CDA. Clusters en centros de datos

Computación distribuida y clustering
Balanceo de Carga
Alta disponibilidad
Clusters de altas prestaciones

Centros de datos
3º Grado en Ingeniería Informática
ESEI

25 de octubre de 2018

Contenido

Computación distribuida

Los clusters son agrupaciones de computadores [nodos] (junto con las infraestructuras de comunicación y almacenamiento asociadas) destinados a trabajar de forma conjunta para ofrecer un determinado servicio y/o realizar operaciones de cómputo específicas.

Dependiendo de la finalidad y objetivos finales de un cluster, tenemos:

1. **Clusters de balanceo de carga:** conjunto de nodos que se reparten la prestación de un servicio determinado (servir aplicaciones web, dar soporte a un gestor de BD, etc) *[load balancing clusters]*
   - Punto clave: ”repartición” del trabajo
   - **Objetivo:** atender el máximo número de peticiones al servicio
2. **Clusters de alta disponibilidad:** conjunto de nodos que garantiza la disponibilidad de un servicio determinado aún en el caso de fallos y/o caídas de algún elemento *[failover clusters]*
   - Punto clave: tolerancia y recuperación ante fallos (*failover*)
   - **Objetivo:** garantizar la prestación (y consistencia) del servicio
3. **Clusters de alto rendimiento:** conjunto de nodos que trabajan conjuntamente en tareas de cálculo intensivo (renderizado de gráficos, análisis de datos, predicción) *[high performance clusters]*
   - Punto clave: procesamiento distribuido
   - **Objetivo:** maximizar el rendimiento y la capacidad de cálculo

Balanceo de carga

Balanceadores de carga: dispositivos hardware y/o software conectados a un conjunto de nodos de procesamiento entre los que reparte las peticiones recibidas por parte de los clientes

Ejemplos:
- granja de servidores web ←
  Ejemplo: HAproxy (`http://haproxy.1wt.eu/`)
- dos o más conexiones/tarjetas de red
  Ejemplo: bonding (`http://en.wikipedia.org/wiki/Channel_bonding`)
- ”balanceo” (migración) de procesos
  Ejemplo: LinuxPMI (`http://en.wikipedia.org/wiki/LinuxPMI`)

Posible solución a problemas de escalabilidad:

- **Escalabilidad vertical** *(scale up)*: ”mejorar” las máquinas/nodos que prestan un servicio añadiendo más recursos (memoria, capacidad CPU, etc)
- **Escalabilidad horizontal** *(scale out)*: agregar más máquinas/nodos para prestar el servicio (uso de balanceadores de carga)

Importante: No ofrece alta disponibilidad (pero sí es un mecanismo para conseguirla)

Conceptos:

- **Servidor virtual** servidores/servicios que se ofrecen a los clientes desde el balanceador (tb. nodo director)
- **Servidor real** servidores/servicios que realmente atienden/procesan las peticiones (nodos del cluster)

Alternativas

Balanceo de carga por DNS [≈ balanceo implícito]

- Se vincula un servicio a un nombre de dominio DNS
- En el servidor DNS se le vinculan distintas direcciones IP (servidores "reales") a ese nombre de dominio
- Se configura servidor DNS para que reordene la lista de direcciones devuelta cada vez que los clientes resuelvan el nombre de dominio del servicio ofrece (balanceo implícito) [*DNS Round Robin*]

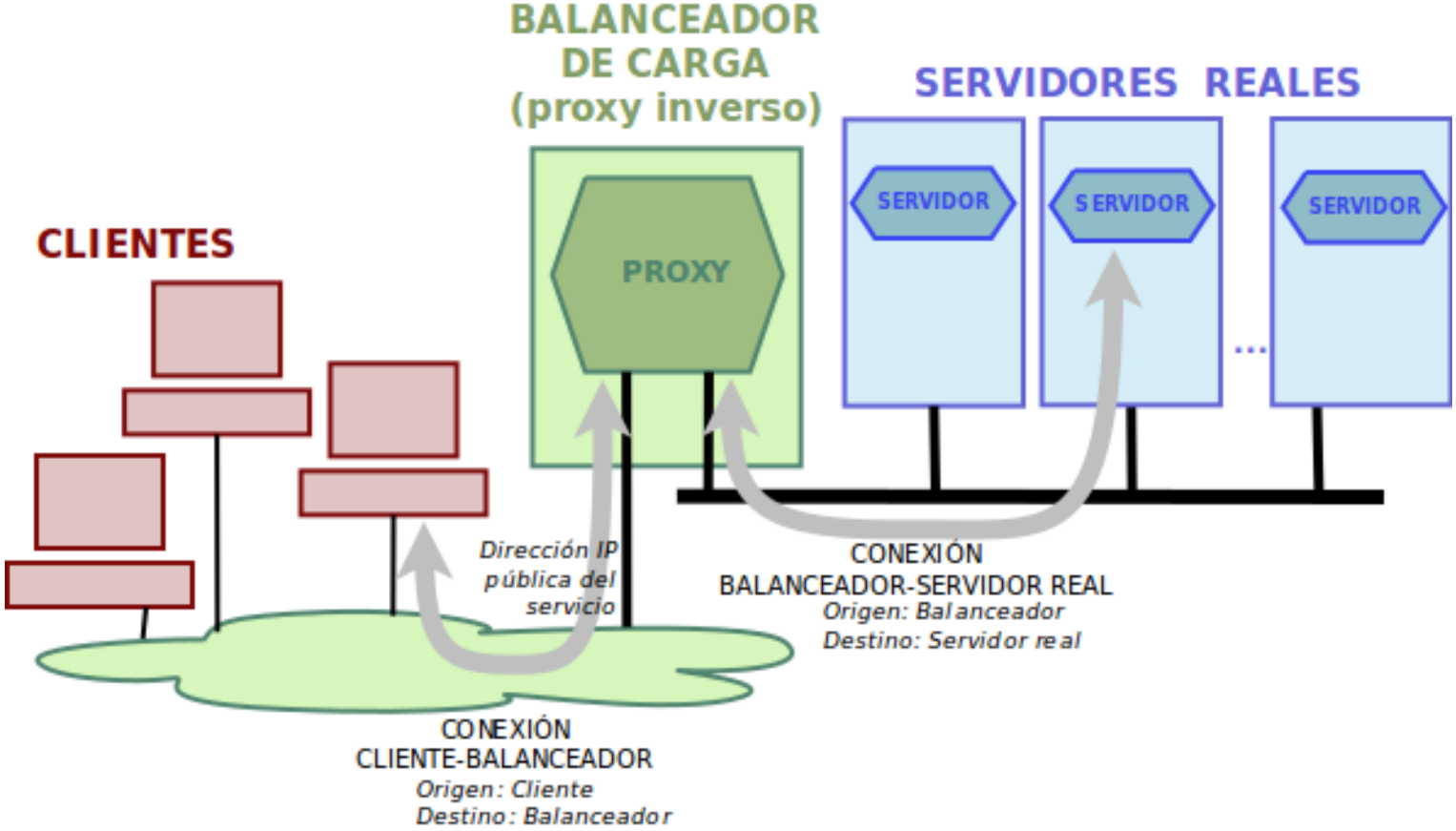
Ventaja: no requiere contar explícitamente con equipo/s dedicado al balanceo
Inconv.: balanceo no equitativo, distorsiones debidas a las caches de nombres.

Balanceo a **nivel de aplicación**

Mediante proxies "inversos"

- **Retransmiten** a **nivel de aplicación** peticiones/respuestas a/desde servidores reales
- Opcionalmente pueden alterar el contenido de las peticiones/respuestas
  → Capaces de manipular datos de capa de aplicación (ej. cabeceras HTTP)
- Ejemplos:
  - HAProxy (`http://www.haproxy.org/`)
  - `mod_proxy` y `mod_proxy_balancer` de Apache
  - Varnish (`http://www.varnish-cache.org/`)

Balanceo de carga a nivel de aplicación (proxy inverso)



Balanceador mantiene dos conexiones: { EXTERNA: Cliente – Balanceador (proxy); INTERNA: Balanceador (proxy)– Servidor Real

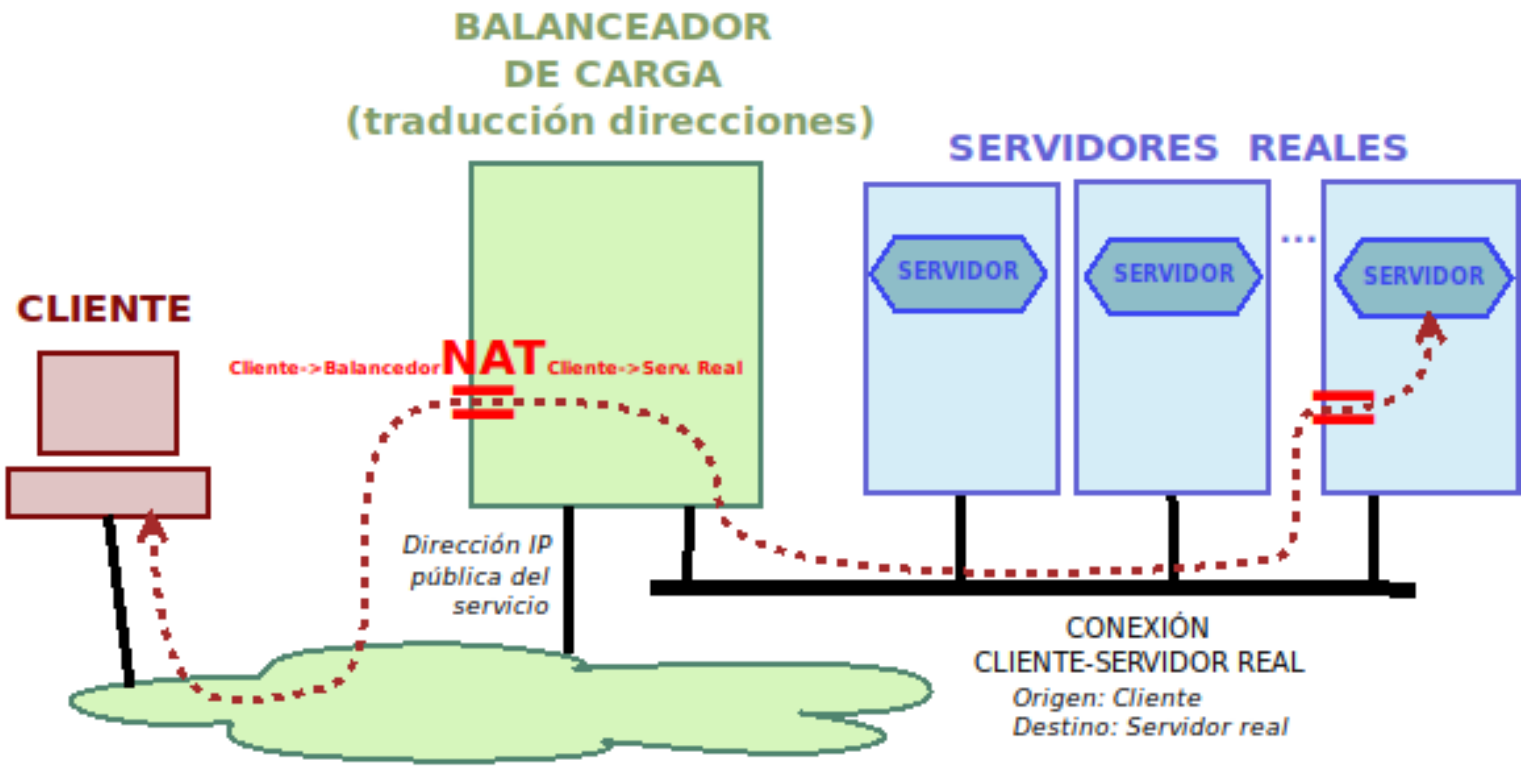
1. Proxy inverso del *Balanceador* recibe la petición del *Cliente* `[conex. externa]`
2. *Balanceador* selecciona *Servidor Real*
3. *Balanceador* **retrasmite petición** hacia *Servidor Real* `[conex. interna]`
4. *Servidor Real* envía la respuesta a *Balanceador* `[conex. interna]`
5. *Balanceador* **retrasmite respuesta** hacia *Ciente* `[conex. externa]`

Balanceo a **nivel IP**

Mediante reescritura de los paquetes IP y/o mediante encapsulado (*tunneling*) de tráfico IP

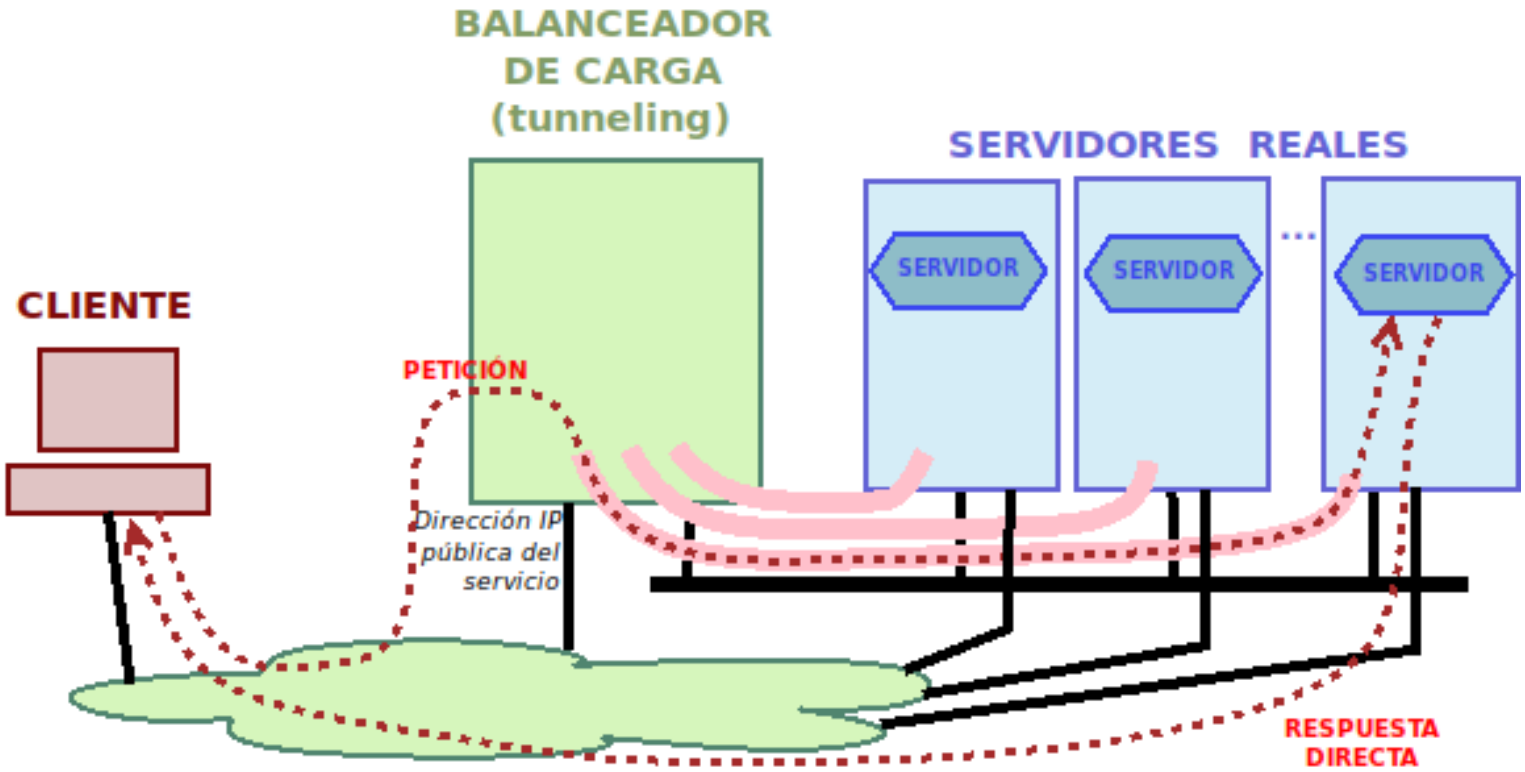
- **Ejemplo:** Linux Virtual Server (LVS)
  `http://www.linuxvirtualserver.org/`
  - Forma parte del kernel de Linux (hace uso del framework NETFILTER)
  - Usa NAT o encapsulado de paquetes IP para redirigir las peticiones a los servidores reales
  - Métodos:
    - **LVS-NAT** mediante traducción de direcciones (NAT)
      [balanceador implementa NAT: "traduce" peticiones entrantes + respuestas salientes]
      `http://www.linuxvirtualserver.org/VS-NAT`
    - **LVS-TUN** mediante *tunneling*
      [encapsula los paquetes de petición (tunel IP) + respuesta directa al cliente]
      `http://www.linuxvirtualserver.org/VS-IPTunneling.html`
    - **LVS-DR** *direct routing*
      [redirige petición sobre Ethernet (dir. MAC) + respuesta directa al cliente]
      `http://www.linuxvirtualserver.org/VS-DRouting.html`

Balanceo de carga a nivel IP LVS-NAT (modo NAT)



1. *Balanceador* selecciona *Servidor Real*
2. Emplea D-NAT *(Destination Address Translation)* para **reescribir dirección destino de las peticiones** y redirigirlas al *Servidor Real* seleccionado
3. Respuestas de *Servidor Real* pasan por *Balanceador* para restaurar la dirección original y enviarlas al *Cliente*

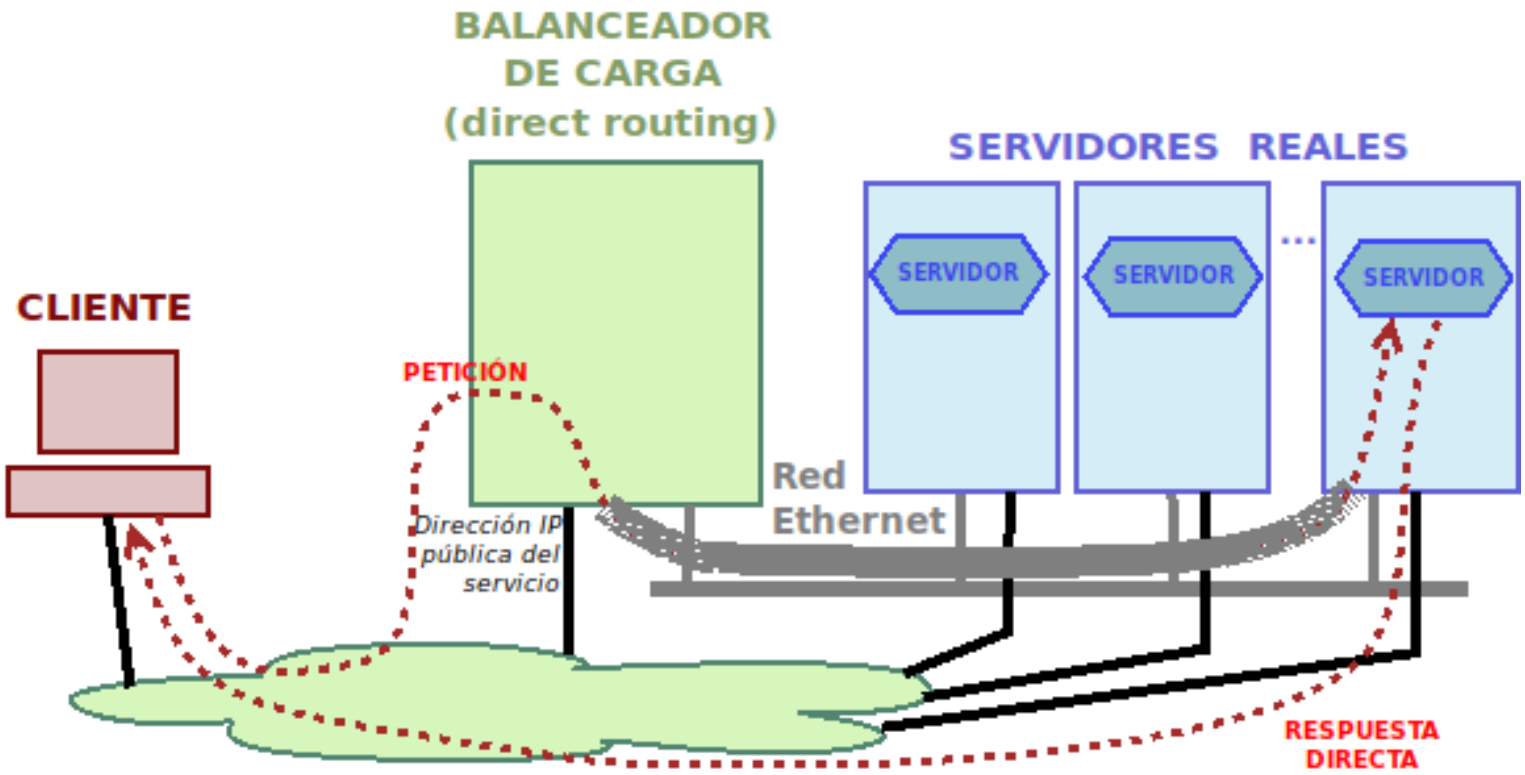
Balanceo de carga a nivel IP LVS-TUN (modo túnel)



Balanceador mantiene un **tunel IP** [encapsula paquetes IP dentro de otros paquetes IP] con cada *Servidor Real*

1. *Balanceador* selecciona *Servidor Real*
2. **Encapsula** las **peticiones** en paquetes IP que atraviesan el túnel hacia el *Servidor Real* seleccionado
3. *Servidor Real* tiene acceso a la red externa, envía las respuestas directamente al *Cliente*

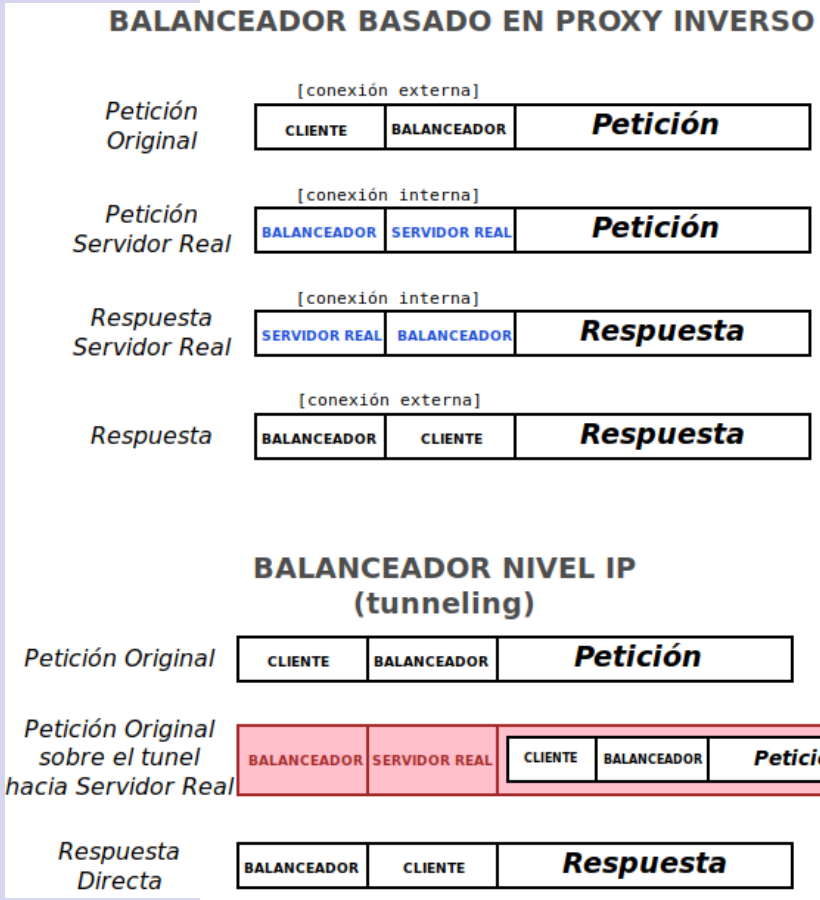
Balanceo de carga a nivel IP LVS-DR (encaminamiento directo)

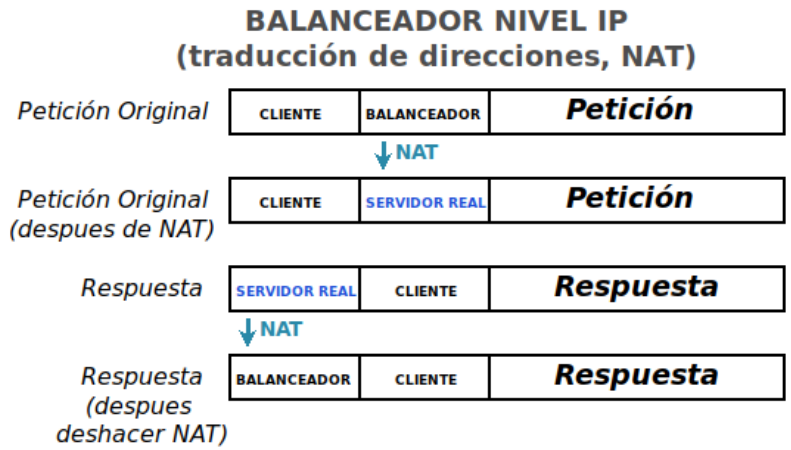


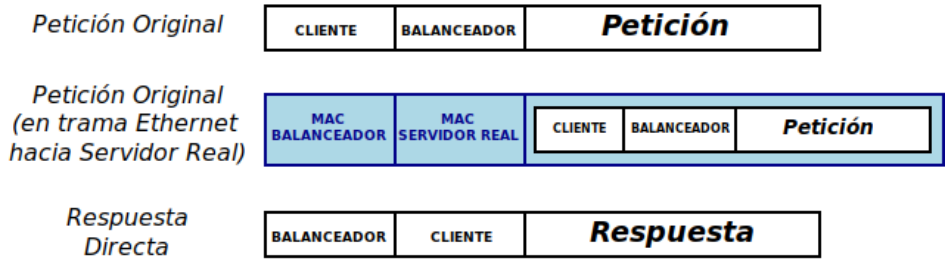
Balanceador accede a nivel Ethernet a la red donde se ubican los *Servidores Reales*

1. *Balanceador* selecciona *Servidor Real*
2. *Balanceador* envía **petición dentro de tramas Ethernet** creadas con la dirección MAC del *Servidor Real* seleccionado
3. *Servidor Real* tiene acceso a la red externa, envía las respuestas directamente al *Cliente*

Mensajes intercambiados







Estrategias de reparto de carga

Round-Robin: alterna el envío de peticiones a servidores reales de forma sucesiva

- intenta reparto equitativo de las peticiones
- asumen que todos los servidores reales tienen la misma capacidad de procesamiento y que las respectivas cargas de trabajo son similares

Weighted Round-Robin: asigna un peso a cada servidor real, el reparto de peticiones es proporcional al peso de los servidores

- reparto de las peticiones proporcional a la capacidad de procesamiento de los servidores reales

Least-Connection: dirige las peticiones al servidor real con menos conexiones abiertas en cada momento

- puede combinarse con pesos (*Weighted Least-Connection*)

Asignación estática: asignación fija de servidores reales en base a direcciones IP origen (*Source Hashing Scheduling*)

- opcionalmente, también info. de IP destino (*Destination Hashing Scheduling*)

Otros: asignación en base a características de las peticiones recibidas (URLs, COOKIES, etc)

- sólo en balanceadores basados en proxy (requiere info. de aplicación)

Requisitos adicionales I

Persistencia de conexiones

En determinados protocolos/aplicaciones puede ser necesario mantener información de conexiones previas

- Sin persistencia de conexiones, cuando un cliente solicite una nueva conexión, el balanceador escogerá un **nuevo servidor real que puede ser distinto** al anterior

Solución 1: asegurar que el mismo servidor real responderá SIEMPRE a todas las conexiones de un determinado cliente

- Implícito en estrategias de asignación estáticas
- Normalmente se implementa con balanceadores de nivel de aplicación

Solución 2: replicar la información de las conexiones entre todos los nodos del cluster de balanceo

- Mismo efecto usando almacenamiento compartido para esa información de conexiones

Requisitos adicionales II

Ejemplo: *sticky sessions* en HTTP

- HTTP es un protocolo sin estado (una conexión por cada petición)
- Ciertas aplicaciones Web requieren estado (sesiones)
  → **Solución**: uso de cookies (identificadores únicos)
  - Asignadas por servidor en la respuesta HTTP a la primera petición (param. `Set-Cookie`)
  - Sucesivas peticiones HTTP del cliente (mientras no se cierra sesión) incluyen ese ID (param. `Cookie`)
  - Servidor HTTP usa esos identificadores únicos para mantener traza de la info. vinculada con la sesión de un cliente concreto
- Uso de *cookies* requiere que balanceador de carga asegure que las peticiones de un mismo cliente siempre se asignarán al mismo servidor Web real
  - normalmente supondrá reescribir las cabeceras HTTP
- Otra alternativa: *session replication*
  - Replicar datos particulares de cada conexión en otros servidores reales o en un almacenamiento compartido

Alta disponibilidad

Un cluster de **alta disponibilidad** (HA: *high availability*) es un conjunto de dos o más nodos que se garantiza que ante el fallo en uno de ellos no se detendrá el servicio que ofrecen en conjunto

- **Idea base:** contar con nodos redundantes que asuman el servicio cuando algún componente falla
  Tipos de interrupciones/paradas:
  - Paradas previstas: mantenimiento, actualización, reparación, ...
  - Paradas imprevistas: desastres naturales, fallos hardware ó software, ...
- Cluster HA es capaz (sin intervención humana) de:
  1. detectar los fallos (hardware o software)
  2. mantener el servicio (retomándolo/iniciándolo en otro nodo)
  3. garantizar la integridad de los datos
- **Objetivo:** evitar los puntos únicos de fallo (SPoF: *single point of failure*)
  $\rightarrow$ garantizar **tolerancia a fallos sin** provocar **inconsistencias** de datos

Uso

Clusters HA suelen usarse para dar soporte a servicios/aplicaciones críticas para una organización que no pueden verse interrumpidas

- Alto coste de *downtime*
- No necesariamente aplicaciones/servicios con grandes requisitos de cómputo, sí requieren **disponibilidad** y **tolerancia a fallos**
- Ejemplos: bases de datos críticas, aplicaciones web de comercio electrónico, sistemas de ficheros compartidos, servidores de correo, ...

Requisitos clusters HA y métricas

Fiabilidad (*reliability*): capacidad del sistema de producir salidas correctas durante un tiempo determinado

- Ante fallos un sistema fiable no continúa de forma descontrolada generado resultados incorrectos/inconsistentes
- Caracterizada por la métrica MTBF (*mean time between failures*) [tiempo medio entre fallos]

Disponibilidad (*availability*): capacidad del sistema de estar operativo durante un tiempo determinado

- Caracterizada por % de tiempo en que el sistema está disponible para sus usuarios
- Grados altos de disponibilidad suponen contar con mecanismos para que le sistema continúe operativo aún ante la presencia de fallos o paradas imprevistas (tolerancia a fallos/*failover*)

Facilidad mantenimiento (*serviceability*): simplicidad y eficacia con la que el sistema puede ser reparado y/o mantenido en condiciones de operación (recuperación de fallos, reparaciones, actualizaciones, tanto software como hardware)

- Caracterizada por la métrica MTTR (*Mean Time To Recovery*) [tiempo medio de recuperación/reparación]

Tríada **RAS**: `http://en.wikipedia.org/wiki/Reliability,_availability_and_serviceability_(computer_hardware)`

Configuraciones típicas (I)

Mecanismo fundamental → **redundancia** (y control de la misma)

- Redundancia hardware: replicación de componentes [procesamiento, almacenamiento, comunicaciones]
- Redundancia software: replicación componentes software, ejecución simultánea [replicación] de procesos, logs de sincronización
- Redundancia de datos: réplicas/copias de seguridad, sincronización
- Administración de la redundancia:
  - Software específico para administrar los componentes redundantes y :
    1. asegurar la continuidad y correcto funcionamiento en caso de la caída de algún elemento
    2. garantizar la consistencia e integridad de los datos/resultados
  - **Ejemplo:** LinuxHA (`http://www.linux-ha.org/`)

Configuración **Activo-Pasivo**

- Los servicios/aplicaciones se ejecutan sobre un conjunto de nodos activos (al menos uno)
- Otro conjunto de nodos pasivos (al menos uno) actúan como respaldo de los servicios ofrecidos (redundancia)
- Nodos pasivos sólo entran en funcionamiento ante fallo de los nodos activos

Esquema más sencillo (configuración simple)

Configuraciones típicas (II)

Configuración **Activo-Activo**

- Todos los nodos actúan como servidores activos de los servicios/aplicaciones
- Cualquier nodo puede servir como respaldo ante fallos en los demás nodos

Mayor aprovechamiento de los recursos

- Es posible combinarlos con componentes de balanceo de carga para repartir la carga entre los nodos.

En ambos casos

- El fallo de un nodo supone que otro nodo/s pasa a hacerse cargo del servicio/aplicación
  - *Failover:* capacidad de recuperarse automáticamente de un fallo en un nodo desplegando el servicio/aplicación en otro nodo
- Se requieren mecanismos para ”sondear” el estado de los nodos para detectar los fallos y actuar
  - *Heartbeat* (latido): mecanismo mediante el cual la infraestructura del cluster HA se comunica periódicamente con los nodos del cluster para conocer su estado
  - Si un nodo activo no responde al ”latido” un nodo de respaldo (activo o pasivo) lo reemplaza y pasará a ocuparse del servicio/aplicación
  - Suele implementarse mediante conexiones de red dedicadas (red *heartbeat* complementaria a la red de comunicaciones)

Mantenimiento de la integridad

La recuperación automática de servicios/aplicaciones debe garantizar la integridad (consistencia) de los datos.

Split-brain

1. Un nodo activo del cluster HA no responde a los *heartbeats*, pero sigue activo
   - Típicamente por fallo de comunicación en la red *heartbeat*
2. Sistema asume que ha fallado y otro nodo asume su tarea
3. Ambos nodos están activos simultáneamente realizando la misma tarea y creen que son los únicos que la realizan (”cluster partido”)
4. Si se usan datos compartidos estos pueden corromperse por el uso simultáneo
5. Posibilidad de *split-brain* crece a medida que crece nº nodos en cluster

Soluciones *split-brain*

Necesidad de mecanismos de *fencing* (vallado) que protejan los recursos compartidos en casos de funcionamiento anómalo

- **Esquemas de *quorum***: prevención *split-brain* mediante mecanismo de votación
  - Cada nodo tiene un voto y la decisión de cual de los nodos puede ejecutar su tarea sobre un recurso compartido exige se tenga la mayoría de votos
  - Con 2 nodos en competición ninguno tendrá mayoría
- **STONITH** (*shot the other node in the head*) mecanismo automático de recuperación/desbloqueo ante un *split-brain*
  - Aísla al nodo fallido para que no pueda dañar el cluster
  - Implementación en Linux-HA envía comando para apagar nodo fallido

Linux-HA

Proyecto de código abierto que provee la infraestructura para el despliegue de clusters HA

- Disponible para GNU/Linux, FreeBSD, OpenBSD, Solaris y MacOS.
- URL: `http://www.linux-ha.org/wiki/Main_Page`

Componentes principales

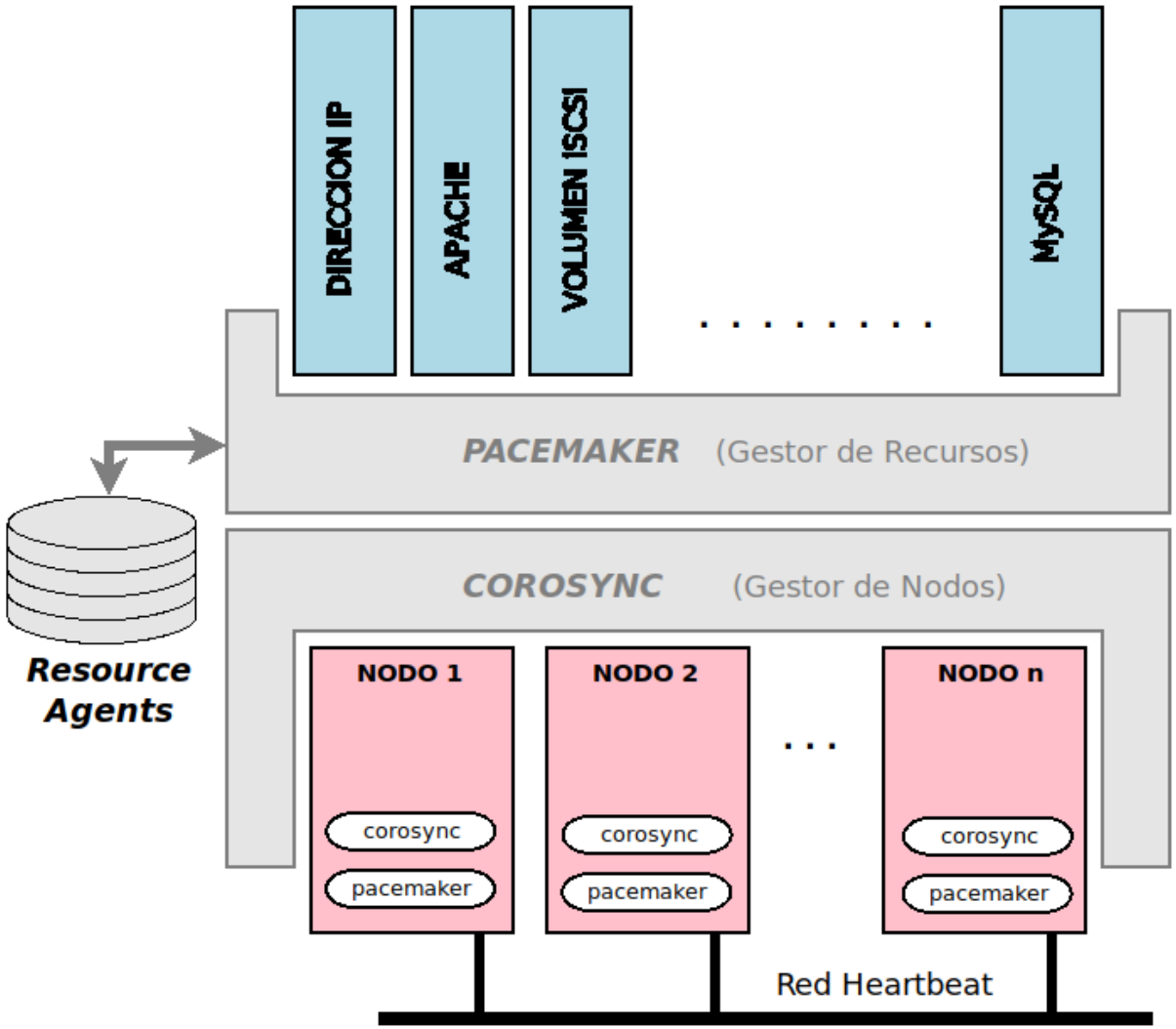
heartbeat/corosync Servicio que proporciona la infraestructura del cluster HA (comunicación y pertenencia) [*cluster messaging layer*]

- Permite que los procesos en ejecución sobre el cluster HA conozcan el estado (presente o ausente) de otros procesos con la misma finalidad (detección de nodos caídos mediante latidos/*heartbeats*)
- Da soporte al intercambio de información en procesos del cluster HA

pacemaker Gestor de recursos del cluster [*cluster resource manager*]

- Responsable de la gestión de las máquinas y servicios del cluster HA
- Da soporte al inicio, parada y reinicio de los componentes del cluster HA
  - Asistido por ”agentes” de control de recursos (*resource agentes*) específicos para cada servicio/aplicación
- Soporta los mecanismos de *quorum* y control de acceso a recursos compartidos

Componentes Linux-HA



Computación de altas prestaciones

Clusters de alto rendimiento (HPC: *High-performance computing)*)

- Utilizado en aplicaciones con requisitos de cálculo intensivos:
  - Simulación científica, renderización de gráficos, modelos de predicción meteorológica, minería de datos (Big Data), ...
- Conformado por un conjunto de nodos (habitualmente con un S.O. específico para HPC) junto con una infraestructura de comunicación de alta velocidad,
- Los nodos del cluster colaboran de forma coordinada en la ejecución de un determinado proceso/procesos concreto con alta demanda de cálculo computacional
- Habitualmente emplean librerías específicas de programación paralela/distribuida
  - Librerías de paso de mensajes como MPI [*Message Passing Interface*] (ejemplos: OpenMPI, MPICH)
  - Entornos Map-Reduce (ejemplos: Apache Hadoop)
  - Multiprocesamiento de memoria compartida (ejemplos: OpenMP)

Ejemplo: MPI

MPI (*Message Passing Interface*)

- Especificación de una librería para paso de mensaje, aceptada como estándar *de facto* por la industria y usuarios
  - Reemplazó a una librería previa similar (PVM: *Parallel Virtual Machine*)
- El mecanismo básico para la coordinación entre los nodos del cluster es el intercambio de mensajes
- Ofrece una colección de primitivas que ocultan los detalles (hardware y software) de bajo nivel
- Unidad básica: procesos independientes con espacios de memoria propios y un identificador único en todo el cluster HPC
- Intercambio de datos y sincronización entre procesos mediante el paso de mensajes
- Programación en Fortran, C, C++, Python, Java y otros
- Implementaciones
  - OpenMPI (`http://www.open-mpi.org/`)
  - MPICH (`http://www.mpich.org/`)

Ejemplo: Entornos Map-Reduce (I)

Framework para dar soporte a computación paralela sobre grandes colecciones de datos en clusters HPC o en redes de computadores convencionales.

- Inicialmente propuesto por Google, popularizado por la implementación libre Apache Hadoop
- Organiza el procesamiento distribuido de los datos en dos operaciones primitivas básicas derivadas de la programación funcional
  - Paralelismo de *"grano grueso"*

Función **map**

Recibe una colección de pares atributo-valor sobre un dominio dado y genera una lista de pares atributo-valor en otro dominio distinto

$$map(k_1, v_1) \rightarrow list < k_2, v_2 >$$

Función **reduce**

Recibe una colección de claves a las que se vincula una lista de valores y devuelve una lista de valores

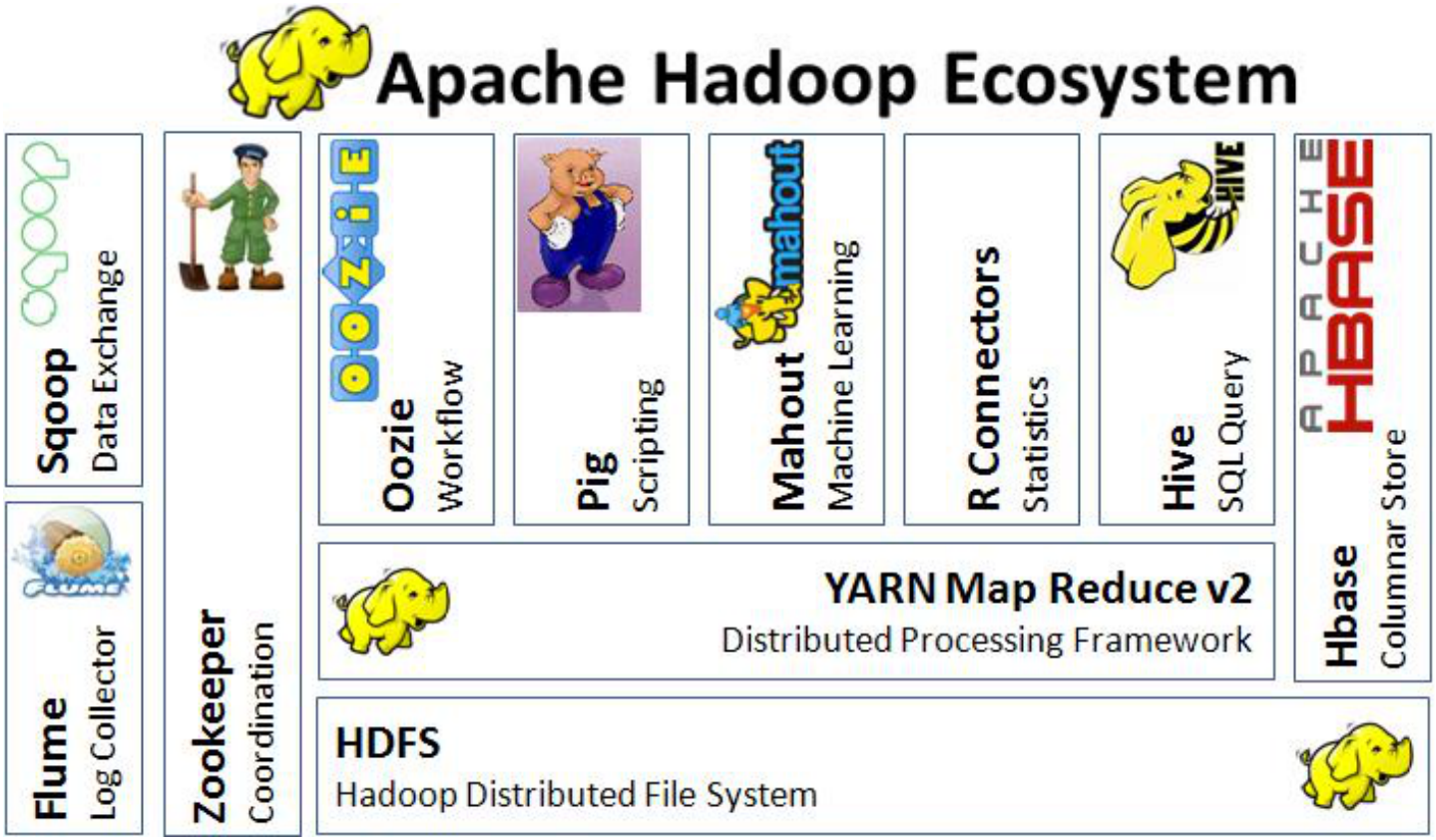
$$reduce(k_2, list < v_2 >) \rightarrow list < v_3 >$$

Ejemplo: Entornos Map-Reduce (II)

- Existe paralelismo implícito: todas las operaciones *map* son independientes entre sí, al igual que las operaciones *reduce* entre sí
- El framework *map-reduce* es resposable de la partición de los datos de entrada (usualmente en bloques de hasta 64 MB) y de su distribución sobre una colección de procesos *map* en ejecución concurrente sobre los nodos del cluster
- Una vez finalizado todos los procesos *map*, sus resultados se agregan y combinan y son de nuevo repartidos sobre los procesos *reduce* que también se ejecutan de forma concurrente.
- Las aplicaciones típicas *map-reduce* consisten en secuencias de operaciones *map* y operaciones *reduce* que sucesivamente procesan los datos recibidos de forma paralela
- Es habitual que los datos procesados por los entornos *map-reduce* se almacenen en sistemas de ficheros distribuidos de alta capacidad sobre los nodos del cluster (GFS [*Google FileSystem*], HDFS [*Haddop FileSystem*]
- Los framework *map-reduce* son tolerantes a fallos: cuentan con réplicas de los bloques de datos y de los procesos *map* y *reduce* invocados por lo que soportan el fallos de múltiples nodos de procesamiento/almacenamiento

Arquitectura de los cluster Hadoop

Hadoop: `http://hadoop.apache.org/`



Proyectos derivados

- HIVE (Plataforma Datawarehouse) `http://hive.apache.org/`
- HBase (BD distribuida NoSQL) `http://hbase.apache.org/`
- Mahout (Aprendizaje automático distribuido) `http://mahout.apache.org/`

Componentes básicos de Hadoop (I)

HDFS. Hadoop Distributed File System

Proporciona soporte al **almacenamiento distribuido**

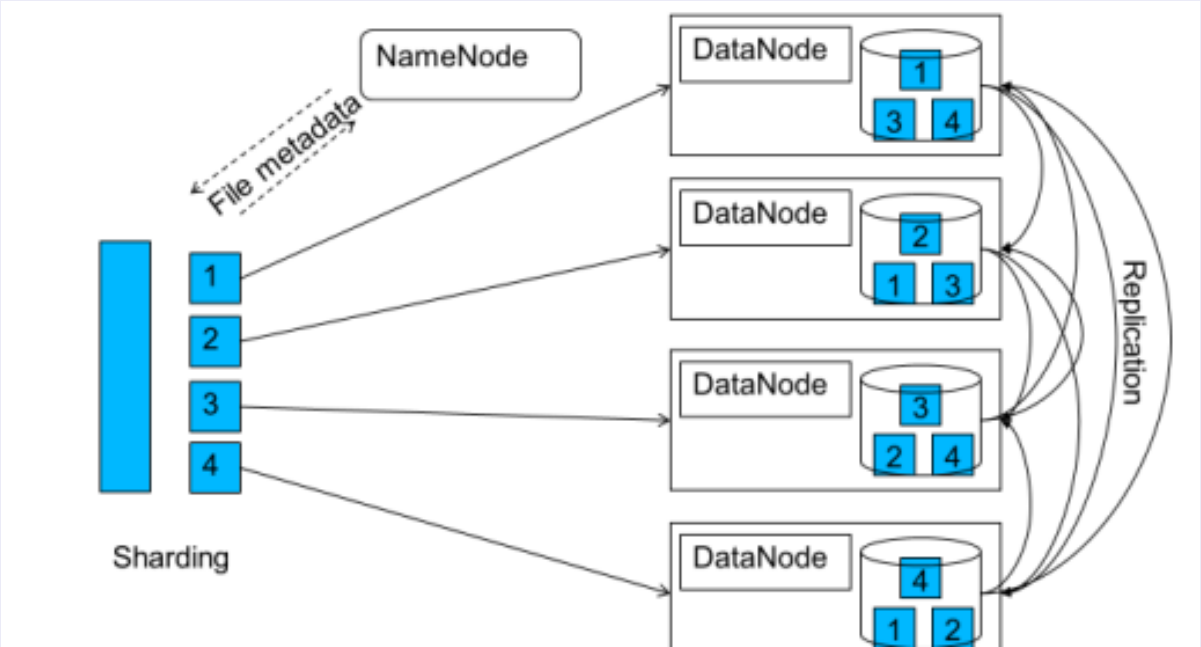
- Bloques ”grandes”, 64 MB
- Multiples replicas en distintos nodos del cluster

Name node (uno por cluster)

- Almacena y gestiona los metadatos de los ficheros y de la distribución de bloques
- Información replicada en el *Secondary Name Node*

Data node (en cada nodo del cluster)

- Responsable del almacenamiento en cada nodo del cluster de los bloques asignados



Componentes básicos de Hadoop (II)

YARN. Yet Another Resource Negotiator

Proporciona soporte para la **computación distribuida**
Componentes:

- **Resource manager** (uno por cluster)
- **Node manager** (uno por nodo del cluster)
- **Aplication master** (uno por cada aplicación ejecutada)
- **Container** (uno por cada tarea [map o reduce] en cada nodo)