener este tipo de escalabilidad. Son los siguientes:

- Reparto de tareas.

- Reparto de datos.
- Replicación.
- Uso de *caches*.

Estos mecanismos comparten unos mismos objetivos: (a) Realizar una adecuada distribución de la carga y la responsabilidad de ejecución entre todos los nodos participantes, (b) Evitar en lo posible las acciones de sincronización entre las diferentes actividades de una aplicación distribuida, reduciendo así sus intervalos de bloqueo (cuando un hilo o proceso esté bloqueado, la aplicación no avanzará y eso no interesa que ocurra), (c) Aumentar la capacidad de servicio global minimizando el esfuerzo necesario.

Esos objetivos están estrechamente ligados a la consistencia, ya que debe haber varias réplicas de cada componente de un determinado servicio. Si los datos en cada réplica mantuvieran una consistencia fuerte con los de otras réplicas, se requeriría una alta sincronización entre las réplicas cada vez que un elemento fuera modificado. Eso impediría que el servicio fuera escalable. Por otra parte, una consistencia relajada permitiría divergencias entre las réplicas cuando el servicio fuera utilizado y los usuarios podrían obtener resultados inesperados. Por tanto, hay un compromiso entre la escalabilidad de un servicio y su consistencia.

Hay otros aspectos de la consistencia en sistemas distribuidos que no han sido presentados todavía en esta asignatura. Esos aspectos deben ser tenidos en cuenta para diseñar y construir servicios escalables:

- El teorema CAP justifica que cualquier modelo de consistencia que exija consenso entre los procesos no podrá soportarse en un servicio geo-replicado, pues hay un compromiso entre: (1) consistencia fuerte, (2) disponibilidad del servicio y (3) tolerancia a las particiones de la red.
- Los modelos de replicación clásicos descritos en el Tema 5 facilitan, al menos, consistencia secuencial. Otros modelos de replicación serán necesarios para desarrollar servicios escalables.
- Según se ha dicho acerca del mecanismo de reparto de datos explicado arriba, los datos persistentes deberían distribuirse entre todas las instancias de un servicio. Sin embargo, las bases de datos relacionales no son fáciles de repartir. Por tanto, otros tipos de almacenes de datos persistentes resultarán necesarios para desarrollar y desplegar un servicio escalable. Los almacenes NoSQL facilitan una solución a este problema.

Esos tres aspectos serán descritos en detalle en las secciones siguientes.

2. Teorema CAP

Desde hace mucho tiempo [3] se ha venido estudiando qué modelo de consistencia es posible alcanzar en aquellos sistemas distribuidos en los que pueda haber particiones en la red y las aplicaciones toleren esas particiones. Que una aplicación tolere una situación de particionado de la red significa que continuará con sus actividades a pesar de que varios subgrupos de ordenadores hayan perdido su conectividad. Davidson et al. [3] identificaron que no se podía mantener simultáneamente una consistencia fuerte, la tolerancia a la situación de

particionado y la disponibilidad de todos los nodos correctos en los que se haya instalado la aplicación.

Años más tarde, Brewer [4] enunció como teorema el hecho de que en un sistema distribuido parcialmente sincrónico no se podrían cumplir simultáneamente estas tres propiedades (debía renunciarse al menos a una de ellas):

- Mantener una consistencia fuerte. "Fuerte" implica a los modelos de consistencia estricto o lineal. Si se obliga a que todos los procesos correctos acepten operaciones de escritura, también se puede extender a otros modelos más relajados como el secuencial.

- Disponibilidad de los servicios. Esto implica que todo nodo correcto debe seguir aceptando y ejecutando las peticiones (tanto lecturas como escrituras) que efectúen los usuarios, independientemente de cuál sea el subgrupo de nodos al que pertenezca en caso de que se divida la red.

- Tolerancia al particionado de la red. La aplicación considerada debe continuar su ejecución sin problemas a pesar de que haya particiones en la red.

Este teorema recibió el nombre de "Teorema CAP" [5], ya que estas tres propiedades podían resumirse con ese acrónimo en inglés (*CAP: Consistency, Availability and Partition-tolerance*). Podo tiempo después, Gilbert y Lynch [6] demostraron la validez del teorema.

Las tres propiedades del teorema CAP son importantes para todo sistema distribuido. Uno de los principales objetivos de un sistema distribuido es la transparencia. Entre los tipos de transparencia se encuentra la transparencia de fallos y la transparencia de replicación. La transparencia de fallos requiere que los servidores estén siempre disponibles (propiedad A del teorema CAP). A su vez, tanto la transparencia de replicación como la imagen de sistema único requieren que se mantenga una consistencia fuerte entre las réplicas de un servicio determinado (propiedad C del teorema CAP). Por último, un tipo particular de fallo en un sistema distribuido es el particionado de la red. La transparencia de fallos también obliga a que las aplicaciones con componentes replicados toleren las situaciones de particionado (propiedad P del teorema CAP). Por tanto, las tres propiedades parecen recomendables en todo sistema distribuido.

Atendiendo el enunciado del Teorema CAP, toda aplicación robusta debería renunciar a una de las tres propiedades. Veamos qué opciones hay (el orden en que se listan no es relevante):

1. Sacrificar la P (tolerancia a particiones). Esto implica que se da precedencia al mantenimiento de una consistencia fuerte y una alta disponibilidad. La única forma de asegurar esas dos propiedades sería construir el sistema de manera que jamás se dé ninguna partición.

   Esto puede intentarse replicando adecuadamente la red. Así se reducirá la probabilidad de que haya particiones. Sin embargo, siempre se podrá dar cierta secuencia de fallos en los componentes de la red que conduzcan a una partición. Una garantía absoluta de ausencia de particiones no se podrá alcanzar.

2. Sacrificar la A (disponibilidad). Esta fue la alternativa tradicionalmente adoptada en los sistemas replicados. En caso de que se diera una partición de la red se optaba por asumir el modelo de *partición primaria* [7]; es decir, sólo el subgrupo mayoritario de nodos podrá continuar, manteniendo una consistencia fuerte entre sus nodos, mientras que el resto de subgrupos tendrán que parar su actividad, renunciando a su disponibilidad.

   Esta situación será detectada por los usuarios de la aplicación distribuida que intenten utilizar los nodos de los subgrupos minoritarios. Observarán que esos nodos renuncian a contestar sus peticiones sin ningún motivo aparente.

3. Sacrificar la C (consistencia fuerte). Cuando ocurra la partición se adoptará un modelo de sistema particionable [7], por lo que se admitirán todas las operaciones en todos los nodos correctos. Por ello, al tener nodos en distintos subgrupos que no pueden comunicarse entre sí, las modificaciones recibidas en distintos servidores serán aplicadas únicamente en sus respectivos subgrupos. Así, la consistencia entre subgrupos inconexos se perderá.

   Esta opción es una de las que ha tenido mayor aceptación. Si se diseña la aplicación con cuidado y se intenta que todas sus operaciones de escritura lleguen a ser conmutativas, se perderá la consistencia fuerte, pero será relativamente sencillo adoptar un modelo de consistencia final [8] [9]. Cuando los problemas de conectividad se resuelvan, resultará muy sencillo recuperar esa consistencia, basta con transferir al resto de subgrupos las operaciones de escritura aplicadas y aplicarlas en esos otros grupos. El orden de aplicación no importará.

La alternativa que deba seleccionarse en cada sistema dependerá de sus objetivos y de las aplicaciones que deban ejecutarse en él. En los sistemas escalables se suele buscar la atención del mayor número posible de usuarios así como su satisfacción ante los servicios utilizados. En esos casos parece crítico respetar la disponibilidad. Por ello suele sacrificarse la consistencia, si la aplicación principal lo tolera.

Por su parte, los sistemas formados por unos pocos nodos podrían centrar sus esfuerzos en garantizar la conectividad entre sus máquinas. En ese caso se sacrifica la tolerancia a particiones como primera opción, recurriendo posteriormente a un modelo de partición primaria (sacrificando entonces la disponibilidad) si finalmente hubiese problemas de conectividad.

3. Replicación Multi-Máster

En el modelo de replicación multi-máster, cada cliente envía su solicitud a una sola réplica: la máster. Sin embargo, cada petición puede ser enviada a una réplica máster diferente. Esto significa que el rol máster se elige en cada solicitud y que todas las réplicas deben seguir ambos roles, dependiendo de la elección del cliente.

En cada momento podrá haber múltiples solicitudes siendo procesadas. Por ello, habrá varios másteres, cada uno sirviendo directamente un subconjunto de las peticiones recibidas.

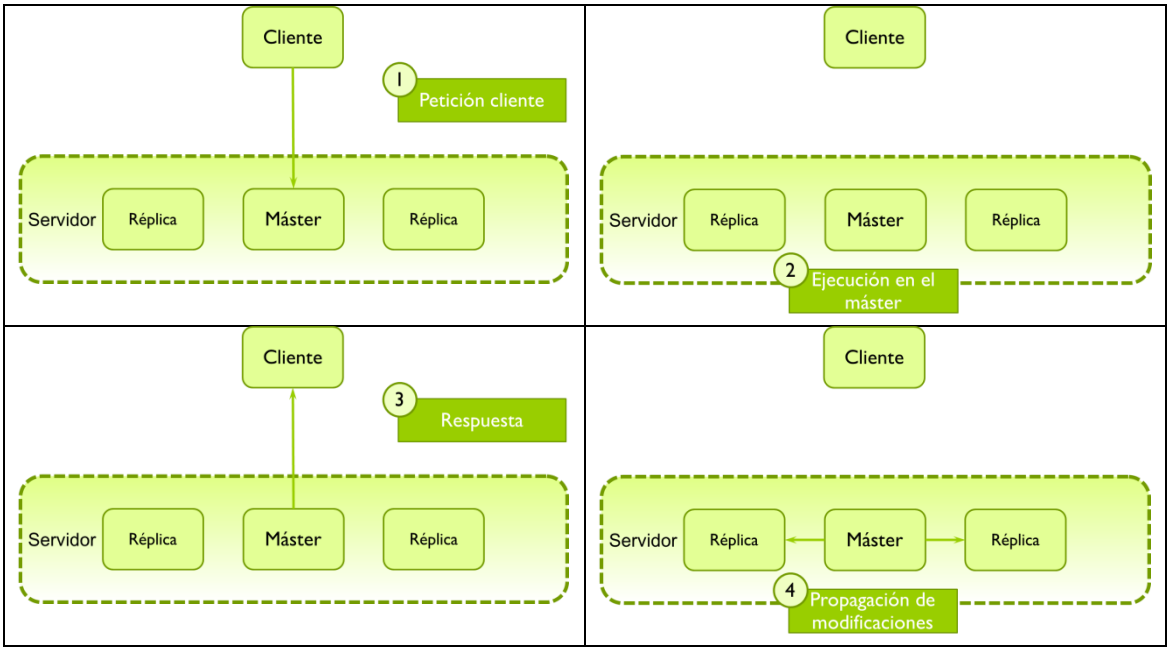

Ilustración 1. Pasos en una operación bajo el modelo multi-máster.

La Ilustración 1 muestra los pasos necesarios para ejecutar una operación en este modelo de replicación. Es similar al model pasivo pues solo una réplica procesa cada operación y propaga las modificaciones de estado a las demás réplicas. Sin embargo, hay dos diferencias entre ambos modelos:

- En la replicación multi-máster, el rol máster es elegido para cada petición (peticiones diferentes pueden ser dirigidas simultáneamente a másteres diferentes) mientras que en el modelo pasivo el rol primario es estático (todas las peticiones deben dirigirse al mismo primario, en todos los casos y para todos los clientes).
- Las modificaciones de estado se propagan de manera sincrónica en el modelo pasivo, antes de enviar la respuesta al cliente. En la replicación multi-máster, esas modificaciones se propagan de manera perezosa, una vez la respuesta ya ha sido enviada al cliente.

  Esa propagación perezosa permite una rápida respuesta a los clientes. Así, el tiempo de respuesta es más breve que en el modelo pasivo. Sin embargo, esto puede generar inconsistencias entre los estados de las réplicas, pues el modelo de consistencia resultante es muy relajado (consistencia final, si las operaciones utilizadas para propagar las modificaciones fueran conmutativas entre sí).

Ventajas

Este modelo de replicación proporciona estas ventajas:

- Sobrecarga mínima. La parte más pesada en la gestión de cada operación se asigna a la réplica máster. Las otras réplicas minimizan su carga. Esto permite que en un sistema con N nodos, N peticiones diferentes pueden ser atendidas a la vez, una en cada réplica.

- Sin control de concurrencia implícito. Múltiples operaciones pueden ser ejecutadas concurrentemente sin utilizar ningún mecanismo de sincronización. Esto se permite cuando se asume un modelo de consistencia relajado.
- Se admite la ejecución de operaciones no deterministas. Una operación se dice que es "no determinista" cuando puede generar diferentes resultados cuando múltiples ejecuciones de esa operación han utilizado los mismos argumentos de entrada.
  Como en este modelo solo la réplica máster ejecuta las operaciones, sus resultados pueden ser recibidos y aplicados sin preocupaciones por las demás réplicas.

Inconvenientes

Por otra parte, el modelo multi-máster ofrece estos inconvenientes:

- Las peticiones en curso pueden perderse cuando un máster falle. Como todos los pasos de comunicación son asincrónicos, no hay garantías acerca de la finalización del procesamiento de una solicitud. Por ejemplo, podría suceder que una petición en curso finalizara su servicio, enviándose la respuesta al cliente y las modificaciones a las demás réplicas. Tras ello, el máster podría fallar y el mensaje de respuesta podría ser perdido en el paso de recepción por parte del cliente. Posteriormente, el cliente solicitaría de nuevo esa respuesta. En ese caso, los efectos de esa petición serían conocidos por las demás réplicas.
  Sin embargo, si el máster falla antes de enviar la respuesta, esa situación sería idéntica a la anterior para el cliente. Eso implica que cuando falle el máster, las peticiones en curso podrían perderse (si no hay reintento por parte del cliente) o duplicarse (si hubiera reintento y las demás réplicas hubieran recibido y aplicado los efectos de la solicitud original).
- El modelo de fallos bizantinos no se soportaría. No hay manera de detectar cuándo las respuestas de un máster son correctas.
- La consistencia entre réplicas es muy relajada. Eso puede provocar divergencias en algunos casos. Esto debe considerarse a la hora de desarrollar las aplicaciones. Si no se hace, los usuarios pueden obtener resultados inesperados en algunos escenarios.

4. Almacenes escalables

El mayor inconveniente de los SGBD relacionales comerciales actuales es su soporte precario a las estrategias de replicación. Ninguno de esos SGBD garantiza una escalabilidad elevada: rara vez soportan más de cinco o seis réplicas y el rendimiento obtenido al incrementar el número de réplicas no llega a ser lineal. Ese rendimiento encuentra rápidamente una cota superior que no puede superar por más réplicas que se añadan. Se invierte más esfuerzo propagando las modificaciones sobre las réplicas que atendiendo las peticiones que se reciban.

A pesar de ello, los SGBD relacionales suelen ser suficientes para la gestión de los datos en entornos empresariales "cerrados". Esto es, en aquellos entornos donde el número de usuarios es estable y la cantidad de información a manejar es fácilmente previsible. Organizar la gestión de las nóminas y la contabilidad de una empresa mediana o grande no plantea problemas. Basta con renovar periódicamente los ordenadores utilizados por el SGBD relacional. La escalabilidad vertical es suficiente en este caso.

La situación cambia cuando las tecnologías de la información son utilizadas por la empresa como la base principal para su modelo de negocio. Cuando la empresa se dedique al comercio electrónico, por ejemplo, un SGBD relacional no podrá gestionar la carga que introducirán los usuarios en caso de la empresa tenga éxito.

Para romper las limitaciones propias de los SGBD relacionales en cuanto a escalabilidad horizontal, la estrategia más sencilla sería la eliminación de algunas de las propiedades ACID de las transacciones (la atomicidad y el aislamiento, por ejemplo), pues así desaparecerían las principales causas de bloqueo. De esa manera se podría aumentar el grado de concurrencia y apenas habría problemas al servir múltiples transacciones de escritura en diferentes réplicas. Sin embargo, aparecerían inconsistencias: al haber simplificado el control de concurrencia podrían haberse admitido algunas escrituras en diferentes nodos que después se llegaría a observar que estaban en conflicto. Dependiendo de la aplicación, esas inconsistencias podrían resolverse (o no) posteriormente.

Por ejemplo, si compramos un libro (en papel) en alguna librería electrónica, el sitio web utilizado puede que indique que la librería tiene suficientes ejemplares en sus almacenes, aceptando nuestra transacción de compra y el pago de ese ejemplar. Si ese título es un "éxito de ventas" puede que muchos otros usuarios hayan realizado una transacción similar al mismo tiempo. Cuando los operarios de la librería pasen a analizar las peticiones recibidas, puede que observen que sus existencias en el almacén han llegado a agotarse. La solución a ese problema no es compleja. Basta con enviar un correo electrónico a algunos de los usuarios que realizaron ese tipo de compra, informándoles de lo que ha sucedido e invitándoles a que escojan entre anular su pedido (reembolsándoles en ese caso la cantidad pagada) o esperar unos cuantos días más hasta que repongan sus existencias en sus almacenes de distribución. En la mayoría de los casos, los usuarios no se sorprenderán ante este aviso y elegirán la opción que más les convenga. Nadie sale perjudicado por esta gestión. Este problema sólo aparecerá en casos puntuales. A la librería electrónica le interesa más tener un sistema rápido y escalable (aunque impreciso en situaciones como ésta) capaz de servir de inmediato a un alto número de usuarios, que tener un sistema correcto y preciso pero limitado a un bajo número de usuarios concurrentes.

En la práctica se han propuesto dos vías para romper estas limitaciones a la escalabilidad de los SGBD relacionales [10]:

1. Utilizar los nuevos "almacenes NoSQL". Estos almacenes están basados en las siguientes características generales:
    - Se simplifica el esquema de la base de datos. En lugar de mantener "relaciones" (es decir, tablas) con un número fijo de atributos donde será posible definir claves primarias y dependencias entre las tablas del esquema (como resultado del proceso de normalización), se pasa a tener tablas con un esquema muy sencillo: una clave y un valor. En estos esquemas desaparece la posibilidad de utilizar el operador "join". También desaparece (o se reduce en gran medida) la gestión de restricciones de integridad (se mantiene la unicidad de clave primaria pero ya no se controla la integridad referencial).

- Se eliminan las transacciones o se simplifican en gran medida. La atomicidad solo se mantiene para sentencias aisladas. No resulta posible garantizar la atomicidad de una secuencia de sentencias (que era el objetivo principal de las transacciones en un SGBD relacional). En la práctica, se eliminan las propiedades ACID. Ninguna de ellas (atomicidad, consistencia semántica, aislamiento y durabilidad) llega a tener sentido con transacciones tan simples. Como algunos conflictos pueden detectarse a posteriori, algunas transacciones conflictivas deben abortarse después de haber sido confirmadas. Ni siquiera la durabilidad está garantizada en algunos casos.

  Al limitar las transacciones a sentencias únicas, los intervalos de bloqueo se limitarán a la duración de las sentencias conflictivas que estén en curso. En los SGBD relacionales, una vez la transacción obtiene un "lock" no lo libera hasta que la propia transacción termina. Estas sentencias requieren un tiempo de servicio muy breve (inferior a los 15 ms en la mayoría de los casos, siendo habitual que la ejecución se complete en 2 o 3 ms [11]), por lo que los bloqueos prolongados desaparecen y esto favorece la escalabilidad.
- Se simplifica el lenguaje de definición de datos y el lenguaje de interrogación. Se abandona el estándar SQL pues los esquemas utilizados para mantener la información no respetan el modelo relacional.

2. Utilizando una versión reformada de la arquitectura de los SGBD relacionales, tal como se propuso en [12] y se ha implantado en el SGBD comercial VoltDB [13]. Esta nueva generación de SGBD relacionales está caracterizada por:
   - Se mantiene tanto el modelo relacional como las transacciones ACID.
   - En lugar de mantener las transacciones como sentencias integradas en el flujo normal del programa, se fuerza a que todas las transacciones se implanten mediante procedimientos almacenados. De esta manera todo el trabajo a realizar por una transacción se completará con una sola invocación. No habrá múltiples cambios de control entre el SGBD y la aplicación que implican cambios de contexto y, en la mayoría de los casos, envío y recepción de mensajes entre el SGBD y la aplicación en caso de aplicaciones distribuidas.
   - En lugar de mantenerla en disco, la base de datos se particiona (se distribuye a múltiples nodos y, además, cada "fragmento" se replica en varias máquinas) y se mantiene en memoria principal. Como los datos están en memoria y la gestión de la transacción se realiza mediante procedimientos almacenados, la ejecución de una transacción puede completarse en pocos microsegundos.
   - No se utilizan mecanismos de control de concurrencia. En caso de recibir múltiples invocaciones (que deberían generar varias transacciones), las solicitudes se dejan en una cola de entrada y se ejecutan en estricto orden secuencial. Como en cada transacción se invierten pocos microsegundos, los intervalos de espera rara vez llegan a un milisegundo, aunque haya que soportar una carga muy alta.
   - La persistencia de los datos está garantizada por la replicación. Mientras no fallen todas las réplicas de cada dato se asegura que esa información se mantendrá.

Dentro de los almacenes NoSQL se pueden distinguir algunas variantes [10] que se describen seguidamente: almacenes clave-valor, almacenes de documentos y almacenes de registros extensibles.

4.1. Almacenes clave-valor

En esta variante se simplifica el esquema de cada tabla a su mínima expresión: sólo se distingue una clave y un atributo o valor. Las búsquedas de información se realizan en base al campo clave, sobre el que se habrá construido un índice para encontrar de inmediato la referencia buscada. El segundo campo no llega a ser estructurado por el SGBD. Para el SGBD es un valor único que el SGBD es incapaz de interpretar, por lo que generalmente no se admiten modificaciones parciales. Además, en caso de que se quieran distinguir varios atributos en ese único campo, tendrá que ser la aplicación la responsable de gestionar tal estructuración.

Es la variante más cómoda para el SGBD pues obliga a realizar una cantidad mínima de trabajo para gestionar los accesos. El lenguaje de interrogación es muy sencillo.

Algunos ejemplos de almacenes de este tipo son: Amazon Dynamo, Voldemort, Riak…

4.2. Almacenes de documentos

En este caso, los elementos que se guardan en la base de datos reciben el nombre de documentos (aunque pueden considerarse objetos, con cierta secuencia de atributos) y ya ofrecen una estructura que el SGBD debe ser capaz de interpretar y gestionar adecuadamente. Cada documento tendrá un campo clave (que suele llamarse “identificador de documento”) y una secuencia de atributos. Algunos de esos atributos pueden ser también otros objetos con atributos internos. Esta estructura puede anidarse tantas veces como sea necesario. El esquema de una colección (“colección” es el término utilizado en esta variante para referirse a una tabla) no es fijo como en el modelo relacional. Es decir, en una misma colección podrá haber documentos con estructura similar pero no necesariamente idéntica. Así, algunos documentos de la colección pueden tener algunos campos más que los demás o haber llamado con diferente nombre a atributos similares. Por tanto, el esquema de una colección es mucho más flexible que en el modelo relacional. Esto no siempre reportará ventajas. Depende del programador el que la estructura de las colecciones de documentos sea uniforme o no.

El lenguaje de interrogación se basa en establecer condiciones que deberán cumplir los documentos que se retornen como resultado de cada consulta. Se puede conseguir una funcionalidad similar a SQL pero la sintaxis utilizada suele ser distinta.

Para equilibrar la carga entre múltiples SGBD, se puede realizar un particionado horizontal de la base de datos. Esto es, el contenido de una misma colección puede repartirse entre múltiples nodos gestores en función del identificador de cada documento. Cada gestor será responsable de un rango de identificadores. Periódicamente se ejecuta un equilibrador de carga que comprueba el número de documentos que hay en cada servidor. Cuando las diferencias son grandes, se modifican los rangos asignados para que la carga vuelva a equilibrarse.

Algunos ejemplos de sistemas de este tipo son: CouchDB, MongoDB, SimpleDB…

4.3. Almacenes de registros extensibles

Los almacenes de registros extensibles ofrecen el modelo más parecido al relacional. Las bases se organizan en tablas. Sin embargo, al igual que ocurría con las colecciones de documentos en el apartado anterior, las tablas de estos almacenes tienen un número variable de columnas. A su vez, el conjunto de columnas puede organizarse en grupos de columnas.

Otra mejora respecto a los almacenes de documentos consiste en la posibilidad de añadir particionado vertical al particionado horizontal ya permitido en el caso anterior. Particionar verticalmente significa que una determinada tabla puede dividirse en varias subtablas que recogen algunas de las columnas de la tabla original.

La posibilidad de particionado mejora las prestaciones. Cada nodo es responsable de una pequeña parte de la tabla particionada. El particionado debe organizarse cuidadosamente una vez se han examinado las consultas típicas que necesitarán las aplicaciones que utilicen esa base de datos. El objetivo es que las consultas más frecuentes obtengan toda la información necesaria desde una única subtabla, respecto al particionado vertical. Así, múltiples consultas podrán ser atendidas simultáneamente por diferentes nodos, cada uno responsable de una subtabla distinta. De esta manera aumenta el rendimiento y mejora la escalabilidad.

Esto puede complementarse con el particionado horizontal. Si se emplea este segundo particionado, una misma consulta se vuelve a repartir entre múltiples nodos, de manera que cada uno de ellos debe retornar los registros que satisfagan la condición utilizada en la consulta. Así se reparte el trabajo de transferencia.

Algunos ejemplos de sistemas de este tipo son: Google Bigtable, Yahoo PNUTs, Apache Cassandra…

5. Elasticidad

Según [14], la elasticidad es "*el grado en que un sistema es capaz de adaptarse a cambios en su carga suministrando y reciclando los recursos de una manera autónoma, consiguiendo que en cada momento los recursos disponibles cubran la demanda existente con la mayor precisión posible*". Veamos qué implicaciones tiene esa definición:

1. El sistema distribuido resultante debe ser escalable, pues esta definición indica que debe ser capaz de atender cargas variables. Por tanto, si la carga crece, el sistema tendrá que incrementar su número de nodos para soportar esa carga creciente.
2. Pero no debe ser sólo escalable. También debe ser un sistema dinámico y adaptable. Cuando la carga disminuya se tendrán que liberar los nodos que no resulten necesarios. Esto implica que cuando ninguna otra aplicación los necesite, esas máquinas (virtuales o físicas) se podrán apagar y con ello reducir el consumo energético del sistema.
3. Esta adaptabilidad debe ser autónoma. El sistema debe ser capaz de monitorizar la carga y su propio rendimiento. En función de los niveles detectados debe decidir (sin necesidad de que intervenga ningún administrador) qué cambios deben realizarse.

4. Por último, los cambios efectuados deben ser precisos, tanto en cantidad (de recursos a incorporar o a parar) como en tiempo. La reacción del sistema frente a cambios en la demanda o carga soportada deben ser rápidos y eficientes.

La elasticidad es una característica exigible a los sistemas *cloud*, regidos por un modelo de pago por uso. Para el usuario, la elasticidad permitirá una mayor precisión en el cómputo de los recursos utilizados. De esta manera se pagará "lo justo". Para el proveedor de estos servicios, la elasticidad permitirá suministrar la cantidad suficiente de recursos a cada usuario. Así se evita la sobreasignación (que el usuario no querrá pagar), reduciendo también el consumo energético de los centros de datos utilizados.

Para implantar un sistema elástico se necesitará:

a) Un mecanismo de monitorización adecuado, que evalúe la carga soportada por cada uno de los recursos que formen el sistema, así como el rendimiento de los módulos que se hayan instalado en el sistema.
b) En función de los índices obtenidos se tendrá que decidir si resulta necesario reconfigurar el conjunto de recursos disponibles y cómo hacerlo. En el caso de sistemas cloud IaaS o PaaS, estas decisiones dependerán del "*Service Level Agreement*" (SLA) que se haya establecido entre el proveedor y el usuario. Un SLA especifica los niveles de rendimiento y disponibilidad que espera obtener el usuario y que se compromete a ofrecer el proveedor, así como los niveles de carga que deberán admitirse.
c) Un mecanismo de reacción que ajuste los recursos utilizados a los índices obtenidos por el mecanismo de monitorización. Este mecanismo debe ser diseñado e implantado por el proveedor.

6. Contención

Cualquier sistema distribuido escalable estará formado por múltiples componentes que serán desplegados en un conjunto de nodos. Estos componentes serán eficientes y utilizarán los mecanismos básicos para obtener escalabilidad que han sido mencionados en este tema.

Las dependencias entre estos componentes así como sus patrones de uso y carga soportada deben considerarse cuidadosamente tanto en la etapa de diseño de la aplicación como en los primeros intervalos de su uso. Es posible que durante la monitorización del sistema se advierta que alguno de los componentes ofrece un "cuello de botella" que limita el rendimiento global de la aplicación. Esto se debe generalmente a que "bloquea" el flujo de información de la aplicación por encontrarse ese componente saturado ante la carga que introducen los demás.

Las causas de este tipo de contención suelen ser:

1. Uso de algoritmos centralizados para realizar tareas "pesadas". El uso de un proceso coordinador solo tiene sentido para ejecutar tareas ligeras o infrecuentes.
2. Uso de herramientas de sincronización. Cuando múltiples actividades utilicen un mismo recurso compartido que requiera acceso exclusivo habrá una sección crítica y ésta deberá protegerse adecuadamente. Cualquier solución correcta para el problema de la sección crítica conduce a la suspensión de aquellas actividades que encuentren la

sección ocupada. Por otra parte, los mecanismos de comunicación basados en intercambio de mensajes también provocan la suspensión del emisor si esta comunicación es sincrónica. A su vez, el receptor también se suspende hasta que llegue el mensaje que pretendía recibir. Todas estas situaciones conducen al bloqueo de alguna actividad, limitando el progreso del algoritmo utilizado.

3. Tráfico excesivo. Una mala distribución de los componentes de la aplicación puede conducir a que estos componentes utilicen recursos ubicados en otros nodos, en lugar de tenerlos disponibles en su propio nodo. Eso conducirá a una elevada utilización de mensajes para acceder a los recursos, ralentizando este acceso.

Las formas de evitar esta contención en función de su causa serían:

1. Gestionar las tareas pesadas y frecuentes utilizando algoritmos descentralizados. Con ello se reparte la carga entre múltiples procesos.
2. Cuando la sincronización se deba al acceso a recursos compartidos, la solución más sencilla consiste en limitar el grado de concurrencia. En lugar de admitir múltiples actividades en un mismo proceso, limitarlas a una sola. Con ello se evitarán los cambios de contexto y se agilizará el servicio de cada petición. Para que ese único hilo no se sature se debe optimizar también su gestión.

   Por ejemplo, tradicionalmente el acceso a los datos persistentes en una aplicación se realiza utilizando un sistema gestor de bases de datos (SGBD) relacional. Los datos se mantienen en almacenamiento secundario. Se utilizan "locks" para realizar el control de concurrencia. Se admiten múltiples conexiones simultáneas, cada una gestionada por uno o más hilos. Cada cambio realizado por una transacción se anota también en un "log" mantenido el disco, para recuperar la transacción en caso de fallo una vez se hubiese aceptado su confirmación. En algunos sistemas recientes [12] [13] se ha demostrado que la gestión es mucho más eficiente si los datos se mantienen directamente en RAM, replicando la información en varios nodos. Cada nodo solo utiliza un hilo. Las transacciones se implantan como procedimientos almacenados. El resultado es que una transacción entera bajo este nuevo modelo tarda unas cien veces menos en ser finalizada. La gestión puede ser secuencial, sin concurrencia. El tiempo medio de respuesta percibido por el usuario es inferior al de un sistema tradicional. El sistema resultante es mucho más escalable que cualquiera de los "tradicionales".
3. Si hay excesivo tráfico en la red, interesará replicar los recursos utilizados. De esta manera cada componente tendrá una copia local de cada recurso y no necesitará utilizar mensajes para acceder a ellos.

7. Ejemplos de escalabilidad

Este apartado facilita una descripción general (o, alternativamente, punteros) sobre tres ejemplos relacionados con la escalabilidad en sistemas distribuidos.

Para empezar, se presenta el módulo "cluster" de NodeJS. Como los procesos NodeJS solo tienen un hilo de ejecución, los servidores NodeJS necesitarían gestionar múltiples procesos para soportar concurrencia en la gestión de las solicitudes que reciban. El módulo "cluster"

posibilita esa gestión concurrente. Para ello, el proceso inicial (o proceso "maestro") puede crear un conjunto de procesos trabajadores. Todos esos procesos compartirán uno o más puertos TCP. Así cada petición podrá ser entregada un proceso trabajador diferente, permitiendo que esas peticiones puedan ser atendidas concurrentemente por varios trabajadores.

El segundo ejemplo presenta varios componentes (concretamente, proxies inversos) que mejoran la escalabilidad de un servicio cuando este necesita ser desplegado en varios ordenadores.

Por último, el tercer ejemplo describe la arquitectura de un almacén NoSQL: MongoDB. Ese almacén escalable utiliza varios mecanismos descritos en este tema: un esquema NoSQL, replicación, distribución de datos, consistencia configurable…

Los próximos apartados proporcionan algunos punteros a la documentación oficial de las herramientas mencionadas en los párrafos anteriores. No se necesita consultar esa documentación en detalle. De hecho, las presentaciones realizadas en el aula deberían ser suficientes para entender los aspectos más relevantes de estos ejemplos. Nuestro objetivo es mostrar que los mecanismos descritos en este tema se utilizan en sistemas reales, pero no será necesario utilizar estas herramientas en esta asignatura. No habrá tiempo suficiente para ello.

7.1. El módulo "cluster" de NodeJS

Los contenidos utilizados en la presentación realizada sobre este módulo en el aula pueden extenderse siguiendo estos punteros:

- Documentación oficial: https://nodejs.org/dist/latest-v8.x/docs/api/cluster.html
- Tutorial breve (en inglés): https://www.sitepoint.com/how-to-create-a-node-js-cluster-for-speeding-up-your-apps/

7.2. Escalabilidad usando múltiples ordenadores

Cuando la capacidad de servicio de un ordenador se sature debido a la carga recibida, se necesita desplegar ese servicio de otra manera: utilizando varios ordenadores en su lugar. Para ello, se utilizarán técnicas de escalado horizontal. Eso significa que en lugar de un único servidor, se tendrán que desplegar múltiples instancias de ese servidor, una por ordenador. Para acceder de manera transparente a esos servidores se necesita otro elemento: un proxy inverso. El proxy inverso proporciona un punto de entrada único para el servicio, donde recibe las peticiones de todos los clientes. Utilizando una estrategia adecuada, reenvía las solicitudes a las instancias servidoras. A continuación, esas solicitudes son procesadas y respondidas por la instancia correspondiente. Tras ello, el proxy inverso retornará cada respuesta al cliente correspondiente.

La política de reenvío utilizada por ese proxy cumple dos objetivos complementarios: (1) equilibrar la carga entre todos los servidores, y (2) mejorar la escalabilidad (pues la capacidad de servicio se incrementa al utilizar múltiples servidores).

La existencia del proxy resulta transparente para los clientes. Esos clientes observan un único punto de entrada (el del proxy inverso) que oculta la ubicación real de cada proceso servidor. Así, la existencia de múltiples ordenadores servidores también se oculta a los usuarios.

No es necesario que analicemos en detalle las diferentes opciones de configuración para los tres proxies mencionados en la presentación de este tema. Solo nos interesa el concepto de proxy inverso: un elemento que permite lograr escalabilidad horizontal de manera transparente para los usuarios. No obstante, sí que es importante que se identifique a esos tres ejemplos de componentes software como proxies inversos. Se puede encontrar información adicional sobre ellos en sus respectivos sitios web oficiales:

- Node-http-proxy: https://github.com/nodejitsu/node-http-proxy
- HAproxy: http://www.haproxy.org/
- Nginx: https://www.nginx.com/

Además del equilibrado de carga, los proxies inversos también mejoran los siguientes aspectos relacionados con la escalabilidad de un servicio:

1. Un proxy inverso también puede configurarse como un cortafuegos, mejorando así la seguridad del servicio. El cortafuegos controla los puertos asociados al servicio y filtra las solicitudes recibidas. Ese filtrado rechaza aquellos mensajes entrantes que no estén relacionados con el servicio facilitado (su admisión entrañaría peligro y, aparte, sobrecargaría innecesariamente a los servidores).
2. Si el proxy dispone de algún hardware acelerador para TLS/SSL, podrá gestionar las tareas relacionadas con el establecimiento y uso de conexiones seguras. Así, los servidores reales no necesitan involucrarse en ese tipo de tareas, liberándose de esa carga y mejorando su rendimiento.
3. El proxy inverso puede actuar como caché para el contenido solicitado por los clientes (por ejemplo, para páginas web estáticas). Así, gran parte de las solicitudes pueden ser respondidas directamente por el proxy cuando el contenido esté en la caché. Eso reduce la carga que gestionarán los servidores reales.
4. El proxy inverso puede comprimir las respuestas proporcionadas a los clientes (por ejemplo, en caso de páginas HTML de gran tamaño o con un buen número de recursos asociados). Eso reduce el tamaño de esas respuestas y acelera su recepción por parte de los clientes.
5. La autenticación de los clientes puede ser gestionada por el proxy inverso en caso de que el servicio requiera autenticación. Así, los servidores reales no necesitan gestionar esta tarea.

Estas mejoras solo serán posibles si el proxy dispone de suficientes recursos. En los cinco casos mencionados se necesita una configuración especial en su máquina. Por ejemplo, el filtrado de las peticiones (aspecto 1) impone una alta carga de CPU, para comparar algunos campos del mensaje entrante con unos valores determinados. El aspecto 3 (gestión de caché) exige que el nodo disponga de mucha memoria RAM. El aspecto 2 requiere un hardware específico que gestiona eficientemente el cifrado. En todos estos casos, el proxy introduce un punto central de gestión. Para mejorar la escalabilidad, ese elemento central no debe ser un punto de contención.

7.3. MongoDB

La documentación oficial sobre MongoDB 3.6 es un manual impreso de 7228 páginas. Está disponible on-line en https://docs.mongodb.com/manual/. Las secciones descritas en la presentación están explicadas en varios capítulos del manual. No es necesario consultar esos capítulos enteros. En su lugar, bastará con revisar los fragmentos que indicamos a continuación:

- Arquitectura y componentes de MongoDB: (basta con leer la página inicial; el capítulo entero es excesivamente largo) https://docs.mongodb.com/manual/sharding/
- Replicación en MongoDB: https://docs.mongodb.com/manual/replication/. De nuevo, la página inicial explica de una manera general (pero suficiente) los conceptos principales de la estrategia de replicación utilizada en MongoDB. Básicamente, se sigue el modelo de replicación pasivo, pero utilizando una propagación de modificaciones asincrónica hacia las réplicas secundarias. Esto relaja la consistencia entre réplicas (con lo que mejora la escalabilidad), pero podría conducir a pérdida de datos según qué valor se seleccione para la sincronización de escrituras ("write concern").

Esos dos apartados proporcionan una descripción concisa y suficiente de los principales mecanismos de escalabilidad utilizados en MongoDB (replicación pasiva asincrónica y escalabilidad horizontal mediante "sharding"). La escalabilidad también depende de la consistencia. En MongoDB, su gestión de fallos y su nivel de consistencia pueden configurarse empleando dos mecanismos complementarios: (1) la sincronización de escrituras ("write concern") y (2) el aislamiento de sus lecturas ("read concern"). Opcionalmente, se puede revisar la documentación sobre esos dos aspectos en estos URL:

- Write concern: https://docs.mongodb.com/manual/reference/write-concern/
- Read concern: https://docs.mongodb.com/manual/reference/read-concern/

Referencias

[1] B. C. Neuman, «Scale in Distributed Systems,» de *Readings in Distributed Computing Systems*, IEEE-CS Press, 1994, pp. 463-489.

[2] M. M. Michael, J. E. Moreira, D. Shiloach y R. W. Wisniewski, «Scale-up x Scale-out: A Case Study using Nutch/Lucene,» de *21th International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, EE.UU., 2007.

[3] S. B. Davidson, H. García-Molina y D. Skeen, «Consistency in Partitioned Networks,» *ACM Comput. Surv.,* vol. 17, nº 3, pp. 341-370, 1985.

[4] E. Brewer, «Towards Robust Distributed Systems,» de *19th Annual Symposium on Principles of Distribruted Computing (PODC)*, ACM Press, 2000, pp. 7-10.

[5] E. Brewer, «CAP Twelve Years Later: How the 'Rules' Have Changed,» *IEEE Computer,* vol. 45, nº 2, pp. 23-29, 2012.

[6] S. Gilbert y N. Lynch, «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,» *SIGACT News,* vol. 33, nº 2, pp. 51-59, 2002.

[7] G. Chockler, I. Keidar y R. Vitenberg, «Group Communication Specifications: A Comprehensive Study,» *ACM Comput. Surv.,* vol. 33, nº 4, pp. 427-469, 2001.

[8] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer y B. B. Welch, «Session Guarantees for Weakly Consistent Replicated Data,» de *3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, Austin, Texas, EE.UU., IEEE-CS Press, 1994, pp. 140-149.

[9] W. Vogels, «Eventually Consistent,» *Commun. ACM,* vol. 52, nº 1, pp. 40-44, 2009.

[10] M. Stonebraker y R. Cattell, «10 Rules for Scalable Performance in 'Simple Operation' Datastores,» *Commun. ACM,* vol. 54, nº 6, pp. 72-80, 2011.

[11] MongoDB Documentation Project, «MongoDB Documentation, Release 2.4.5,» 10gen, Inc., 2013.

[12] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem y P. Helland, «The End of an Architectural Era (It's Time for a Complete Rewrite),» de *33rd International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, ACM, 2007, pp. 1150-1160.

[13] M. Stonebraker y A. Weisberg, «The VoltDB Main Memory DBMS,» *IEEE Data Eng. Bull.,* vol. 36, nº 2, pp. 21-27, 2013.

[14] N. R. Herbst, S. Kounev y R. Reussner, «Elasticity in Cloud Computing: What It Is and What It Is Not,» de *10th International Conference on Autonomic Computing (ICAC)*, San Jose, CA, EE.UU., USENIX, 2013, pp. 24-28.