Diseño y Arquitectura de Servicios Escalables

Unidad 4
Descentralización y asincronía

Índice

Bibliografía

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber: “Bigtable: A Distributed Storage System for Structured Data”. *ACM Trans. Comput. Syst.* 26(2) (2008)

[DG08] Jeffrey Dean, Sanjay Ghemawat: “MapReduce: simplified data processing on large clusters”. *Commun. ACM* 51(1): 107-113 (2008)

[EFG+03] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, Anne-Marie Kermarrec: “The many faces of publish/subscribe”. *ACM Comput. Surv*. 35(2): 114-131 (2003)

[Eug07] Patrick Th. Eugster: “Type-based publish/subscribe: Concepts and experiences”. *ACM Trans. Program. Lang. Syst.* 29(1) (2007)

[OPS+93] Brian M. Oki, Manfred Pflügl, Alex Siegel, Dale Skeen: “The Information Bus - An Architecture for Extensible Distributed Systems”. SOSP 1993: 58-68

[SC11] Michael Stonebraker, Rick Cattell: “10 rules for scalable performance in 'simple operation' datastores”. *Commun. ACM* 54(6): 72-80 (2011)

Índice

1. Introducción

- Para que un servicio sea escalable…
  - Hay que minimizar sus necesidades de sincronización
  - ¿Cómo lograrlo?
    - Mediantes algoritmos descentralizados.
    - Mediante comunicación asincrónica.
    - Mediante almacenes persistentes NoSQL.
      - Donde es más fácil aplicar “sharding”

Índice

2. Algoritmos descentralizados

- Si los algoritmos utilizados para implantar un determinado servicio fueran centralizados:
  - La toma de decisiones sólo se realizaría en un nodo.
  - Sería relativamente fácil desbordar la capacidad de ese nodo.
  - Si fallara ese proceso, los demás no podrían continuar mientras no se reconfigurara el servicio:
    - Debería seleccionarse un nuevo “coordinador”.
    - Debería regenerarse todo el estado que mantenía el coordinador previo antes de fallar.
    - Esto puede requerir un tiempo considerable.

2. Algoritmos descentralizados

- Para resolver esos problemas hay que utilizar “algoritmos descentralizados”.
- Un algoritmo descentralizado cumple estas cuatro propiedades:
  - P1: Ningún proceso mantiene la información completa que necesite el sistema o la aplicación.
  - P2: Los procesos toman sus decisiones utilizando únicamente información local.
  - P3: El fallo de un proceso no compromete el progreso del algoritmo.
  - P4: No se asume la existencia de un reloj global que todos los procesos puedan compartir.

2. Algoritmos descentralizados

- Veamos qué implica cada propiedad:
  - P1: Ningún proceso mantiene la información completa que necesite el sistema o la aplicación.
    - El estado global debe dividirse entre todos los procesos participantes: reparto o particionado del estado.
    - Cada uno sólo es responsable de la parte que le ha sido asignada.
    - Se evitan problemas de saturación.
    - Al incorporar más procesos no deberíamos reestructurar el “reparto”. Cada proceso es responsable de una parte de la gestión de manera “natural”, sin redistribuciones.
    - Es la solución habitualmente empleada en los sistemas P2P estructurados para implantar su servicio de directorio.
    - MapReduce [DG08] también la cumple.

2. Algoritmos descentralizados

- P2: Los procesos toman sus decisiones utilizando únicamente información local.
  - No pueden utilizarse protocolos bloqueantes para la toma de decisiones.
    - El consenso necesita un algoritmo bloqueante.
  - Cuando se llegue a un paso determinado del algoritmo donde deba tomarse alguna decisión, el proceso no debe bloquearse esperando mensajes de otros procesos.
  - Las decisiones se toman en base a la información local.
    - Ésta puede haberse obtenido mediante mensajes recibidos previamente.

2. Algoritmos descentralizados

- P3: El fallo de un proceso no compromete el progreso del algoritmo.
  - Aunque algún proceso falle, los demás podrán continuar sin bloquearse.
  - Si algún proceso fuera responsable de alguna tarea que no pueda ser interrumpida, habrá que replicarlo.
  - P3 puede considerarse una consecuencia de P2.

2. Algoritmos descentralizados

- P4: No se asume la existencia de un reloj global que todos los procesos puedan compartir.
  - Las decisiones no deben depender del transcurso del tiempo.
  - Ni del grado de sincronización entre los relojes de cada proceso.
  - Propiedad implícita en un sistema asincrónico.
  - La sincronía compromete la escalabilidad.
  - Los algoritmos escalables deben ser asincrónicos.

2. Algoritmos descentralizados

- ¿Interesa siempre la descentralización?
  - No. Hay ciertos casos en que tomar alguna decisión implica sincronizar a todos los procesos.
  - Una sincronización “descentralizada” no es eficiente.
  - Es mejor delegar esos pasos en un coordinador.
    - Criterio determinista para seleccionarlo, que no requiera intercambiar mensajes.

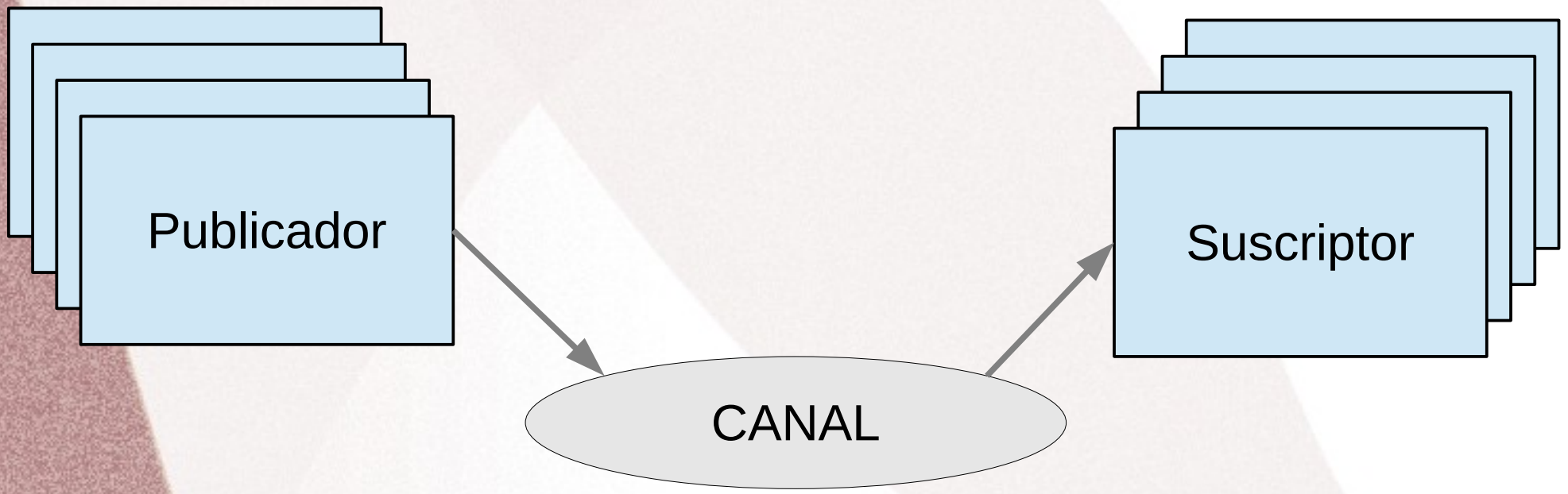
Índice

3. Comunicación “publish-subscribe”

- El modelo “publish-subscribe” [OPS+93] utiliza comunicación asincrónica. Además:
  - El *middleware* de comunicaciones se comporta como un bus compartido por publicadores y suscriptores.
  - La comunicación es anónima.
  - La comunicación puede ser persistente.
  - La comunicación es “uno a muchos” o “muchos a muchos”.
  - Los suscriptores seleccionan sus “asuntos” de interés.

3. Comunicación “publish-subscribe”

- Modelo de comunicación



3. Comunicación “publish-subscribe”

- La comunicación está “mediada”
  - Habrá un middleware que gestione todos los posibles “canales”
    - Un canal por “asunto”
    - La publicación deja cada mensaje en su canal (según el “asunto” utilizado)
  - El middleware mantendrá los mensajes (o no) pero...
    - El publicador no necesita conocer a los suscriptores
    - El middleware gestionará la entrega a los suscriptores
    - Comunicación asíncrona para el publicador

3. Comunicación “publish-subscribe”

- Al emplear comunicación anónima:
  - El conjunto concreto de destinatarios de cada mensaje es desconocido por el publicador.
    - Para gestionar los fallos de los suscriptores, convendría tenerlos replicados.
    - No es relevante para el publicador.
  - Los procesos publicador y suscriptor pueden ser actualizados sin problemas.
    - Gestión más sencilla de su ciclo de vida.

3. Comunicación “publish-subscribe”

- ¿Por qué mejora la escalabilidad?
  - Se utiliza comunicación asincrónica.
    - Los publicadores no se bloquean.
  - No hay un límite preestablecido sobre el número de suscriptores en un conector.
    - No se monitoriza su estado.
    - Escalabilidad de tamaño.
  - No hay garantías explícitas en el orden de entrega de los mensajes.
    - Comunicación eficiente, por ser sencilla.

3. Comunicación “publish-subscribe”

- ¿Qué problemas comporta?
  - Consistencia relajada:
    - Si hay múltiples publicadores, cada suscriptor puede ordenar sus mensajes de manera diferente.
    - Puede proporcionarse consistencia FIFO si cada publicador numera sus mensajes secuencialmente, pero...
      - Necesitamos otro conector auxiliar para solicitar un reenvío de los mensajes “perdidos”.
      - No se pueden implantar consistencias más estrictas de manera sencilla.

3. Comunicación “publish-subscribe”

- El modelo de comunicaciones “publish-subscribe” admite diferentes tipos de gestión: basado en tópicos o asuntos [EFG+03], basado en contenidos [EFG+03], tipificado [Eug07]...
- Para desarrollar servicios escalables basta con utilizar sus principios de interacción.
  - El “asunto” al que suscribirse será el “servicio” de interés.
  - No entraremos en detalle sobre las distintas variantes de gestión que se dan en este modelo de comunicación.

3. Comunicación "publish-subscribe"

- Resumen:
  - Interesa el patrón "*publish-suscribe*" por:
    - Asincronía.
    - Es unidireccional.
    - Implanta difusión.
  - Dificultades:
    - En muchos casos, comunicación no fiable.
    - Sin orden garantizado.

3. Comunicación "publish-subscribe"

- Conclusiones:
  - Es crítico encontrar un patrón de comunicaciones asincrónico.
  - Interesan módulos de comunicación de este tipo, como 0MQ
    - http://www.zeromq.org
  - Si no se necesita difusión, existen otros patrones asincrónicos:
    - Push-pull.

Índice

4. Distribución de datos

- ¿Cómo mantener la información persistente?
  - Tradicionalmente se ha resuelto mediante SGBD relacionales, pero...
    - Los accesos se realizan utilizando transacciones con garantías ACID.
    - Utilizan mecanismos de control de concurrencia relativamente pesados.
    - Esos mecanismos introducen bloqueos en las actividades.
    - No todas las aplicaciones “escalables” se adaptan bien a un modelo relacional.

4. Distribución de datos

- Por tanto...
  - Hay que buscar modelos de datos más “ligeros” que el relacional.
  - En algunos casos sin soporte transaccional.
  - Que puedan ser suficientemente flexibles para reflejar cualquier esquema/estructura de datos.
  - Que admitan “*sharding*”.
    - Véase tema 3 y sección 4.1.

4. Distribución de datos

- Las primeras soluciones se centraron en sustituir los esquemas de BD relacionales por bases clave-valor, renunciando a las transacciones.
- Posteriormente, el “valor” se pudo estructurar de diferentes maneras (incluso entre diferentes filas de una misma “tabla”). Ejemplo: Google Bigtable [CDG+08].
- Secuencias contiguas de filas se mantienen en diferentes servidores, repartiendo la carga de gestión entre ellos.
  - Fácil adaptación al modelo de programación MapReduce.

4. Distribución de datos

- Para mejorar la escalabilidad, recientemente se han propuesto varios tipos de almacenes persistentes no relacionales [SC11]:
  - 1) Almacenes clave-valor:
    - Colecciones de “objetos”. Cada “objeto” (o valor) asociado a una clave diferente.
    - No permiten modelar la estructura de esos objetos.
    - El lenguaje de interrogación no permite acceder a los atributos de cada objeto: se obtiene el “objeto” completo.
    - Ejemplos: Dynamo, Voldemort, Riak...

4. Distribución de datos

- 2) Almacenes de documentos:
  - El modelo de datos utiliza también objetos, pero en este caso sí que refleja sus atributos.
  - En la mayoría de los almacenes se pueden anidar objetos.
  - El lenguaje de interrogación admite especificar restricciones sobre el valor de múltiples atributos.
    - También puede ser “procedural”.
  - Ejemplos: CouchDB, MongoDB, SimpleDB...

4. Distribución de datos

- 3) Almacenes de registros extensibles:
  - Cada registro puede tener un número variable de “columnas” (atributos).
  - El lenguaje de interrogación es más complejo que en los casos anteriores, pero no es SQL.
  - Los datos se pueden repartir entre los nodos con particionado vertical, horizontal o ambos.
  - Ejemplos: Bigtable, PNUTs, HBase ...

4. Distribución de datos

- [SC11] proponen VoltDB:
  - Modelo relacional. Garantías ACID. SQL.
  - Los datos se mantienen en memoria principal, repartidos entre los diferentes nodos.
  - Se eliminan los logs.
  - Se elimina la concurrencia y su gestión.
  - Ejecución estrictamente secuencial de las transacciones.
  - Transacciones soportadas mediante procedimientos almacenados.
  - Replicación (aunque no aclaran cómo ni cuándo propagar las escrituras).
  - Eficiente y escalable.

4. Distribución de datos

- Stonebraker y Cattel [SC11] proponen diez “consejos” para garantizar la escalabilidad en la gestión de datos persistentes.
  - 1. Particionar el estado entre los nodos.
    - Para reducir las necesidades de sincronización e interacción entre nodos, los datos deben dividirse en conjuntos disjuntos y asignarse a diferentes nodos.
    - Así se mejora la paralelización del acceso a datos y se incrementa el rendimiento.
    - Problemas:
      - a) No es sencillo realizar un particionado perfecto.
      - b) En el modelo relacional, los “joins” serán costosos.

4. Distribución de datos

- 2. Utilizar lenguajes de alto nivel no perjudicará el rendimiento.
  - En su momento, SQL (lenguaje declarativo utilizado en el modelo relacional) supuso un avance frente a los modelos jerárquicos o en red, más orientados a la ubicación real de los datos y a la implantación del mecanismo de acceso.
  - Los SGBD relacionales ofrecen buen rendimiento cuando se utilizan “procedimientos almacenados” para implantar las transacciones.
    - Minimiza los cambios de contexto entre gestor y aplicación.
  - Los sistemas de almacenamiento escalables deberían evitar el uso de lenguajes de bajo nivel, dependientes del esquema que imponga cierta estructura a los datos.

4. Distribución de datos

- 3. Implantar la BD en memoria principal.
  - El soporte de los datos en disco introduce penalizaciones grandes sobre el rendimiento.
  - En ciertos experimentos se comprobó que un SGBD relacional descompone su tiempo de ejecución de la siguiente manera:
    - a) Trabajo útil: 13%.
    - b) Control de concurrencia: 20%.
    - c) “Logging” (recuperabilidad): 23%.
    - d) Gestión de cachés y buffers: 33%.
    - e) Multithreading: 11%.
  - Los apartados b), c), d) y e) pueden eliminarse si la BD se implanta en RAM y las transacciones se ejecutan de manera secuencial, sin hilos.

4. Distribución de datos

- 4. La replicación y la recuperación automatizada son esenciales para garantizar la escalabilidad.
  - Se necesita replicación para sobreponerse a las situaciones de fallo en alguna de las máquinas o en la red.
    - Pero las situaciones de particionado de la red no siempre pueden superarse. Ver teorema CAP.
  - Si el sistema es escalable habrá muchas máquinas y no tiene sentido esperar que la intervención de un operador restaure un nodo caído.
    - Debe instanciarse una nueva réplica y transferir el estado de manera automática.

4. Distribución de datos

- 5. Siempre “on-line”.
  - Aparte de los fallos, hay otras situaciones que impiden que el acceso a los datos sea posible:
    - Cambios en el esquema de la BD.
    - Cambios en los índices.
    - Reprovisión. Es decir, modificar el número de nodos en los que resida la BD.
    - Actualización del software gestor.
  - Todo ello debe resolverse hoy en día sin comprometer la disponibilidad.

4. Distribución de datos

- 6. Evitar operaciones que impliquen a más de un nodo.
  - Ese tipo de operaciones requerirían sincronización entre los nodos que intervengan.
    - Con ello se perderían los beneficios del consejo 1.
  - La clave para solucionarlo reside en una estrategia acertada para el reparto de la información.
  - La información que rara vez se modifique pero que se lea frecuentemente puede estar replicada en todos los nodos.

4. Distribución de datos

- 7. No trates de proporcionar ACID por ti mismo.
  - Las propiedades ACID se garantizaron en las BBDD relacionales.
  - En los almacenes clave-valor, esas garantías no se soportan.
  - Si se necesitan, la mejor opción será buscar el SGBD relacional más escalable.
  - No tiene sentido “reconstruir” todo ese soporte sobre un almacén clave-valor.

4. Distribución de datos

- 8. Administración sencilla.
  - Si una BD es muy escalable pero su administración es compleja, poco interesará su uso.
  - Tanto el conjunto de herramientas administrativas como el propio sistema gestor deben ofrecer una interfaz de usuario sencilla que permita realizar o automatizar cómodamente las tareas de administración.
  - Debería bastar con configurarlas, para que se realicen en el momento oportuno de manera automática.

4. Distribución de datos

- 9. Prestar atención al rendimiento por nodo.
  - Aunque varios sistemas sean escalables, puede que interese más elegir aquél capaz de obtener un mejor rendimiento por nodo.
  - Con ello habrá que desplegar los datos en un menor número de máquinas y se reducirá el coste de adquisición o de alquiler (en un entorno cloud).

4. Distribución de datos

- 10. Mejor control con BD “open-source”.
  - Así se intenta evitar:
    - Coste de las actualizaciones.
    - Mantenimiento de las licencias.
    - Dependencia de un determinado proveedor.
    - Imposibilidad de migración a otros entornos, plataformas o infraestructuras.

4.1. Ejemplos de “sharding”

- En el Tema 3 ya vimos que el sharding…:
  - Mejora la escalabilidad.
  - Aumenta la concurrencia sin introducir bloqueos.
  - Puede ser fácilmente combinado con replicación.
- Pero falta ver ejemplos de sistemas que lo utilicen.

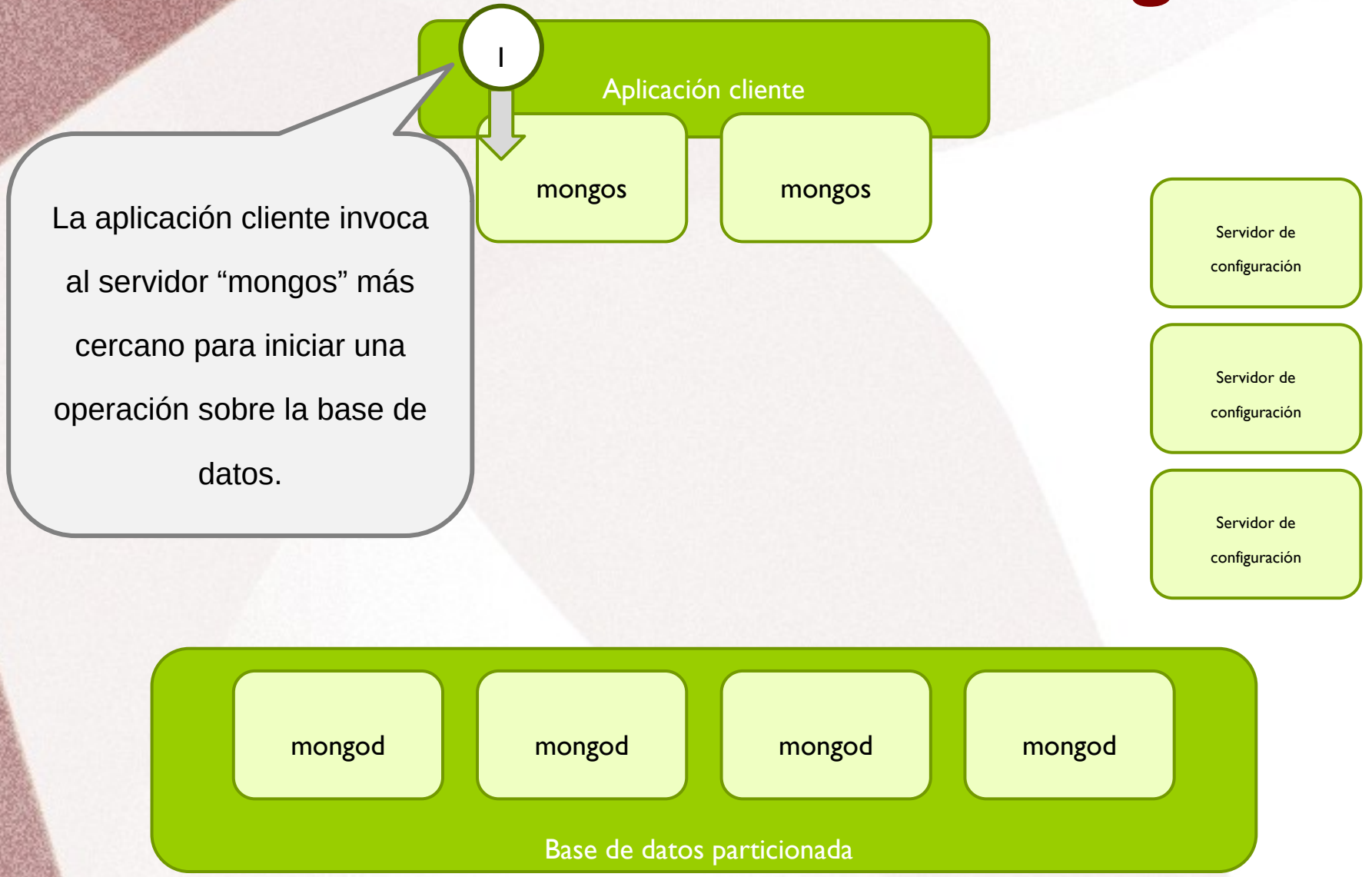
4.1. Ejemplos de "sharding"

- Ejemplo 1: MongoDB.
  - MongoDB es una base de datos NoSQL que sigue el modelo de "almacén de documentos".
  - Documentación disponible en: https://docs.mongodb.com/manual/sharding/
  - Utiliza "*sharding*" automatizado para obtener escalabilidad horizontal.
- Ejemplo 2: Apache HBase.
  - Base de datos NoSQL (almacén de registros extensibles) utilizada en Apache Hadoop.
  - Con "sharding" automatizado.
  - Documentación: http://hbase.apache.org/book.html#regions.arch
- Ejemplo: Windows Azure SQL Database.
  - Base de datos relacional escalable dentro de la plataforma Azure.
  - También utiliza reparto de datos, aunque lo llama "federación".
  - Véase: "SQL Azure - Scaling Out with SQL Azure Federation"

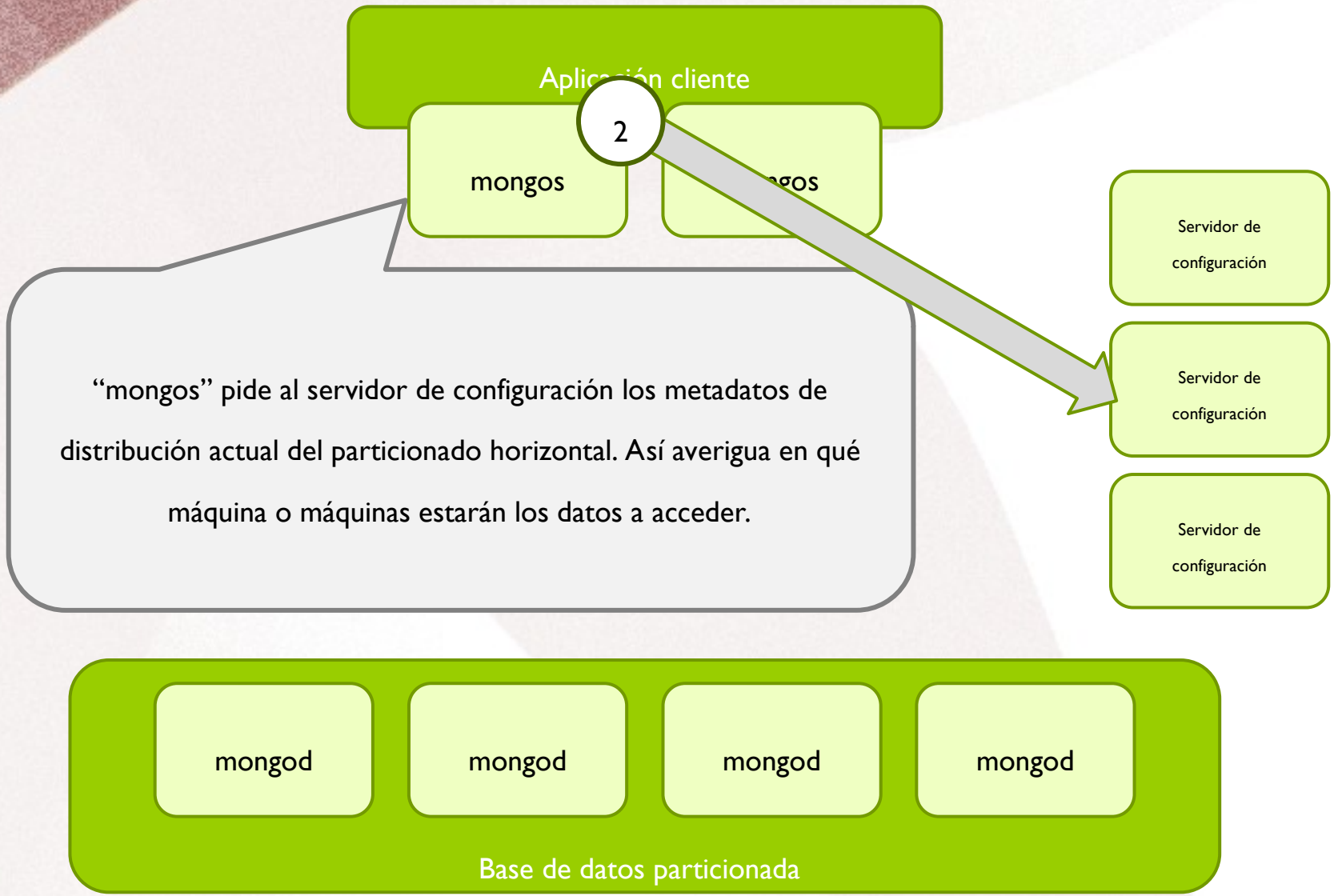
4.1.1. MongoDB

- Cuando se acceda a una BD grande en MongoDB, habrá que utilizar tres tipos de “servidores” (cada servidor en un nodo):
  1. Procesos “mongod”
     - Cada uno mantiene un subconjunto de filas de la base de datos.
     - Llamaremos “partición” (“*shard*”) a cada subconjunto.
     - Para mejorar la escalabilidad, se utiliza particionado horizontal.
     - Cada “partición” puede replicarse.
       - Modelo de replicación pasivo. Consistencia secuencial. Partición primaria.
  2. Procesos “mongos”
     - En caso de utilizar particionado, son los procesos que actúan como interfaz con la aplicación cliente.
     - Encaminan las peticiones hacia el proceso “mongod” apropiado.
     - Consultan a los servidores de configuración para saber en qué “mongod” se encuentran las filas a utilizar.
     - Redirigen posteriormente la petición a esos procesos.
  3. Servidores de configuración
     - Guardan los metadatos de la BD.
     - Saben qué filas forman cada partición y en qué nodo se ubica cada partición.

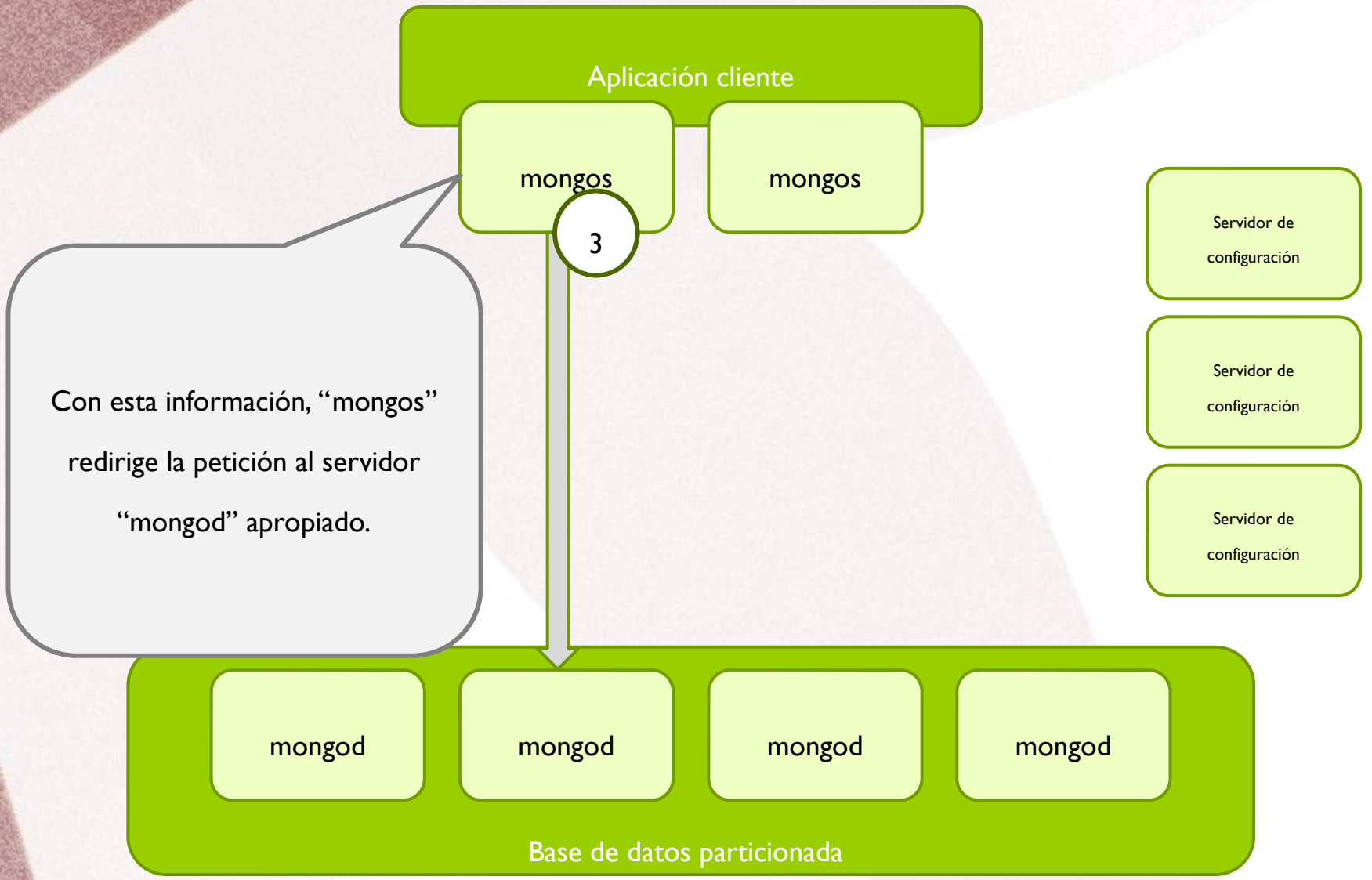
4.1.1. MongoDB


1
Aplicación cliente
mongos
mongos
La aplicación cliente invoca al servidor “mongos” más cercano para iniciar una operación sobre la base de datos.
Servidor de configuración
Servidor de configuración
Servidor de configuración
mongod
mongod
mongod
mongod
Base de datos particionada

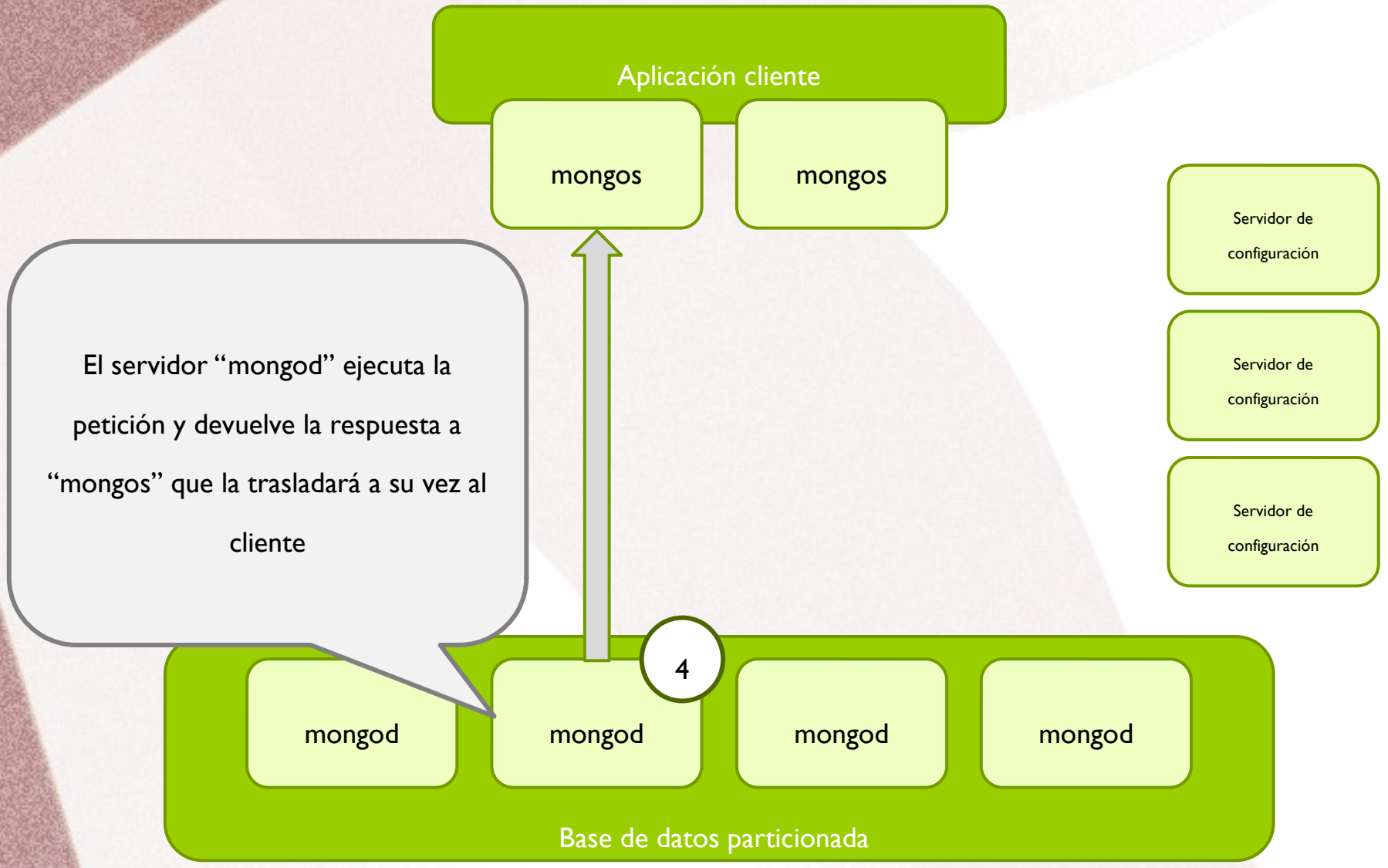
4.1.1. MongoDB


Aplicación cliente
2
mongos
Servidor de configuración
Servidor de configuración
Servidor de configuración
"mongos" pide al servidor de configuración los metadatos de distribución actual del particionado horizontal. Así averigua en qué máquina o máquinas estarán los datos a acceder.
mongod
mongod
mongod
mongod
Base de datos particionada

4.1.1. MongoDB


Aplicación cliente
mongos
mongos
3
Con esta información, "mongos" redirige la petición al servidor "mongod" apropiado.
Servidor de configuración
Servidor de configuración
Servidor de configuración
mongod
mongod
mongod
mongod
Base de datos particionada

4.1.1. MongoDB


Aplicación cliente
mongos
mongos
Servidor de configuración
Servidor de configuración
Servidor de configuración
El servidor "mongod" ejecuta la petición y devuelve la respuesta a "mongos" que la trasladará a su vez al cliente
4
mongod
mongod
mongod
mongod
Base de datos particionada

4.1.1. MongoDB

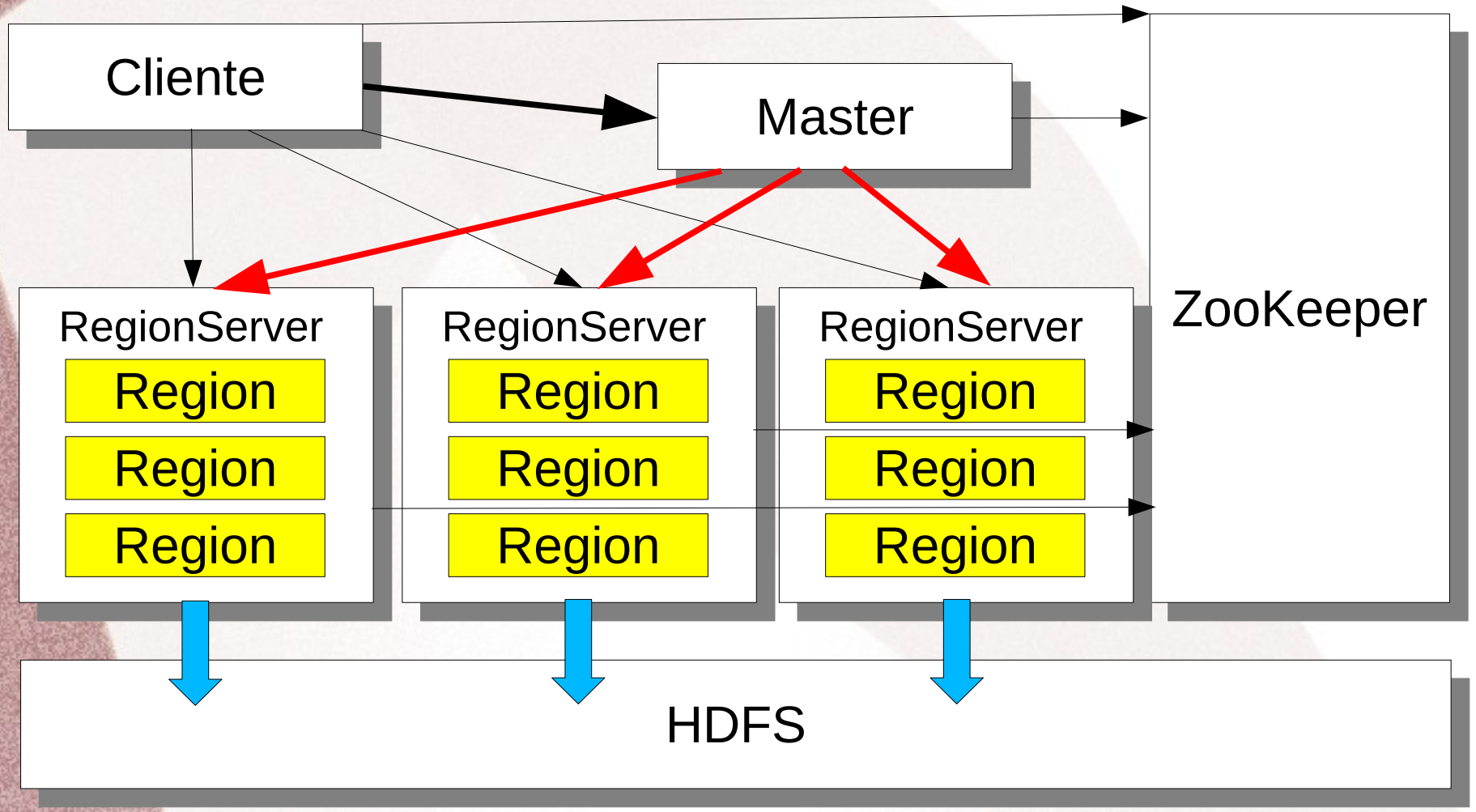
- "*Sharding*" automatizado:
  - El subconjunto ("*shard*") de cada servidor se divide en fragmentos ("*chunks*").
  - Cada fragmento ("*chunk*") no podrá superar los 64 MB.
    - Cuando eso sucede, el fragmento se divide en dos.
  - Un proceso equilibrador de carga monitoriza cuántos fragmentos tiene cada servidor "mongod".
    - Si el número es desigual, se migra al menos un fragmento del servidor que tenga más al que tenga menos.
      - Hasta que el número de fragmentos se equilibre.
    - Para migrar un fragmento:
      - Se inicia la copia a su servidor destino.
      - Mientras dure la copia, todas los accesos se dirigen al servidor origen.
      - Cuando la copia finaliza:
        - Se guarda en los servidores de configuración que la migración terminó.
        - Se borra el fragmento del servidor origen.

4.1.2. HBase

- HBase es un “almacén de registros extensibles”:
  - Utilizado en Apache Hadoop.
  - Esquema basado en tablas con columnas agrupadas en familias.
    - Sigue el modelo de Google Bigtable.
  - Implantado sobre HDFS.
    - HDFS ya proporciona replicación de bloques.
      - 3 réplicas por omisión.
      - Ubicadas en nodos diferentes.
  - Diseñado para grandes volúmenes de datos.
    - En el orden de los terabytes.

4.1.2. HBase

- Arquitectura:



4.1.2. HBase

- Componentes de la arquitectura:
  - Master:
    - Gestor principal de HBase.
    - Decide cómo repartir cada tabla en regiones.
      - Esa información se guarda en ZooKeeper.
        - Tabla **`hbase:meta`**.
    - Monitoriza el estado y la carga de los RegionServers.
      - Se redistribuye el reparto (“*sharding*”) en caso de necesidad.

4.1.2. HBase

- Componentes (ii):
  - ZooKeeper:
    - Dos funciones principales:
      - Mantiene la configuración del sistema (en disco).
        - Esto incluye qué nodos han caído o si ha habido alguna partición en la red.
      - Proporciona sincronización.
    - Los clientes consultan qué RegionServer mantiene cada región.

4.1.2. HBase

- Componentes (iii):
  - RegionServer:
    - Mantiene un conjunto de regiones.
      - Cada región mantiene una subparte de una tabla.
        - Un rango contiguo de su clave.
      - Organización jerárquica:
        - Las regiones se dividen en Stores (cada uno guarda una ColumnFamily distinta).
        - Los Stores se dividen en MemStores (memoria principal).
        - Los MemStores se guardan en disco usando StoreFiles.
        - Los StoreFiles son secuencias de bloques (replicados).

4.1.2. HBase

- Componentes (iv):
  - RegionServer:
    - Como los StoreFiles y los Blocks son persistentes, cada región puede asignarse a otro RegionServer en caso de fallo.
      - Los StoreFiles están en HDFS.
      - El Master realiza la reasignación.
    - En caso de carga desequilibrada, el Master también redistribuye las regiones entre los RegionServers para equilibrarla.

4.1.2. HBase

- Con lo dicho en la hoja anterior el sharding automático es sencillo.
  - Si una región crece demasiado, su RegionServer la divide en dos y se lo comunica al Master.
  - Si dos regiones contiguas han perdido contenido, pueden fundirse en una sola.
    - El RegionServer lo reporta al Master.
  - El propio equilibrado de carga efectuado periódicamente por el Master completará el "*sharding*".

4.1.2. HBase

- Replicación:
  - Hbase no introduce replicación de RegionServers por omisión.
    - Pues HDFS ya usa replicación de bloques.
  - Si se considera necesario, se puede configurar otra replicación por regiones.
    - Sigue el modelo pasivo.
    - Admite dos niveles de consistencia:
      - Strong: Estricta (solo primario).
      - Timeline: Secundarios accesibles en lecturas.
        - En principio, consistencia secuencial.
        - Pero puede relajarse mucho, pues HBase no garantiza que cada cliente acceda siempre a la misma réplica.