T6.- Escalabilidad

TSR 2021-22, Juansa Sendra, Grupo B

Introducción

- Visto en CSD
  - Escalabilidad = Capacidad para adaptarse a cambios en la carga sobre el sist.
  - Mecanismos: Distribución procesos, Distribución datos, Replicación, "Caching"
- Relación escalabilidad -- consistencia de datos
  - Consistencia fuerte $\rightarrow$ alta sincronización procesos (bloqueos) $\rightarrow$ No escalable
  - Consistencia débil (relajada) menor necesidad de comunicación $\rightarrow$ escala bien
  - Norma general: $\uparrow$ consistencia $\downarrow$ escalabilidad (hay que estudiar cada caso)
  - Los modelos de replicación clásicos ofrecen consistencia fuerte $\rightarrow$ escalan mal
- Teorema CAP $\rightarrow$ un servicio a escala global no puede ofrecer consistencia fuerte
- Nivel de datos: Las Bases de datos relacionales escalan mal $\rightarrow$ almacenes NoSQL

Teorema CAP

- Partición = Pérdida de conectividad (fallo múltiple de comunicaciones) que deja varios subgrupos de nodos sin posibilidad de intercomunicarse
    - Tolerar la partición $\rightarrow$ todos los nodes siguen trabajando de manera normal
- Un sistema distribuido ha de proporcionar:
    - Imagen de sistema único $\rightarrow$ las réplicas deben mantenir un model de **Consistencia (C)** fuerte (ex. secuencial)
    - Transparencia de fallos: **Disponibilidad (A)** (todos los clientes pueden utilizar el servicio)
        - Deben replicarse todos los servicios proporcionados
        - Debe comportarse igual aunque aparezca una Partición (P) de la red
- Teorema CAP: **En un sistema distribuido no podemos mantener a la vez consistencia fuerte (C), disponibilidad (A = availability), y tolerancia a partición (P) de la red**

Teorema CAP.- Debemos sacrificar una de les tres propiedades CAP

- P partición.- no se admite que la red se particione
  - tratamos de asegurar la conectividad (ex. replicando la red)
  - difícil assegurar conectividad continuada (únicamente en LAN) $\longrightarrow$ no escala
- A disponibilidad.- En caso de partición, adopta el modelo de partición primaria
  - Continuará el subgrupo primario (mayoritario), y el resto se detienen
  - En el grupo mayoritario mantenemos consistencia fuerte, pero perdemos la disponibilidad para los clientes asociados a nodos de los grupos minoritarios
- C consistencia.- Modelo particionable = permite que todos los subgrupos avancen
  - Pierde la consistencia fuerte.- Puede soportar consistencia final (reconciliación entre réplicas) con un diseño adecuado y operaciones conmutativas

Teorema CAP

- En sistemas escalables en tamaño (soporta un elevado número de usuarios) y distancia (desplegado en múltiples centros de dades)
  - Pueden producirse particiones (ej. un "rack" en un centro de datos queda desconectado del resto), y debe tolerarse
  - Se suele sacrificar la consistencia fuerte (C) $\rightarrow$ garantiza disponibilidad (A) y soporta Particiones (P)

Replicación multi-master

- Los dos modelos de replicación clásicos (pasivo/activo)
  - Mantienen consistencia fuerte $\rightarrow$ no escalan bien
- Alternativa escalable: replicación multi-master
  - Basada en el modelo pasivo, pero:
    - Cada petición puede servirse directamente por una réplica master (cada partición usa un master diferente)
    - La respuesta se envía inmediatamente al cliente, las modificaciones se envían al resto de réplicas de forma perezosa (lazy) $\rightarrow$ se responde al cliente sin tener garantías de que otras réplicas han aplicado las modificaciones

Replicación multi-master: ventajas e inconvenientes

- Ventajas
  - Sobrecarga mínima (cada petición únicmaente por una réplica)
  - Altamente escalable (no requiere ningún mecanismo de control de concurrencia)
  - Admite operaciones no deterministas (cada operación se ejecuta en un master)
- Inconvenientes
  - Si falla un master, podemos perder la petición en curso
  - No puede gestionar fallos bizantinos
  - Pueden surgir inconsistencias
    - Diferentes masters pueden servir operaciones concurrentes que modifican algún dato compartido
    - El sistema no garantiza ningún modelo de consistencia (ni siquiera final) $\longrightarrow$ Queda bajo la responsabilidad del programador

Almacenes NoSQL

- Repartir los datos de Bases de Datos Relacionals (SQL) escala mal
  - Necesita mucha sincronización entre los nodos
- Para mejorar la escalabilidad del nivel de datos $\longrightarrow$ Almacenes NoSQL
- Mecanismos básicos para mejorar la escalabilidad (escalar = añadir nodos)
  - Simplifica esquema: (tablas relacionales) tablas clave/valor (mas simples)
    - Simplifica el índice
    - Simplifica el lenguaje de interrogación (y lo hace mas eficiente)
    - Reduce el espacio ocupado por datos (BD en RAM, persistencia por replicación)
  - Elimina transacciones: No permite grupos de sentencies atómicas
    - Bloqueos cortos (imperceptibles)
  - Reparte las tablas (equilibra) dividiéndolas en forma horitzontal (filas) o vertical (atributos)
  - Reparte responsabilidades entre múltiples nodos.

Variantes mas habituales

- Almacenes clave-valor (ej. Dynamo, Voldemort, Riak)
  - tablas clave-valor
  - Escaso soporte para atributos en el valor (no podemos interrogar por atributos)
- Almacén de documentos (CouchDB, MongoDB, SimpleDB...)
  - Esquema compuesto por objetos con un número variable de atributos (en algunos casos anidables)
  - Lenguaje de interrogación basado en restricciones sobre los valores de los atributos
- Almacén de registres extensibles (Bigtable, PNUTs, Cassandra...)
  - Esquemas formados por tablas con un número variable de columnas
    - En algunos casos se organizan en grupos de columnas.

Elasticidad

- Sistema elástico = escalable + adaptable
  - Adaptable = asigna a cada aplicación la cantidad exacta de recursos que necesita
    - de forma dinámica (depende de la carga en cada momento)
    - de forma autónoma (administra los recursos sin intervención humana)

  “Elasticidad es el grado en que un sistema es capaz de adaptarse a cambios en su carga suministrando y reciclando los recursos de forma autónoma, consiguiendo que en cada momento los recursos disponibles cubran la demanda existente con la mayor precisión posible”

Elasticidad en sistemas Cloud

- 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 y del rendimento obtenido
  - Un sistema de actuación para automatizar la reconfiguración de los servicios
    - Reacciona en función del “Service Level Agreement” (SLA) establecido
      - Incorpora nodos cuando la carga aumenta
      - Libera recursos cuando la carga disminuye
      - La reacción debe ser rápida, para no incumplir el SLA

Contención y cuellos de botella

- Un servicio distribuido tiene múltiples componentes
- Debe evitarse la contención en alguno de ellos
  - Evitar que un componente se sature o bloquee a otros, mientras los otros admiten cargas majores
- Causas de la contención:
  - Uso de algoritmos centralitzados en algunas tareas “pesadas”.
    - Centralización únicamente para reducir tráfico en tareas ligeras o infrecuentes
  - Uso de herramientas de sincronización para proteger acceso a recursos compartidos
    - Sección crítica distribuida $\rightarrow$ bloqueos
  - Tráfico excesivo
    - Mala distribución de los recursos $\rightarrow \uparrow$ accesos remotos

Contención y cuellos de botella.- ¿Cómo evitar la contención?

- Ante la “centralización”
  - Para tareas "pesadas" adoptar una solución descentralizada
- En el acceso a recursos compartidos:
  - Serializando los accessos y adoptando un paradigma de programación asincrónica
    - Resulta más fácil gestionar la carga si no hay competición por el uso de los recursos
    - El sistema resultante es más eficiente
      - Menos cambios de contexto
      - Menor carga para el sistema operativo y el middleware
- Para sistemas con tráfico excesivo:
  - Replicar los recursos y mantenerlos consistentes
  - Los accesos remotos se transforman en accesos locales.

Examplo de escalabilidad.- módulo "cluster" de NodeJS

- Una aplicación NodeJS tiene únicamente un hilo de ejecución
  - ¿qué hacemos si, por razones de escalabilidad, queremos varias instancias de un componente?
  - Hasta ahora hemos creado varios procesos en el mismo o distintos nodos, utilizando el patrón client/broker/workers
  - Otra opción es el módulo cluster de NodeJS. Facilita:
    - La creación y gestión de un pool de trabajadores (processos de Node)
    - El reparto o equilibrado de la carga del servicio entre los procesos
- El módulo Cluster permite aprovechar los sistemas multiprocesador, mediante la ejecución de varios processos Node
  - Reparte la carga de servicio entre los procesos que se ejecutarán en los distintos procesadores (workers)
  - Los workers comparten todos los ports asociados al servicio que prestan.

Módulo Cluster.- esquema básico de un pool de trabajadores

- Un proceso master crea workers mediante cluster.fork (ej. tantos como núcleos)
- El mismo código se usa para master (rama if) y worker (rama else)
- El master supervisa a los workers (escucha eventos del objeto cluster)

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

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

Módulo Cluster.- Código de un cluster de servidores http

- Varios workers (servidores http) escuchan en el port 8000

```
const cluster= require('cluster')
const http   = require('http')
let numCPUs  = require('os').cpus().length

if (cluster.isMaster) {  // code of the master one
   for (let 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)
}
```

Módulo Cluster.- eventos del objeto Cluster

- **fork**.- se crea un nuevo worker
- **online**.- mensaje de un worker indicando que empieza a ejecutarse
- **listening**.- un worker ejecuta un `listen()`
- **disconnect**.- se cierra el canal IPC de un worker (por exit, kill, o disconnect)
- **exit**.- un worker finaliza ejecución

```
var timeouts= []
cluster.on('fork', function(worker) { // new worker must listen on-time
   timeouts[worker.id] = setTimeout(()=>{console.error('Error ...')}, 2000)
})
cluster.on('listening', function(worker, address) { // on-time
   clearTimeout(timeouts[worker.id]) // cancel timeout
})
cluster.on('exit', function(worker, code, signal) { // worker has finished
   console.log('worker %d died (%s). restarting...', worker.process.pid, signal || code)
   cluster.fork()
})
```

Módulo Cluster.- Objetos

- Objeto cluster.workers
  - Es un hash (diccionario) 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 master puede enviar mensajes a un worker determinado o a todos
- Objectos worker (uno por worker, accesible mediante cluster.worker)
  - Atributos
    - id (identificador único del worker en cluster.workers)
    - process (su procés)
    - exitedAfterDisconnect (para distinguir entre exit voluntario o aborto)
  - Métodos.- se invocan mediante su atributo process: process.send(...)
    - send(envio de mensajes al master), disconnect, kill, ...
  - Eventos: message online listening disconnect exit error

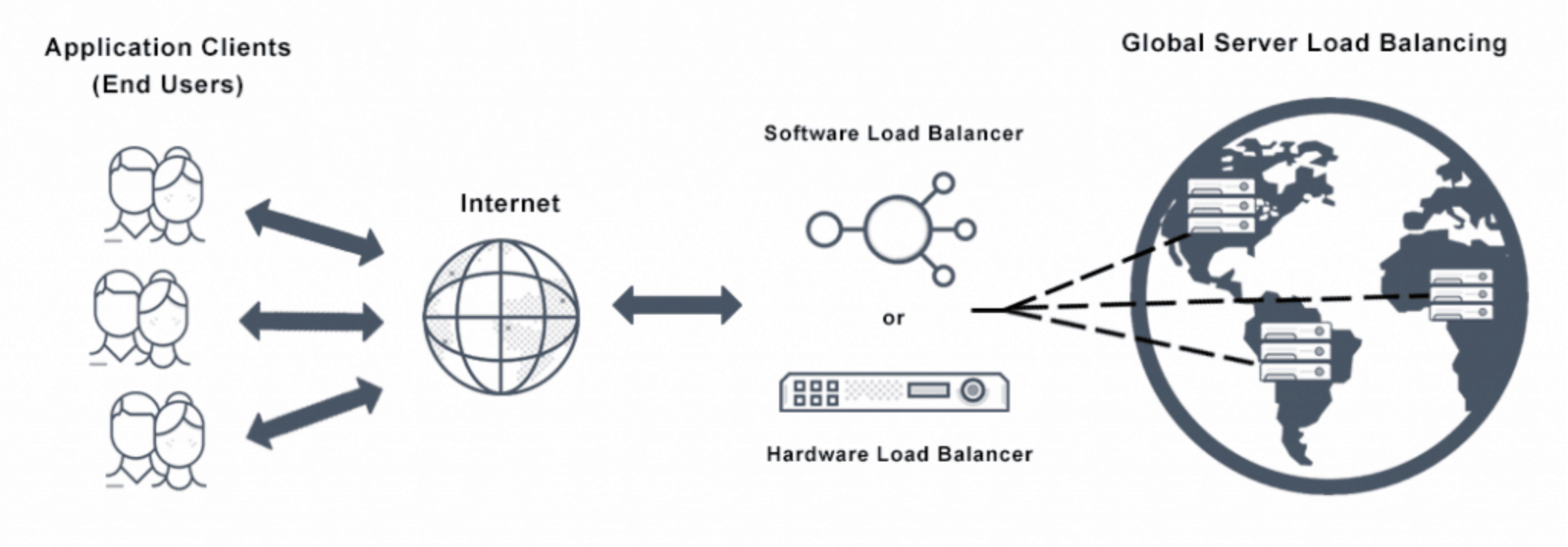
Módulo Cluster.- Mensajes master ↔ workers (evento 'message')

- Pueden utilizarse, por ejemplo, para contabilizar les peticiones de servicio:

```
const cluster = require('cluster')
let 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 (let i=0; i < numCPUs; i++) cluster.fork()
   for (let 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)
}
```

Ejemplo de escalabilidad.- escalabilidad en varios ordenadores

- Pocos clientes → aplicación node tradicional (monoprocesador)
- ↑↑ clientes → módulo cluster (explota multiprocesador)
- ↑↑↑ clientes → repartir (equilibrar) carga entre varios nodos
  - node-http-proxy (https://github.com/nodejitsu/node-http-proxy)
  - HAProxy (http://www.haproxy.org/)
  - nginx [engine x] (http://nginx.org/en/)



node-http-proxy.- Servidor proxy para aplicaciones HTTP y Node

- Código abierto
- 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)
```

HAproxy.- Servidor proxy para aplicaciones TCP y HTTP

- Código abierto, dirigido por eventos, también paralelismo en multi-núcleo
- Proporciona equilibradp de carga, alta disponibilidad y fiabilidad
- Ejemplo.- Servicio http de node, y esquema básico de configuración para el mismo

```
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)
```

```
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
```

Nginx.- Servidor proxy inverso para aplicaciones HTTP

- Equilibra la carga entre servidores http en diferentes máquinas
- Código abierto, dirigido por eventos, fácilmente extensible
- Lo utilizan más del 25% de los websites de mayor carga (agosto 2018)
- Esquema básico de configuración (load balancing, round robin). Ej.

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

Ejemplo de escalabilidad.- MongoDB (almacén de documentos)

(para aclarar conceptos indicamos el `concepto equivalente en el modelo relacional`)

- Mantiene un conjunto de colecciones (Base de Datos) a repartir entre varios nodos
  - Colección (`Tabla`) = conjunto de objetos (documentos) estructurados
  - Objeto = identificador (`Clave primaria`) + múltiples atributos
    - Los atributos pueden ser tipos simples, vectores, objetos (con atributos)...
    - Cada atributo puede interpretarse como un par <clave (nombre), valor>
- Esquema flexible $\rightarrow$ Los objetos de una misma colección no tienen necesariamente idéntica estructura
- Para conseguir escalabilidad utiliza particiín horizontal
  - Dividimos el rango de claves primaries en fragmentos
  - Las filas de cada fragmento se guardan en una máquina diferente
- Para mejorar disponibilidad utilizando replicación pasiva

MongoDB.- Componentes

(Como BD ejemplo: BD de alumnos universitarios. DNI = clave primaria)

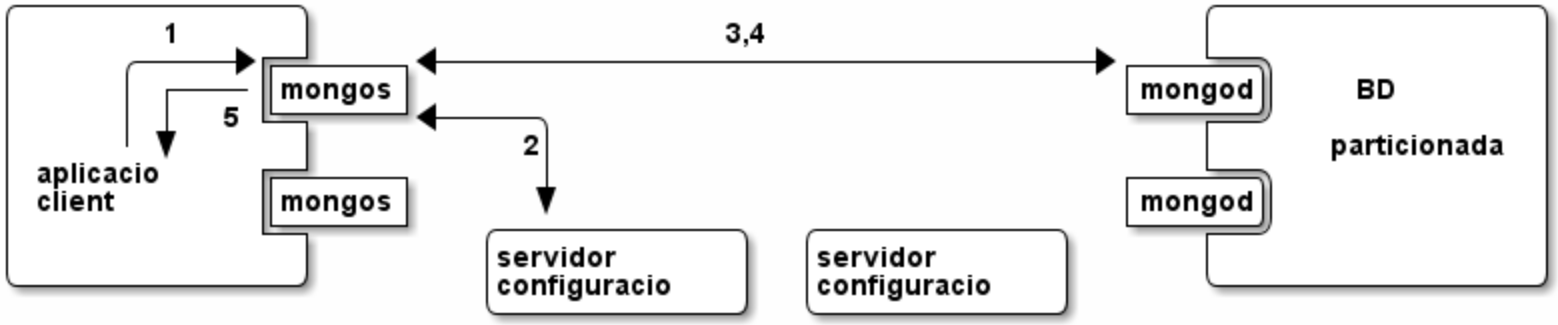
- **mongod** (servidor de datos)
  - un mongod por máquina
  - Cada uno mantiene un subconjunto de filas de la BD (partición horizontal)
    - dividimos el rango de claves primarias en N fragmentos: los documentos en cada uno de los subrangos va a un mongod diferente
- **mongos** (mongo shell)
  - interfaz JavaScript para acceder a los servidores
  - permite obtener ayuda (db.help), proporciona entorno interactivo para pruebas
- **servidores de configuración**
  - Guardan los metadatos de la BD, o sea:
    - qué filas forman cada partición
    - en qué nodo se situa cada partición

MongoDB.- Configuraciones

- Una única máquina
  - Simple, pero no explota las ventajas de escalabilidad
  - Un servidor mongod mantiene toda la BD (debe arrancarse primero)
  - Para interactuar con la BD, shell mongo o programa + driver de mongoDB
- BD repartida en N máquinas (BD particionada) (Lo asumimos)
  - Un proceso mongod por máquina, particionado horizontal
    - Ej: 26 nodos, en cada uno los DNIs con la misma letra
  - Un proceso mongos por máquina
    - Actua como interfaz con la aplicación cliente (enruta las peticiones)
      - Si lo sabe (info en caché), enruta la petición al proceso mongod
      - Si no, consulta servidor de configuración (averigua mongod) y enruta
  - Servidores de configuración
    - forman un "conjunto de réplicas" (replica-set) $\longrightarrow$ replicación pasiva

MongoDB.- Ejemplo funcionamiento BD particionada

1. La aplicación cliente invoca al servidor “mongos” más próximo
2. mongos pide al servidor de configuración los metadatos de distribución
    - De ese mode averigüa en qué máquina o máquinas están los datos requeridos
3. mongos redirige la petición al servidor “mongod” apropiado
4. El servidor “mongod” ejecuta la petición y retorna la respuesta a “mongos”
5. Mongos traslada la respuesta al cliente



MongoDB.- Detalles mongod

- El particionado horizontal se denomina sharding (shard = datos en un mongod)
  - Cada shard se divide en fragmentos (chunks) $\leq$ 64MB (si $>$ se divide en dos)
- Un proceso equilibrador de carga vigila la cantidad de fragmentos por mongod
  - Si desequilibrio, migra fragmentos hasta equilibrio. Para migrar un fragmento:
    - Mantiene los accesos al servidor origen, inicia copia al mongod destíno
    - Al acabar actualiza metadatos (servidors conf.) y borra fragmento origen
- Cada mongod utiliza journaling para evitar inconsistencias
  - Cada modificación se escribe primero en un journal y después se aplica a la BD
    - anota tipo de operación, datos que modificará, y resultado
  - Si mongod falla durante la modificación
    - El cliente no obtiene respuesta
    - Cuando se recupera, lee el journal y aplica a la BD toda operación incompleta
  - Acelera recuperación y evita escrituras incompletas (garantiza integridad BD)

MongoDB.- Detalles mongos

- Único tipo de servidor directamente accesible por parte de las aplicaciones y usuarios
- No mantiene ningún elemento de la BD (datos exclusivamente en mongod)
- Actua únicamente como encaminador de las peticions
- Obtiene de los servidores de configuración la información sobre el reparto de les datos
    - Mantiene esa información en una memoria caché local
    - Refresca la memoria caché cuando su información no permite encontrar el documente
    - El reparto se modifica cuando se migren fragmentos

MongoDB.- Detalles Servidores de configuración

- Son procesos mongod 'especiales'
  - mantienen metadatos sobre el reparto de shards, no datos
- Están replicados (réplicas en máquinas diferentes)
  - recuperación rápida en caso de fallo de otra réplica
- Todas las modificaciones sobre los metadatos se aplican mediante un protocolo de “commit” en dos fases
  - Asegura consisténcia fuerte
  - Es un protocolo pesado que requiere un alto número de mensajes

Mongod.- Replicación de servidors mongod

- La replicación de servidores mongod mejora:
  - la disponibilidad.- soporta los errores de forma transparente
  - el rendimento (escalabilidad).- las consultas pueden repartirse entre réplicas
- Adopta un modelo de replicación pasiva (replica-set), con 3 roles:
  - primario.- única réplica que atendrá a las operaciones que impliquen escritura
  - secondario.- mantienen copia de los datos y pueden gestionar consultas
  - árbitro.- réplica lógica sin datos $\rightarrow$ pero con voto para elegir primario
- Transparencia de replicación: podemos cambiar cualquier mongod por un replica-set
  - cada partición puede ser gestionada por un 'replica-set' diferente

Mongod.- Gestión de defectos

- Requiere mayoria de réplicas activas
  - Para soportar N defectos simultáneos 1+2N réplicas (ej. 5 para 2 defectos)
- En caso de partición de red sólo puede continuar el subgrupo mayoritario
  - Recomendable número impar de réplicas (o añadir una réplica árbitro)
- En caso de error del primario debe elegirse ptro
  - Detección de errores y sustitución del primario gestionados automáticamente
- Limites: Un replica-set no puede tener más de 50 unidadess, y sólo votan 7

Mongod.- Sincronia en les modificaciones (write-concern)

- Las escrituress (inserción, modificación, borrado) admiten una opción `w`
  - w (write-concern) = cuántas réplicas deben confirmar la realización de la escritura para retornar el control al invocante
- Valores posibles de w (por defecto 1)
  - -1 se despreocupa completamente
  - 0 reporta si han habido errors en la comunicación con los servidores
  - majority = mayoria de réplicas con derecho a voto, o bien 1 si no replicación
  - Cualquier número concreto >0. Cuelga la conexión si no quedan bastantes réplicas
    - para elegir un valor razonable se debe conocer el número inicial de réplicas
    - Pérdida de transparencia!!!
- 0 o -1 implica asincronia total $\longrightarrow$ no garantiza persistencia (peligroso)
- El valor recomendado es `majority`