



Tema 6 – Escalabilidad



Tecnologías de los Sistemas de Información en la Red



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



Objetivos

- ▶ Conocer el compromiso entre consistencia, disponibilidad y tolerancia a las particiones de red cuando se utilicen servicios distribuidos.
- ▶ Identificar los mecanismos básicos que permiten mejorar la escalabilidad de un sistema distribuido.
- ▶ Gestionar adecuadamente un sistema escalable para que sea elástico.
- ▶ Identificar las dificultades a vencer para implantar un sistema escalable: puntos de contención.



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



I. Introducción

- ▶ El curso pasado, en CSD, se describieron varios mecanismos de escalado:
 - ▶ Distribución de procesos
 - ▶ Distribución de datos
 - ▶ Replicación
 - ▶ Uso de cachés
- ▶ Todos guardan relación con la consistencia de datos:
 - ▶ Cuanto más fuerte sea la consistencia mayor sincronización habrá entre los procesos.
 - ▶ Eso conduce a bloqueos. Por tanto, no es escalable.
 - ▶ La consistencia relajada no necesita intercambiar muchos mensajes
 - ▶ Puede ser escalable
 - ▶ Pero, en algunos casos, puede generar resultados inesperados.



I. Introducción

- ▶ Hay algunos aspectos de la consistencia que deben considerarse en los servicios escalables:
 - ▶ Teorema CAP
 - ▶ No se podrá utilizar una consistencia fuerte en un servicio altamente escalable.
 - ▶ Los modelos clásicos de replicación proporcionan consistencia fuerte (p.ej., secuencial)
 - ▶ Se necesitan otras aproximaciones para relajar la consistencia.
 - ▶ ¿Cómo podemos repartir los datos entre múltiples servidores?
 - ▶ Técnicas de particionado de datos ---> Almacenes NoSQL.



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



2. Teorema CAP

- ▶ En un sistema distribuido no se puede mantener a la vez:
 - ▶ Una consistencia fuerte (C); p.ej., secuencial. *requiere consenso.*
 - ▶ La disponibilidad de los servicios (“Availability”:A).
 - ▶ Disponibilidad: Todo cliente obtiene respuesta siempre.
 - ▶ Cada réplica servidora puede tener sus clientes.
 - Todos los clientes deben ser capaces de obtener respuesta para considerar que el servicio está disponible.
 - ▶ La tolerancia al particionado de la red (P).
 - ▶ Que el fallo que se produzca sea en las comunicaciones y deje varios subgrupos sin posibilidad de intercomunicarse.
 - Pérdida de conectividad.
- ▶ Hay que sacrificar una de las tres propiedades.

2. Teorema CAP

- ▶ ¿Por qué se plantea este teorema?
 - ▶ En un sistema distribuido debe proporcionarse transparencia de fallos.
 - ▶ Para ello, hay que replicar todos los servicios proporcionados.
 - ▶ Así se asegura la DISPONIBILIDAD (A).
 - ▶ Debe proporcionarse la imagen de un sistema único.
 - ▶ Esto implica que los usuarios deben observar el mismo estado en todas las réplicas: CONSISTENCIA (C) secuencial.
 - ▶ La transparencia de fallos también obliga a que el sistema siga comportándose de igual manera aunque haya problemas de conectividad.
 - ▶ Esto fuerza a que se garantice la tolerancia al PARTICIONADO (P) de la red.



2. Teorema CAP

- ▶ Opción I: Sacrificar la tolerancia al particionado de la red (P).
- ▶ Cuando se quiera garantizar una consistencia fuerte y una disponibilidad total de los servicios, hay que renunciar a la tolerancia del particionado.
 - ▶ No se admitirá que la red se particione.
 - ▶ Habrá que garantizar (replicando la red, por ejemplo) la conectividad.
 - ▶ Pero eso es difícil de asegurar:
 - Solo se podría soportar en un despliegue local en un solo laboratorio
 - Incluso entonces sería extremadamente difícil de garantizar
 - ¡Hay que buscar otras alternativas!

2. Teorema CAP

- ▶ Opción 2: Sacrificar la disponibilidad (A).
 - ▶ En este caso se pretende asegurar la consistencia fuerte y la tolerancia a las particiones.
 - ▶ Cuando haya una partición:
 - ▶ Se adoptará el modelo de partición primaria.
 - Todos los subgrupos minoritarios pararán.
 - Se perderá la disponibilidad en sus nodos.
 - Los clientes asociados a esos nodos podrán advertir esa situación.
 - Solo el subgrupo “primario” continuará
 - Sus nodos mantendrán una consistencia fuerte.
 - Es el subgrupo con una mayoría de nodos.



2. Teorema CAP

- ▶ Opción 3: Sacrificar la consistencia (C).
 - ▶ En este caso se pretende garantizar la disponibilidad de todas las réplicas correctas y la tolerancia al particionado de la red.
 - ▶ Cuando haya una partición:
 - ▶ Se admitirá que todos los subgrupos avancen.
 - Modelo particionable.
 - ▶ Al procesar escrituras en cada subgrupo, se perderá la consistencia fuerte.
 - Los demás subgrupos no pueden ver esas escrituras.
 - ▶ Si se diseña el sistema con cuidado y las operaciones son conmutativas, se podrá soportar una consistencia eventual.
 - Reconciliación de estado sencilla.

- ▶ ¿Cuál es la mejor opción?
 - ▶ Los servicios escalables deben estar siempre disponibles
 - ▶ Su objetivo es atender a un gran (y potencialmente creciente) número de usuarios
 - ▶ Una vez desplegados en múltiples centros de datos, podrán darse problemas de conectividad
 - Esas situaciones de particionado deben admitirse
 - ▶ Por tanto, debe sacrificarse la consistencia fuerte
 - De hecho, ese sacrificio no es una novedad pues ya es un requisito para ser altamente escalable



Índice

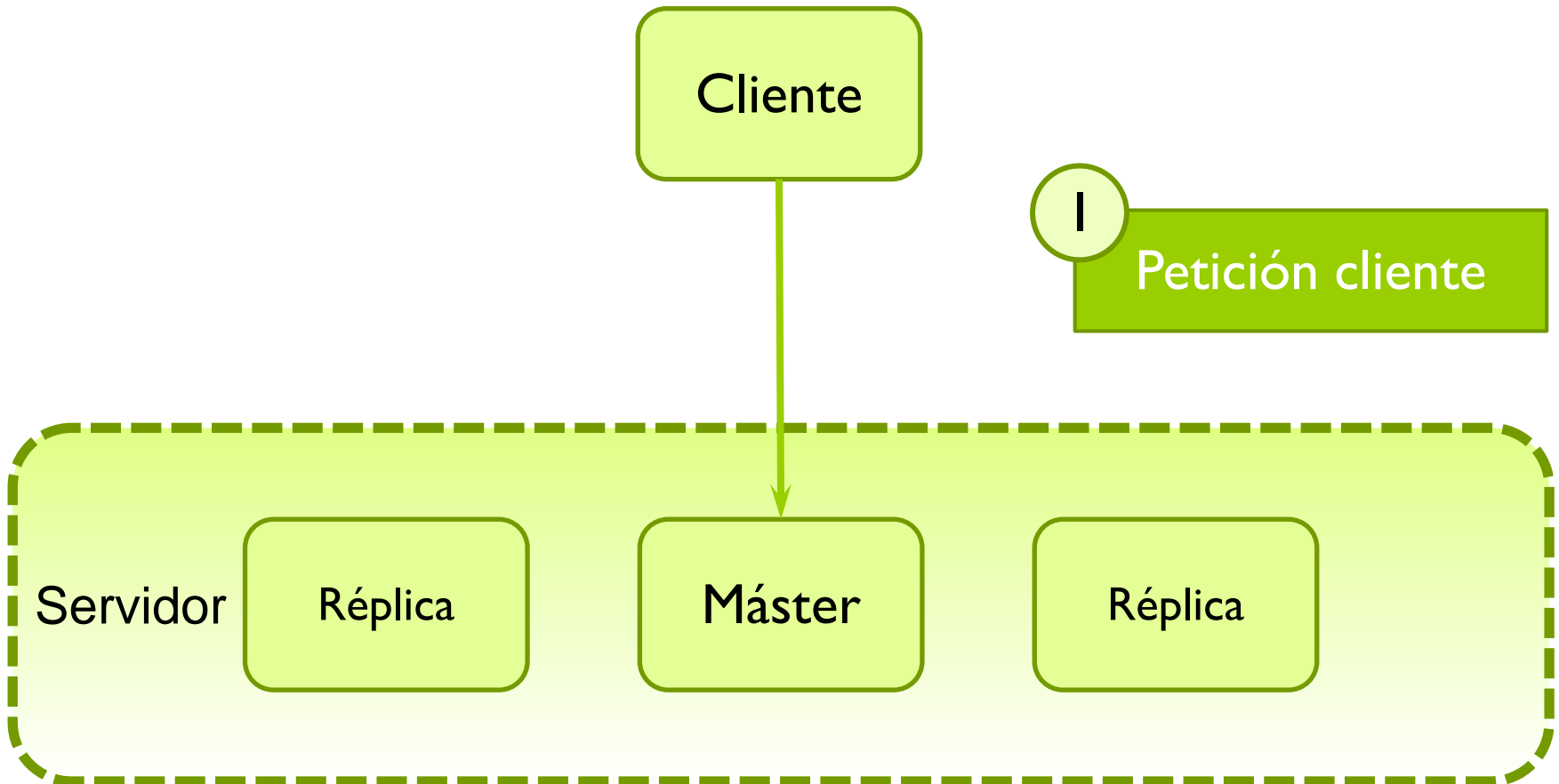
1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



3. Replicación multi-máster

- ▶ Hay dos modelos de replicación clásicos:
 - ▶ Pasivo (o replicación “primary/backup”)
 - ▶ Activo (o replicación de máquina de estados)
- ▶ Desafortunadamente, ambos son fuertemente consistentes
 - ▶ Esto limita su escalabilidad
- ▶ Hay una tercera solución:
 - ▶ Replicación multi-máster
 - ▶ Basada en el modelo pasivo, pero...
 - No hay una sola réplica primaria
 - Cada solicitud puede ser atendida por una sola réplica “master”
 - **¡Pero cada solicitud puede usar un máster distinto!**
 - La respuesta se envía de inmediato al cliente
 - Las modificaciones se envían de manera perezosa a las demás réplicas posteriormente

3. Replicación multi-máster

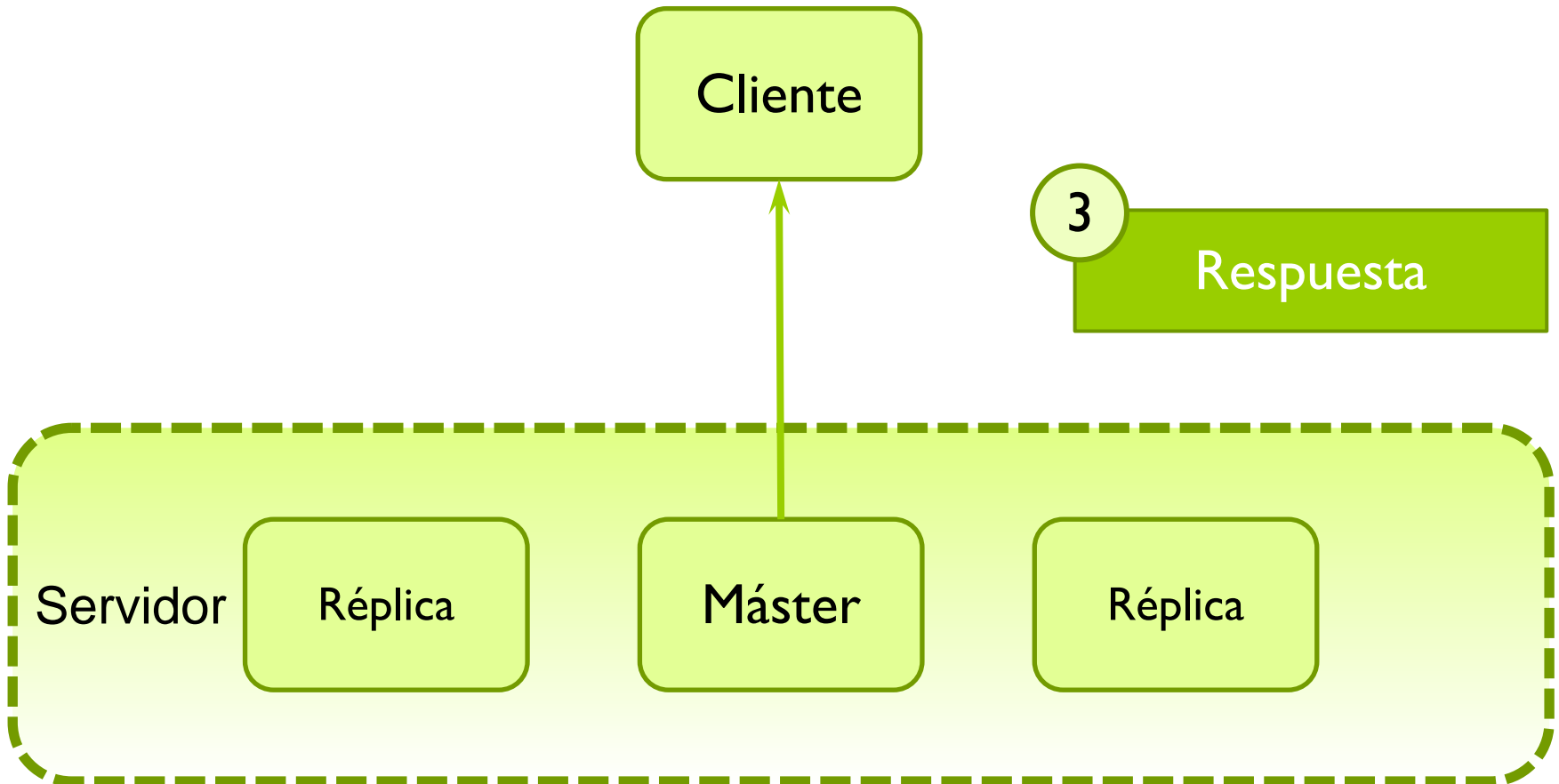


3. Replicación multi-máster

Cliente



3. Replicación multi-máster



3. Replicación multi-máster

Cliente



3. Replicación multi-máster

▶ Ventajas

▶ Mínima sobrecarga

- ▶ Cada petición es procesada por una sola réplica
 - Tanto para peticiones de solo lectura como de modificación

▶ Altamente escalable

- ▶ No asume ningún mecanismo de control de concurrencia

▶ Admite operaciones no deterministas

- ▶ Solo son ejecutadas por un máster
- ▶ Nunca surgirán inconsistencias entre réplicas causadas por el servicio de una misma operación



3. Replicación multi-máster

▶ Inconvenientes

▶ Problemas en caso de fallos

▶ Si un máster cae...

- Sus peticiones en curso pueden perderse

▶ Hereda la débil gestión de fallos del modelo pasivo

▶ Los fallos arbitrarios no pueden gestionarse

▶ Pueden surgir inconsistencias fácilmente

▶ Entre operaciones concurrentes sobre un dato compartido atendidas por diferentes maestros

- Esto puede resolverse si el programa fue escrito asumiendo consistencia eventual
- ¡Pero es responsabilidad del programador!



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



4. Almacenes NoSQL

- ▶ ¿Cómo mejorar la escalabilidad de una base de datos?
 - ▶ **Simplificando el esquema**
 - ▶ Sustituir múltiples tablas con múltiples columnas por tablas clave/valor.
 - Índices simples
 - ▶ Esto simplifica el lenguaje de interrogación
 - Procesamiento directo
 - ▶ El espacio ocupado por la base también se reduce
 - Las bases de datos pueden mantenerse en memoria principal
 - Persistencia asegurada mediante replicación
 - ▶ **Eliminando las transacciones**
 - ▶ No se podrá agrupar una secuencia de sentencias en una misma transacción.
 - ▶ Atomicidad limitada a cada sentencia por separado.
 - Bloqueos breves: imperceptibles.
- ▶ **Resultado: Almacenes NoSQL.**



4. Almacenes NoSQL

- ▶ Variantes de almacenes NoSQL:
 1. Almacenes clave-valor.
 2. Almacenes de documentos.
 3. Almacenes de registros extensibles.



4.1. Almacenes clave-valor

- ▶ Esquema compuesto por dos campos: clave y valor.
- ▶ No hay apenas soporte para distinguir atributos en el “valor”.
- ▶ No hay forma de “interrogar” por atributos no primarios.
- ▶ Ejemplos: Dynamo, Voldemort, Riak...



4.2. Almacenes de documentos

- ▶ Esquema compuesto por objetos con un número variable de atributos.
- ▶ En algunos casos estos atributos pueden ser, a su vez, objetos.
- ▶ Lenguaje de interrogación basado en establecer restricciones sobre los valores de los atributos.
- ▶ Ejemplos: CouchDB, MongoDB, SimpleDB...



4.3. Almacenes de registros extensibles

- ▶ Esquemas formados por tablas que pueden tener un número variable de columnas.
 - ▶ En algunos casos se organizan en grupos de columnas.
- ▶ Las tablas pueden particionarse tanto vertical como horizontalmente.
 - ▶ Técnica conocida como “*sharding*”.
 - ▶ Reparte responsabilidad entre múltiples nodos.
 - ▶ Equilibrado de la carga.
 - ▶ Mayor concurrencia.
 - ▶ ¡Fácilmente escalable sobre múltiples servidores!
- ▶ Ejemplos: Bigtable, PNUTs, Cassandra...



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



5. Elasticidad

- ▶ Un sistema se considera elástico cuando:
 - ▶ Es escalable.
 - ▶ Es adaptable.
 - ▶ Asigna a cada aplicación la cantidad justa de recursos que necesite.
 - ▶ Esta adaptación es:
 - Dinámica: Depende de la carga que haya en cada momento.
 - Autónoma: El sistema administra sus recursos sin intervención humana.
- ▶ Definición:
 - ▶ “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”.

5. Elasticidad

- ▶ La elasticidad es importante en los sistemas “cloud” modernos.
 - ▶ A la hora de ajustar el coste para el usuario.
 - ▶ A la hora de administrar los recursos del proveedor.
- ▶ Requiere:
 - ▶ Un sistema de monitorización:
 - ▶ De la carga soportada.
 - ▶ Del rendimiento obtenido.
 - ▶ Un sistema de actuación:
 - ▶ Para automatizar la reconfiguración de los servicios.
 - Para incorporar más nodos cuando la carga aumente.
 - Para liberar recursos cuando la carga disminuya.
 - ▶ En función del “*Service Level Agreement*” (SLA) establecido.
 - La reacción del sistema debe ser rápida, para no incumplir las condiciones del SLA.



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía

6. Contención y cuellos de botella

- ▶ En cualquier servicio distribuido habrá múltiples componentes.
- ▶ Su diseño debe ser cuidadoso para evitar la contención (bloqueo, saturación) en alguno de ellos.
 - ▶ Es decir, que alguno de los componentes se sature o bloquee a los demás, mientras los demás admitan cargas mayores.
- ▶ Causas de la contención:
 - ▶ Uso de algoritmos centralizados en algunas tareas “pesadas”.
 - ▶ La centralización solo tiene sentido para reducir tráfico en casos de tareas “ligeras” o infrecuentes.
 - ▶ Uso de herramientas de sincronización.
 - ▶ Si hay recursos compartidos entre múltiples actividades, habrá una sección crítica y su protección conducirá a bloqueos.
 - ▶ Tráfico excesivo.
 - ▶ Una mala distribución de los recursos puede conducir a un incremento de las necesidades de comunicación remota.

6. Contención y cuellos de botella

- ▶ ¿Cómo evitar la contención?
 - ▶ Ante “centralización”:
 - ▶ Adoptando una solución descentralizada, en caso de que la tarea a desarrollar sea “pesada”.
 - ▶ En el acceso a recursos compartidos:
 - ▶ Serializando los accesos y adoptando un paradigma de programación asincrónica.
 - Si no hay competición por el uso de los recursos, resulta más fácil gestionar la carga.
 - El sistema resultante es más eficiente.
 - Menos cambios de contexto.
 - Menor carga para el sistema operativo y el middleware.
 - ▶ Por tráfico excesivo:
 - ▶ Replicar los recursos y mantenerlos consistentes.
 - ▶ Los accesos remotos se transforman en accesos locales.



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
 1. El módulo “cluster” en NodeJS
 2. Escalabilidad utilizando múltiples ordenadores
 3. MongoDB
8. Resultados de aprendizaje
9. Bibliografía



7.1. El módulo “cluster” de NodeJS

- ▶ Los procesos en Node.js solo tienen un hilo de ejecución.
- ▶ Si, por razones de escalabilidad, se necesita que un componente del sistema distribuido disponga de más instancias, no se podrán crear más hilos:
 - ▶ Habrá que crear más procesos, instancias del mismo componente
 - ▶ Si existen varios procesos, réplicas de un mismo componente, se podrá repartir el incremento de la carga de servicio entre ellos
- ▶ El módulo **cluster** de Node.js facilita:
 - ▶ La creación y gestión de un pool de trabajadores (procesos de Node)
 - ▶ El reparto equilibrado de la carga de servicio entre los procesos

7.1. El módulo “cluster” de NodeJS

- ▶ El módulo **cluster**:
 - ▶ Permite aprovechar los sistemas multiprocesador, mediante la ejecución de un cluster de procesos Node.
 - ▶ Para ello, reparte la carga de servicio entre los procesos que se ejecutarán en los distintos procesadores.
 - ▶ Permite crear procesos que comparten todos los puertos asociados al servicio que presten.
 - ▶ Incluye una clase **Worker** que modela los procesos (trabajadores) del cluster.
 - ▶ Son supervisados y creados por un proceso “master”.
 - ▶ Dispone de eventos (‘fork’, ‘online’, ‘listening’, ‘disconnect’, ‘exit’...) que le permiten supervisar el estado de los procesos trabajadores (workers).
 - ▶ Usa IPC en la comunicación de los workers con el cluster master.

7.1. El módulo “cluster” de NodeJS

- Esquema básico de un cluster o pool de trabajadores:

```
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;
var server = ... // the server which runs the workers
var port = ... // the port where the server binds

if (cluster.isMaster) { // code of the master one
    // fork: create workers
    for (var i=0; i < numCPUs; i++) cluster.fork();
    // listening death of workers
    cluster.on('exit', function(worker, code, signal) {
        console.log('worker', worker.process.pid, 'died');
    });
} else { // code of any worker
    server.listen(port); // each worker runs the server
}
```

7.1. El módulo “cluster” de NodeJS

- Ejemplo: Código de un cluster de servidores http:

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) { // code of the master one
  for (var i=0; i < numCPUs; i++) cluster.fork();
  cluster.on('exit', function(worker, code, signal) {
    console.log('worker', worker.process.pid, 'died');
  });
} else { // code of any worker
  http.createServer(function(req, res) {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);
}
```

7.1. El módulo “cluster” de NodeJS

- ▶ Eventos del objeto **Cluster**:
 - ▶ ‘**fork**’: cuando se crea un nuevo worker
 - ▶ ‘**online**’: cuando se recibe un mensaje de un worker indicando que ha comenzado a ejecutarse
 - ▶ ‘**listening**’: cuando un worker ejecuta un `listen()`
- ▶ Estos eventos pueden usarse para supervisar la actividad de los workers y asociarles timeouts. Por ejemplo:

```
var timeouts = [];  
function errorMsg() { console.error('Something wrong...');}  
cluster.on('fork', function(worker) {  
    timeouts[worker.id] = setTimeout(errorMsg, 2000);  
});  
cluster.on('listening', function(worker, address) {  
    clearTimeout(timeouts[worker.id]);  
});
```




7.1. El módulo “cluster” de NodeJS

- ▶ Eventos del objeto **Cluster**:
 - ▶ ‘**disconnect**’: cuando el canal IPC de un worker se desconecta (porque el worker termine, ‘**exit**’, o sea eliminado, ‘**kill**’, o se desconecte, ‘**disconnect**’)
 - ▶ ‘**exit**’: cuando un worker muere
- ▶ Estos eventos pueden usarse para detectar la inactividad de los workers y reiniciarlos, usando **fork**. Por ejemplo:

```
cluster.on('exit', function(worker, code, signal) {  
    console.log('worker %d died (%s). restarting...',  
                worker.process.pid, signal || code);  
    cluster.fork();  
});
```



7.1. El módulo “cluster” de NodeJS

- ▶ El objeto `Cluster` dispone de un objeto `cluster.workers`:
 - ▶ Es un **hash** que almacena los workers activos, indexados por su id.
 - ▶ El evento ‘`fork`’ añade un worker a `cluster.workers`.
 - ▶ Los eventos ‘`disconnect`’ y ‘`exit`’ eliminan un worker del cluster.
- ▶ El cluster master puede enviar mensajes a un worker determinado o a todos los workers.

7.1. El módulo “cluster” de NodeJS

- ▶ Cada objeto **worker** (accesible mediante “*cluster.worker*”) dispone de:
 - ▶ **Atributos:**
 - ▶ **id** (identificador único del worker en *cluster.workers*)
 - ▶ **process** (su proceso)
 - ▶ **exitedAfterDisconnect** (boolean que permite distinguir entre exit voluntario (true) o muerte causada por otro proceso (false))
 - ▶ **Métodos**, que se invocan mediante su atributo *process*, por ejemplo: *process.send(...)*:
 - ▶ **send** (envío de mensajes al master)
 - ▶ **disconnect**
 - ▶ **kill**
 - ▶ **Eventos:**
 - ▶ ‘message’, ‘online’, ‘listening’, ‘disconnect’, ‘exit’, ‘error’

7.1. El módulo “cluster” de NodeJS

- ▶ Los mensajes entre master y workers (eventos ‘message’) pueden usarse, por ejemplo, para contar las peticiones de servicio:

```
var cluster = require('cluster');
var http = require('http');
if (cluster.isMaster) {
  var numReqs = 0;
  setInterval(function() { console.log("numReqs =", numReqs); }, 1000);
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd == 'notifyRequest') numReqs++;
  }
  var numCPUs = require('os').cpus().length;
  for (var i=0; i < numCPUs; i++) cluster.fork();
  for (var i in cluster.workers)
    cluster.workers[i].on('message', messageHandler);
} else {
  http.Server(function(req, res) {
    res.writeHead(200); res.end('hello world\n');
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}
```



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
 1. El módulo “cluster” en NodeJS
 2. Escalabilidad utilizando múltiples ordenadores
 3. MongoDB
8. Resultados de aprendizaje
9. Bibliografía



7.2. Escalabilidad sobre múltiples máquinas

- ▶ El módulo cluster es adecuado para aprovechar los recursos multiprocesador en una sola máquina.
- ▶ Sin embargo, las peticiones de servicio pueden sobrepasar la capacidad de una máquina multiprocesador. Se hará necesario escalar el servicio sobre varias máquinas.
- ▶ Algunas soluciones:
 - ▶ node-http-proxy
 - ▶ HAProxy
 - ▶ nginx [engine x]

7.2. Escalabilidad sobre múltiples máquinas

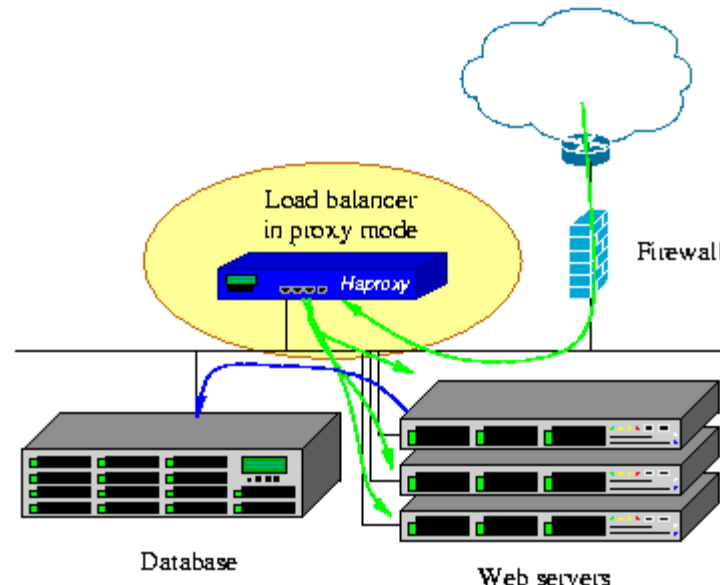
- ▶ **node-http-proxy** (<https://github.com/nodejitsu/node-http-proxy>)
 - ▶ Es un servidor proxy de código abierto para aplicaciones Node
 - ▶ Permite configurar servidores http en diferentes máquinas
 - ▶ Permite balancear la carga (peticiones de servicio) entre los servidores
- ▶ Esquema básico del servidor proxy:

```
var proxyServer = require('http-proxy');
var port = ...
var servers = [{ host: ... , port: ... }, ... ,
               { host: ... , port: ... } ];

proxyServer.createServer(function (req, res, proxy) {
  var target = servers.shift();
  proxy.proxyRequest(req, res, target);
  servers.push(target);
}).listen(port);
```

7.2. Escalabilidad sobre múltiples máquinas

- ▶ **HAProxy** (<http://www.haproxy.org/>)
 - ▶ Servidor proxy (para aplicaciones TCP y HTTP), de código abierto (escrito en C)
 - ▶ Proporciona equilibrado de carga, alta disponibilidad y fiabilidad, y escalado sobre múltiples máquinas
 - ▶ Usado en websites como GitHub, Stack Overflow, Tumblr, Twitter...



7.2. Escalabilidad sobre múltiples máquinas

- ▶ **HAProxy.** Ejemplo: Sea un servicio http de Node (server.js):

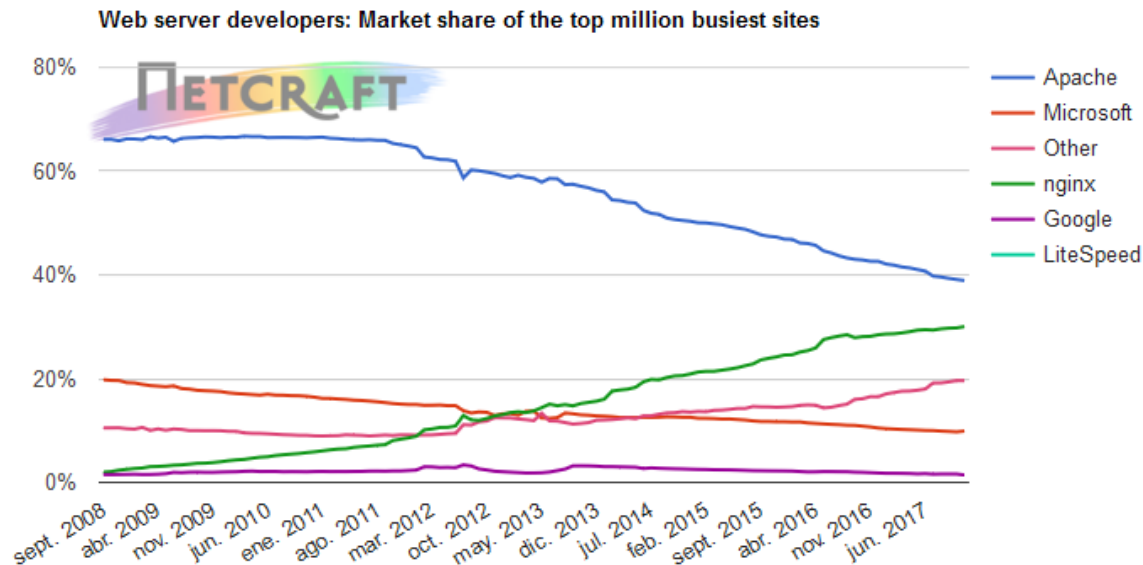
```
var http = require('http');  
function serve(ip, port) {  
  http.createServer(function (req, res) { ... }  
}  
serve('0.0.0.0', 9000);  
serve('0.0.0.0', 9001);  
serve('0.0.0.0', 9002);
```

- ▶ Un esquema básico de configuración de HAProxy para ese servicio:

```
frontend localnodes // where HAProxy listens to connections  
  bind *:80  
  mode http  
  default_backend nodes  
backend nodes // where HAProxy sends incoming connections  
  mode http  
  balance roundrobin  
  ... // other configuration options of the backend  
  server web01 127.0.0.1:9000 check  
  server web02 127.0.0.1:9001 check  
  server web03 127.0.0.1:9002 check
```

7.2. Escalabilidad sobre múltiples máquinas

- ▶ **nginx** (<http://nginx.org/en/>)
 - ▶ Servidor proxy inverso, de código abierto. Desarrollado por Igor Sysoev. Soportado comercialmente por Nginx, Inc.
 - ▶ Implantado inicialmente en websites de Rusia. Actualmente, se ha extendido por todo el mundo y lo usan más del 20% de los websites de mayor carga.





7.2. Escalabilidad sobre múltiples máquinas

- ▶ **nginx** (<http://nginx.org/en/>)
 - ▶ Es un servidor proxy inverso (para aplicaciones HTTP y de email)
 - ▶ Permite configurar servidores http en diferentes máquinas y balancear la carga (peticiones de servicio) entre los servidores
- ▶ Esquema básico de configuración (load balancing, round robin):

```
http {  
    upstream myapp1 {  
        server srv1.example.com;  
        server srv2.example.com;  
        server srv3.example.com;  
    }  
    server {  
        listen 80;  
        location / { proxy_pass http://myapp1; }  
    }  
}
```



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
 1. El módulo “cluster” en NodeJS
 2. Escalabilidad utilizando múltiples ordenadores
 3. MongoDB
8. Resultados de aprendizaje
9. Bibliografía



7.3. MongoDB

- ▶ MongoDB es “almacén de documentos” escalable.
- ▶ Por ser un almacén de documentos:
 - ▶ Una base de datos es un conjunto de colecciones.
 - ▶ Colección = Tabla.
 - ▶ Cada colección es un conjunto de objetos estructurados.
 - ▶ Cada objeto / documento tiene un identificador y múltiples atributos.
 - El identificador es equivalente a la clave primaria en el modelo relacional.
 - ▶ Los atributos, a su vez, pueden estructurarse: tipos simples, vectores, objetos (con atributos)...
 - Cada atributo puede verse como un par <clave, valor>.
 - Donde “clave” es el nombre del atributo.
 - ▶ No se exige que todos los documentos de una misma colección tengan idéntica estructura.
 - Esquema flexible.

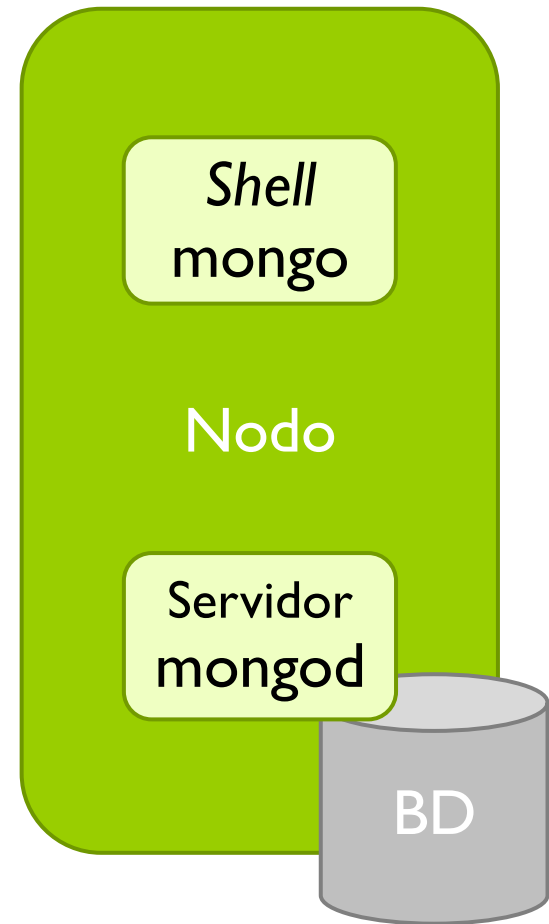


7.3.1. Componentes en MongoDB

- ▶ Para interactuar con un SGBD MongoDB se puede utilizar el shell “mongo”.
 - ▶ Proporciona una interfaz JavaScript para acceder a los servidores MongoDB.
 - ▶ Su orden “help” proporciona una primera descripción básica de todas las órdenes que admite.
 - ▶ A su vez:
 - ▶ “db.help()” describe todos los métodos que se pueden utilizar sobre una BD.
 - ▶ “db.colec.help()” describe todos los métodos que se pueden utilizar sobre una colección “colec” de la BD.
- ▶ En la configuración más sencilla, “mongo” interactuará con un servidor “mongod”.
 - ▶ El servidor “mongod” debe ser iniciado en primer lugar.

7.3.1. Componentes en MongoDB

- ▶ Configuración más sencilla:
 - ▶ Se puede utilizar una sola máquina.
 - ▶ Habrá un servidor “mongod” para gestionar la BD.
 - ▶ Se utilizará un *shell* “mongo” para interactuar con “mongod” y acceder así a la BD.
 - ▶ O bien un programa escrito en alguno de los lenguajes para los que haya algún “driver” de MongoDB.



7.3.1. Componentes en MongoDB

- ▶ Configuraciones más complejas en sistemas escalables.
- ▶ Se utiliza particionado horizontal.

En la configuración más sencilla un solo servidor mantiene todos los documentos de la BD.
Ejemplo: Alumnos en universidades españolas.
La “clave primaria” sería el DNI.

Servidor
mongod

BD “normal”

mongod

mongod

mongod

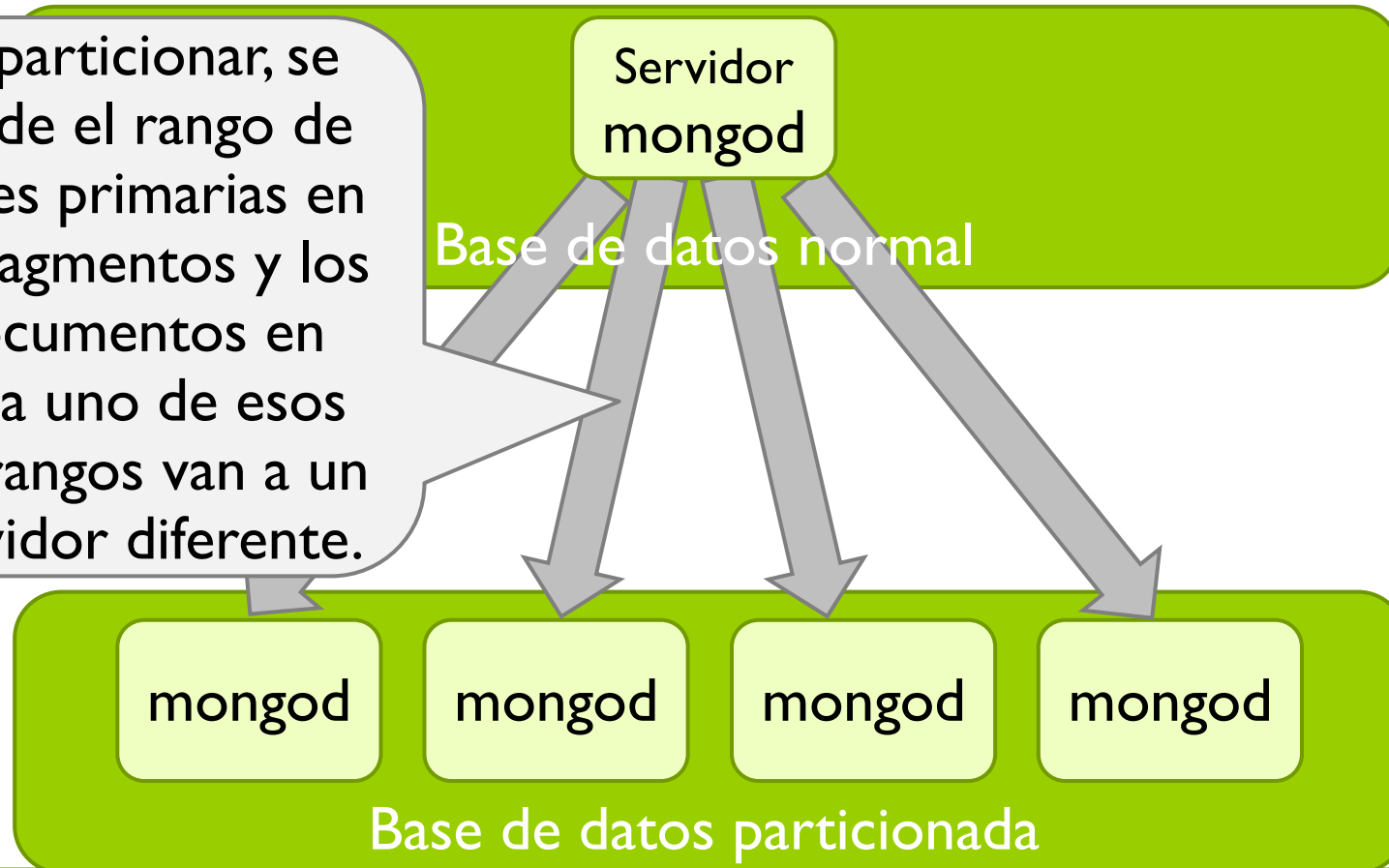
mongod

Base de datos particionada

7.3.1. Componentes en MongoDB

- ▶ Configuraciones más complejas en sistemas escalables.
- ▶ Se utiliza particionado horizontal.

Al particionar, se divide el rango de claves primarias en N fragmentos y los documentos en cada uno de esos subrangos van a un servidor diferente.



7.3.1. Componentes en MongoDB

- ▶ Configuraciones más complejas en sistemas escalables.
- ▶ Se utiliza particionado horizontal.

Servidor
mongod

Base de datos normal

En este ejemplo, los documentos podrían distribuirse inicialmente del siguiente modo: los estudiantes con DNI entre 0 y 24 MM en el servidor A, aquellos con DNI entre 25 MM y 49 MM en B, los DNI entre 50 y 74 MM en C y finalmente los DNI entre 75 y 99 MM en el servidor D.

mongod
A

mongod
B

mongod
C

mongod
D

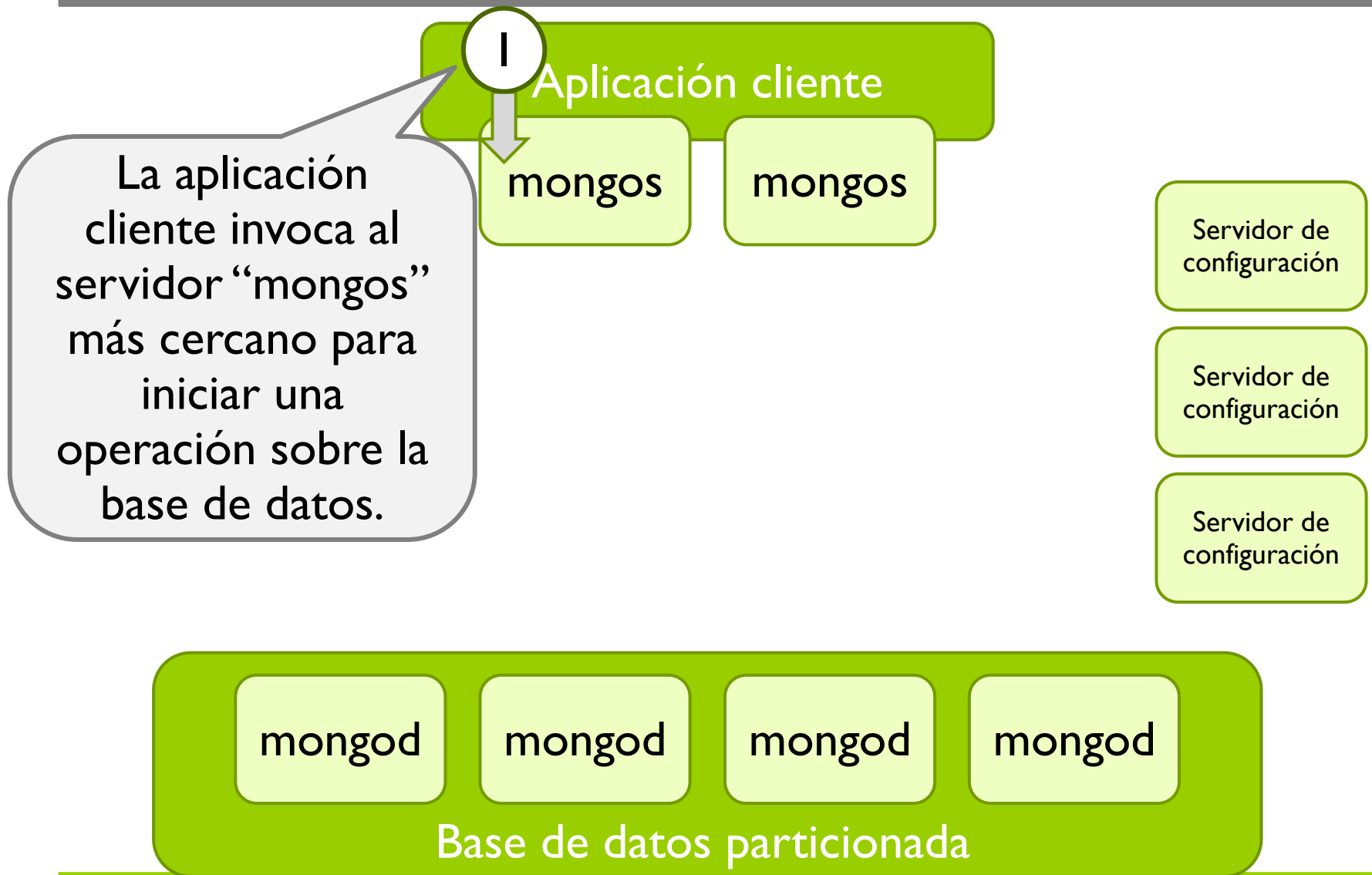
Base de datos particionada



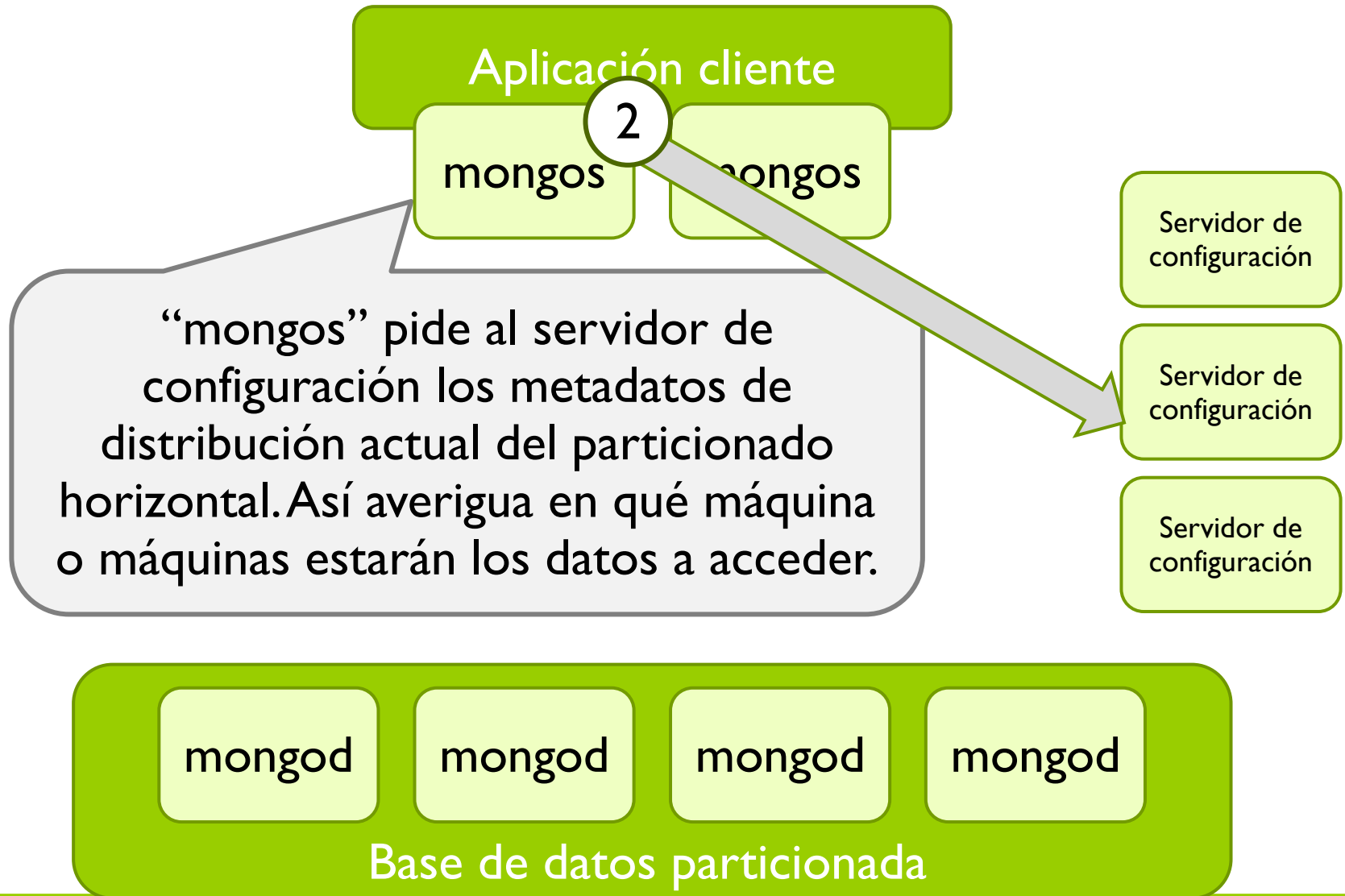
7.3.1. Componentes en MongoDB

- ▶ Cuando se acceda a una BD grande, habrá que utilizar tres tipos de “servidores” (cada servidor en un nodo) [2]:
 - ▶ **Procesos “mongod”**
 - ▶ Cada uno puede mantener un subconjunto de filas de la base de datos.
 - ▶ Para mejorar la escalabilidad, se utiliza particionado horizontal.
 - ▶ Cada “partición” puede replicarse.
 - ▶ **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: “routers”.
 - ▶ Consultan a los servidores de configuración para saber en qué “mongod” se encuentran las filas a utilizar.
 - Redirigen posteriormente la petición a esos procesos.
 - ▶ **Servidores de configuración**
 - ▶ Guardan los metadatos de la BD.
 - ▶ Saben qué filas forman cada partición y en qué nodo se ubica cada partición.
 - ▶ Forman un “conjunto de réplicas”; es decir, usan el protocolo de replicación normal de MongoDB.

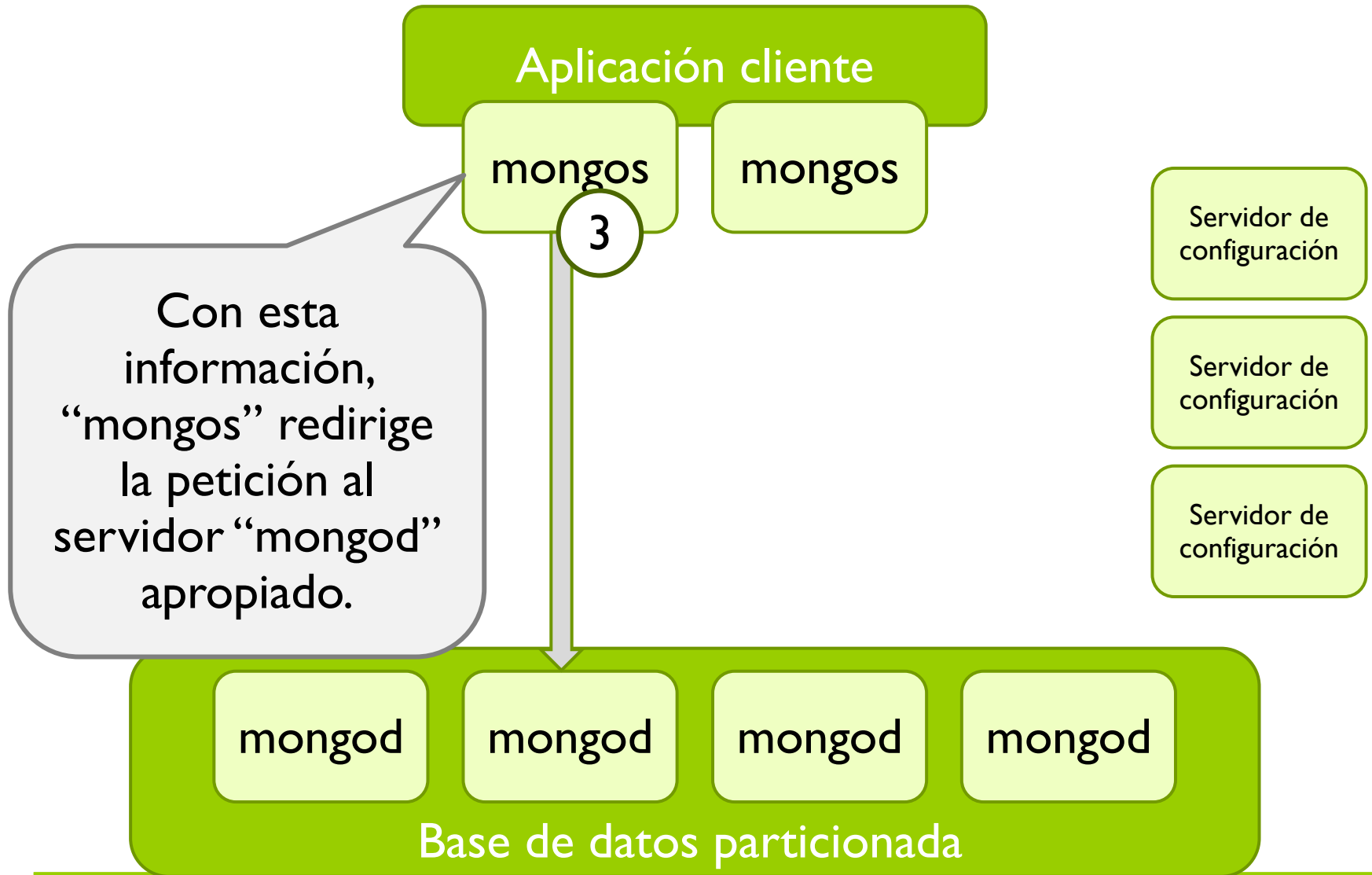
7.3.1. Componentes en MongoDB



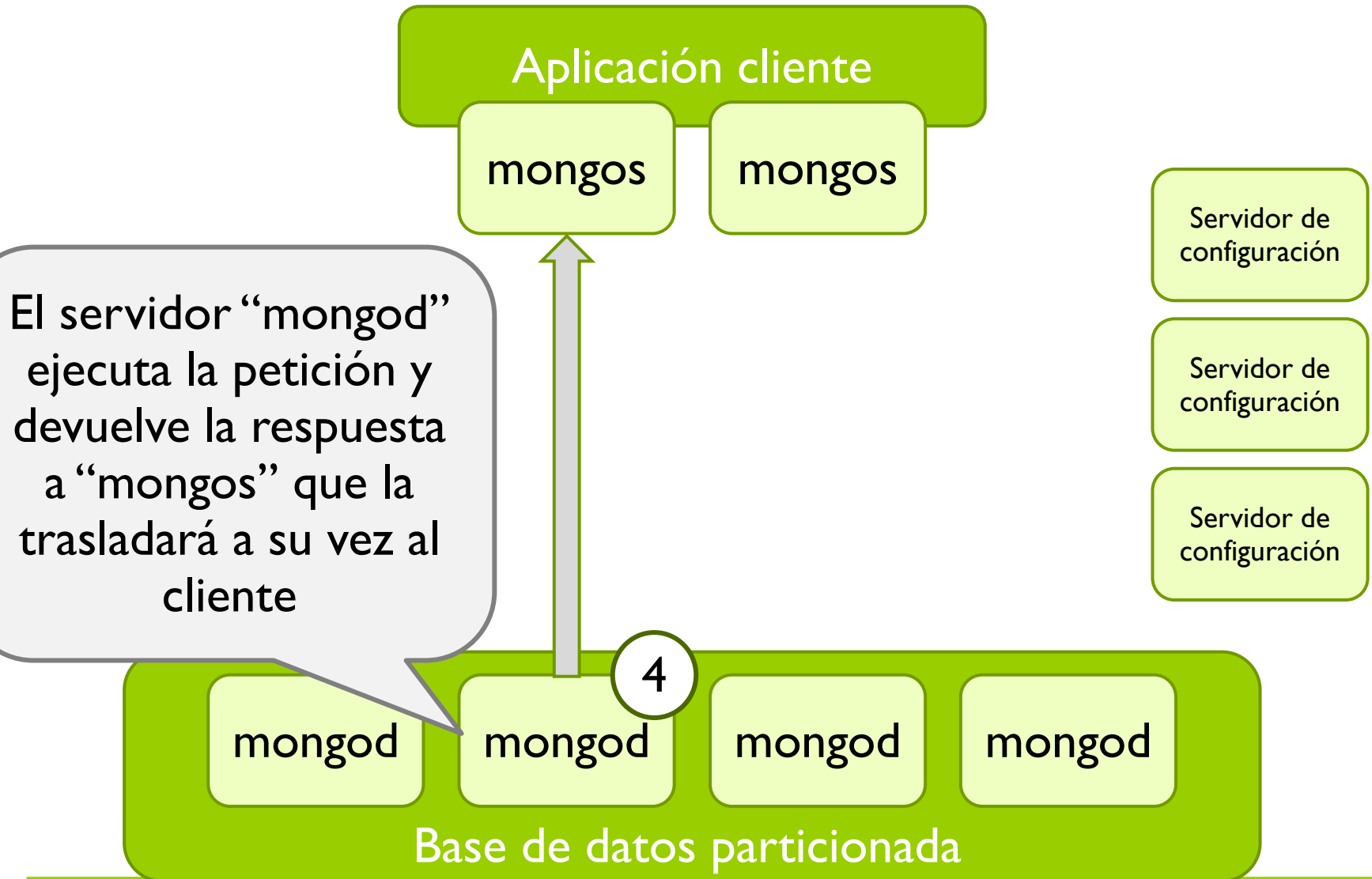
7.3.1. Componentes en MongoDB



7.3.1. Componentes en MongoDB



7.3.1. Componentes en MongoDB





7.3.2. Servidor “mongod”

- ▶ Es el servidor del SGBD.
- ▶ En caso de que no se use particionado ni replicación, basta con un “mongod” para acceder a la BD.
- ▶ Habrá que particionar la BD cuando:
 - ▶ Un solo “mongod” no soporte toda la carga que introducen los usuarios, o...
 - ▶ Los datos guardados en la BD llenen el disco de la máquina en que se ejecute “mongod”.

7.3.2. Servidor “mongod”

- ▶ Si se particiona la base:
 - ▶ 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, todos los accesos se dirigen al servidor origen.
 - Cuando la copia finaliza:
 1. Se guarda en los servidores de configuración que la migración terminó.
 2. Se borra el fragmento del servidor origen.

7.3.2. Servidor “mongod”

- ▶ Para evitar inconsistencias, se utiliza “journaling”:
 - ▶ Cada modificación se escribe primero en un “journal”.
 - ▶ Se anota el tipo de operación, los datos que modificará y su resultado.
 - ▶ Posteriormente se aplica sobre la BD.
 - ▶ Si ocurre algún fallo durante la modificación:
 - ▶ Si el fallo se da antes de terminar, el cliente no obtiene respuesta.
 - ▶ Al reanudar la ejecución, se examina el fichero “journal” y se comprueba si sus operaciones fueron aplicadas sobre la base.
 - Toda operación no finalizada se aplica ahora sobre la BD.
 - ▶ Ventajas:
 - ▶ Se evita la corrupción de los datos de la BD en caso de fallo.
 - No habrá escrituras incompletas.
 - ▶ Se acelera la reanudación del nodo.



7.3.3. Servidor “mongos”

- ▶ Debe ser el único tipo de servidor directamente accesible por parte de las aplicaciones y usuarios en caso de particionar la BD.
- ▶ “mongos” no mantiene ningún elemento de la BD.
 - ▶ Los elementos / datos están almacenados en los “mongod”.
- ▶ Actúa únicamente como “encaminador” de las peticiones.
- ▶ La información sobre el “reparto” efectuado debe obtenerse desde los servidores de configuración.
 - ▶ Una vez obtenida se mantiene en una caché local.
 - ▶ La caché se refrescará cuando su información no consiga encontrar los documentos solicitados.
 - ▶ El reparto se modifica cuando se migran fragmentos para equilibrar la carga.

7.3.4. Servidor de configuración

- ▶ Son procesos “mongod”.
 - ▶ Pero no mantienen datos de la BD.
 - ▶ Mantienen los metadatos: reparto de los “shards”.
- ▶ Otras características:
 - ▶ Están replicados.
 - ▶ Ubicados en máquinas diferentes.
 - ▶ Esto permite agilizar su recuperación cuando algunos de ellos fallen.
 - ▶ Generalmente, los demás seguirán siendo “mayoría”.
 - ▶ Todas las modificaciones sobre los metadatos se aplican mediante un protocolo de “commit” en dos fases.
 - ▶ Asegura consistencia fuerte.
 - ▶ Pero es un protocolo pesado que requiere un alto número de mensajes.

7.3.5. Replicación

- ▶ Se pueden replicar los servidores “mongod” para:
 - ▶ **Mejorar la disponibilidad**
 - ▶ Se soportarán los fallos de manera transparente
 - ▶ **Incrementar el rendimiento**
 - ▶ Las consultas pueden repartirse entre las réplicas
 - Con ello se incrementa la escalabilidad
- ▶ Se adopta un modelo de replicación pasivo
 - ▶ **La réplica primaria es la única que puede ejecutar inserciones, modificaciones y borrados**
 - ▶ Posteriormente propaga el resultado de estas acciones a las demás réplicas, para que también los apliquen
 - ▶ La propagación es asincrónica
 - ▶ **Las réplicas secundarias también pueden ejecutar consultas**
 - ▶ Sin que intervenga la réplica primaria
- ▶ **Transparencia:** Cualquier servidor “mongod” puede ser reemplazado por un “*replica set*” (conjunto de réplicas bajo el modelo pasivo)
 - ▶ Interesante para el “*sharding*”: Cada partición puede ser gestionada por un “*replica set*” diferente.



7.3.5. Replicación

▶ Roles

▶ Se distinguen tres posibles roles

- ▶ Primario: única réplica que atenderá las operaciones que impliquen escritura
 - Normalmente también atiende las consultas
- ▶ Secundario: resto de réplicas que mantienen copia de los datos
 - Se admite que gestionen consultas
- ▶ Árbitro: réplica lógica que no mantiene datos
 - Participa en las votaciones para elegir un primario en caso de fallo

▶ Un conjunto de réplicas no podrá tener más de 50 unidades

- ▶ El número de “votantes” en caso de fallo está limitado a 7
 - El resto de réplicas serán “secundarios sin voto”.

7.3.5. Replicación

▶ Gestión de fallos

- ▶ No se toleran las situaciones de particionado
- ▶ En caso de fallo, MongoDB solo podrá continuar si hay una mayoría de réplicas correctas
 - ▶ Correcta -> Aquella que siga activa y pueda comunicarse con las demás activas en un subgrupo mayoritario
 - ▶ Se recomienda:
 - Tener un número impar de réplicas
 - Pej., 3 réplicas soportan 1 fallo, 5 réplicas soportan 2 fallos, 7 réplicas soportan 3 fallos...
 - Si el número debiera ser par, se añadirá una réplica “árbitro”
 - En caso de partición, permite elegir qué subgrupo será mayoritario
 - ▶ Pej., con 6 réplicas y 1 árbitro... una partición podría dejar aislados dos subgrupos:
 - ▶ A: 3 réplicas
 - ▶ B: 3 réplicas + árbitro
 - ▶ Sólo podrá continuar B, porque tiene 4 de las 7 réplicas (mayoría)
- ▶ La detección del fallo y la sustitución del primario son gestionadas automáticamente por MongoDB
 - ▶ Ni el programador ni el usuario deben preocuparse

7.3.5. Replicación

- ▶ Sincronía en las modificaciones: “*write concern*”
 - ▶ Las operaciones de inserción, modificación y borrado admiten una opción “w” (“*write concern*”):
 - ▶ Especifica cuántas réplicas deben confirmar la realización de la escritura para retornar el control al invocador
 - ▶ Por omisión, su valor es 1
 - Si fuera -1 o 0, asincronía total
 - El valor -1 se despreocupa completamente.
 - El valor 0 reporta, al menos, si ha habido errores en la comunicación con los servidores
 - No se asegura que el primario haya completado la escritura: persistencia no garantizada
 - Excesivamente peligroso en caso de fallo
 - Otras opciones:
 - ‘majority’:
 - ▶ Será 1 cuando no haya replicación
 - ▶ Será una mayoría de réplicas con derecho a voto en caso de replicación
 - ▶ ¡Es el valor recomendable!!
 - Cualquier número concreto
 - ▶ Puede plantear problemas en caso de fallos múltiples
 - ▶ Se debe conocer el número inicial de réplicas para fijar un número razonable ¡Pérdida de transparencia!!!
 - ▶ Problemático: si no hay suficientes réplicas, “colgaremos” la conexión



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



8. Resultados de aprendizaje

- ▶ Al finalizar este tema, el alumno debe ser capaz de:
 - ▶ Conocer el “Teorema CAP” y sus implicaciones en el diseño de servicios distribuidos escalables
 - ▶ Identificar los mecanismos básicos para conseguir escalabilidad
 - ▶ Consistencia relajada
 - ▶ Replicación multi-máster
 - ▶ Distribución de datos
 - ▶ Identificar las claves para tener un sistema distribuido elástico.
 - ▶ Conocer los principios de diseño para evitar la contención en los sistemas distribuidos escalables.



Índice

1. Introducción
2. Teorema CAP
3. Replicación multi-máster
4. Almacenes NoSQL
5. Elasticidad
6. Contención y cuellos de botella
7. Ejemplos de escalabilidad
8. Resultados de aprendizaje
9. Bibliografía



9. Bibliografía

- ▶ Wilson, Jim R., *Node.js the Right Way*. Ed. Jacquelyn Carter, Col. The Pragmatic Bookshelf, 2013.
- ▶ <http://nodejs.org/api/cluster.html> (NodeJS v13.3.0 Manual & Documentation)
- ▶ Ihrig, C., *Pro Node.js for Developers* (capítulo 6: Application Scaling), ed. Apress, 2013.
- ▶ Cirkel, K., *Load balancing Node.js* (<http://www.keithcirkel.co.uk/load-balancing-node-js/>), 2014
- ▶ Robbins, C., *node-http-proxy* (<https://github.com/http-party/node-http-proxy>)
- ▶ Tarreau, V., *HAProxy* (<http://www.haproxy.org/>) (<http://en.wikipedia.org/wiki/HAProxy>)
- ▶ Sysoev, I., *nginx* (<http://nginx.org/en/>) (<http://en.wikipedia.org/wiki/Nginx>)