

Sistemas Operativos Distribuidos

4

Sistemas de ficheros distribuidos

Índice

- Introducción
- Estructura de un SFD
- Resolución de nombres
- Acceso a los datos
- Gestión de cache
- Gestión de cerrojos
- Estudio de ejemplos: NFS, AFS y Coda
- Sistemas de ficheros paralelos
 - *General Parallel File System* (GPFS)
 - *Google File System* (GFS)

Conceptos básicos

- Sistema de ficheros distribuido (SFD)
 - Sistema de ficheros para sistema distribuido
 - Gestiona distintos dispositivos en diferentes nodos ofreciendo a usuarios la misma visión que un SF centralizado
 - Permite que usuarios compartan información de forma transparente
 - Misma visión desde cualquier máquina
- Numerosos aspectos similares a SF centralizados
- Algunos aspectos específicos como por ejemplo:
 - Traducción de nombres involucra a varios nodos
 - Uso de cache afecta a múltiples nodos
 - Aspectos de tolerancia a fallos

Características deseables del SFD

- Aplicables las correspondientes al SD global:
 - Transparencia, fiabilidad, rendimiento, escalabilidad, seguridad.
- Específicamente:
 - Espacio de nombres único
 - Soporte de migración de ficheros
 - Soporte para replicación
 - Capacidad para operar en sistemas “desconectados”
 - Una red “partida” o un cliente que usa un sistema portátil
 - Soporte de heterogeneidad en hardware y S.O.
 - Integración de nuevos esquemas de almacenamiento (SAN)
 - Paralelismo en acceso a datos de un fichero

Sistemas Operativos Distribuidos

Estructura del SFD

- Cliente (nodo con aplicación) – Servidor (nodo con disco)
- ¿Cómo repartir funcionalidad de SF entre cliente y servidor?
- Arquitectura “tradicional”
 - servidor: proporciona acceso a ficheros almacenados en sus discos
 - cliente: pasarela entre aplicación y servidor
 - con más o menos funcionalidad (clientes *fat* o *thin*)
- Arquitectura “alternativa”
 - servidor: proporciona acceso a bloques de disco
 - cliente: toda la funcionalidad del SF
 - Utilizada en sistemas de ficheros para *clusters*
 - se estudiará más adelante

Arquitectura del SFD

- Solución basada en arquitectura tradicional parece sencilla:
 - SF convencional en servidor
 - Exporta servicios locales para abrir, cerrar, leer, escribir, cerrojos,...
- ¿Asunto zanjado? No todo está resuelto:
 - Resolución de nombre de fichero:
 - Cliente y varios servidores involucrados. ¿Cómo se reparten trabajo?
 - Acceso a los datos:
 - ¿se transfiere sólo lo pedido? ¿más cantidad? ¿todo el fichero?
 - Uso de cache en el cliente. Coherencia entre múltiples caches.
 - Gestión de cerrojos:
 - ¿Qué hacer si se cae un cliente en posesión de un cerrojo?
 - Otros: migración, replicación, heterogeneidad, ...

Operaciones sobre los ficheros

- Apertura del fichero: Traducción del nombre
 - Gestión de nombres
- Lecturas/escrituras sobre el fichero
 - Acceso a datos
 - Uso de cache
- Establecimiento de cerrojos sobre el fichero

Gestión de nombres

- Similar a SF convencionales:
 - Espacio de nombres jerárquico basado en directorios
 - Esquema de nombres con dos niveles:
 - Nombres de usuario (*pathname*) y Nombres internos
 - Directorio: Relaciona nombres de usuario con internos
- Nombres de usuario
 - Deben proporcionar transparencia de la posición
 - Nombre no debe incluir identificación del nodo donde está
triqui.fi.upm.es:/home/fichero.txt
- Nombres internos
 - Identificador único de fichero (UFID) utilizado por el sistema
 - Puede ser una extensión del usado en SF convencionales.
 - Por ejemplo:
id. de máquina + id. disco + id. partición + id. inodo

Sistemas Operativos Distribuidos

Espacio de nombres

- Similar a SF convencionales:
 - Espacio de nombres dividido en volúmenes (o particiones, o ...)
 - Cada volumen gestionado por un servidor
 - Espacio único mediante composición de volúmenes:
 - Extensión distribuida de operación de montaje de UNIX
- Alternativas en la composición:
 - Montar sistema de ficheros remoto sobre la jerarquía local (NFS)
 - Montaje en cliente: información de montaje se almacena en cliente
 - Espacio de nombres diferente en cada máquina
 - Único espacio de nombres en todas las máquinas (AFS)
 - Montaje en servidor: información de montaje se almacena en servidor
 - Espacio de nombres común para el SD

Sistemas Operativos Distribuidos
9

Fernando Pérez Costoya

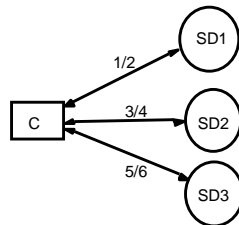
Resolución de nombres

- Traducir una ruta que se extiende por varios servidores
- ¿Quién busca cada componente de la ruta?
 - cliente: solicita contenido del directorio al servidor y busca (AFS)
 - servidor: realiza parte de la búsqueda que le concierne
- Alternativas en la resolución dirigida por los servidores
 - iterativa, transitiva y recursiva
- “Cache de nombres” en clientes
 - Almacén de relaciones entre rutas y nombres internos
 - También existe en SF convencional
 - Evita repetir proceso de resolución
 - Operación más rápida y menor consumo de red (*escalabilidad*)
 - Necesidad de coherencia
 - Fichero borrado y nombre interno reutilizado
 - Uso de contador de versión del inodo
 - id. de máquina + id. disco + id. partición + id. inodo + n° versión*

Sistemas Operativos Distribuidos
10

Fernando Pérez Costoya

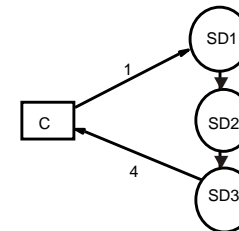
Resolución iterativa



Sistemas Operativos Distribuidos
11

Fernando Pérez Costoya

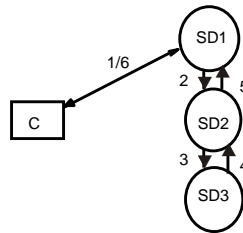
Resolución transitiva



Sistemas Operativos Distribuidos
12

Fernando Pérez Costoya

Resolución recursiva



Localización de ficheros

- Resolución obtiene nombre interno (UFID)
- Uso de UFID con dir. máquina donde fichero está localizado
 - No proporciona, en principio, independencia de la posición
 - Nombre de fichero cambia cuando éste migra
- Uso de UFID que no contenga información de máquina:
 - Por ejemplo (AFS):
 - *id. único de volumen + id. inodo + n° versión*
 - Permite migración de volúmenes
 - Requiere esquema de localización: Volumen → Máquina
- Posibles esquemas de localización:
 - Tablas que mantengan la información de ubicación (AFS)
 - Uso de *broadcast* para localizar nodo
- Uso de “cache de localizaciones” en clientes

Acceso a datos del fichero

- Una vez abierto el fichero, se tiene info. para acceder al mismo
- Aspectos de diseño vinculados con acceso a datos:
 - ¿Qué se garantiza ante accesos concurrentes?
 - Semántica de uso concurrente
 - ¿Qué información se transfiere entre cliente y servidor?
 - Modelo de acceso
 - ¿Qué info. se guarda en cache y cómo se gestiona?
 - Gestión de cache

Semánticas de uso concurrente

- Sesión: serie de accesos que realiza cliente entre *open* y *close*
- La semántica especifica el efecto de varios procesos accediendo de forma simultánea al mismo fichero
- Semántica UNIX
 - Una lectura *ve* los efectos de todas las escrituras previas
 - El efecto de dos escrituras sucesivas es el de la última
- Semántica de sesión (AFS)
 - Cambios a fichero abierto, visibles sólo en nodo que lo modificó
 - Una vez cerrado, cambios visibles sólo en sesiones posteriores
 - Múltiples imágenes simultáneas del fichero
 - Dos sesiones sobre mismo fichero que terminan concurrentemente:
 - La última deja el resultado final
 - No adecuada para procesos con acceso concurrente a un fichero

Sistemas Operativos Distribuidos

Modelo de acceso

- Modelo carga/descarga
 - Transferencias completas del fichero
 - Localmente se almacena en memoria o discos locales
 - Normalmente utiliza semántica de sesión
 - Eficiencia en las transferencias
 - Llamada *open* con mucha latencia
- Modelo de servicio remoto
 - Servidor debe proporcionar todas las operaciones sobre el fichero
 - Acceso por bloques
 - Modelo cliente/servidor

Sistemas Operativos Distribuidos
17

Fernando Pérez Costoya

Modelo carga/descarga

- Correspondencia petic. de aplicación y mens. de protocolo:
 - *open* → mensaje de descarga (*download*)
 - se realiza traducción y servidor envía fichero completo
 - cliente almacena fichero en cache local
 - *read/write/seek* → no implica mensajes de protocolo
 - lecturas y escrituras sobre copia local
 - *close* → mensaje de carga (*upload*)
 - si se ha modificado, se envía fichero completo al servidor

Sistemas Operativos Distribuidos
18

Fernando Pérez Costoya

Modelo de serv. remoto: con estado

- Correspondencia petic. de aplicación y mens. de protocolo:
 - *open* → mensaje de apertura
 - se realiza traducción
 - servidor habilita zona de memoria para info. de la sesión
 - retorna id. específico para esa sesión
 - *read/write/seek* → mensaje de protocolo correspondiente
 - mensaje incluye id. sesión
 - mensaje lectura y escritura incluye tamaño pero no posición
 - uso de información sobre la sesión almacenada en servidor (posición)
 - *close* → mensaje de cierre
 - servidor libera zona de memoria de la sesión
- Ventajas servicio con estado (frente a sin estado):
 - Mensajes más pequeños, posibilidad de realizar políticas “inteligentes” en servidor (p.ej. lectura anticipada), procesamiento de peticiones posiblemente un poco más eficiente

Sistemas Operativos Distribuidos
19

Fernando Pérez Costoya

Modelo de serv. remoto: sin estado

- Correspondencia petic. de aplicación y mens. de protocolo:
 - *open* → mensaje de apertura
 - se realiza traducción
 - servidor no habilita zona de memoria para info. de sesión (cliente sí)
 - retorna id. interno del fichero UFID
 - *read/write* → mensaje de protocolo correspondiente
 - mensaje autocontenido
 - mensaje incluye UFID, tamaño y posición
 - *seek* → no implica mensaje de protocolo
 - *close* → no implica mensaje de protocolo
 - cliente libera zona de memoria de la sesión
- Ventajas servicio sin estado (frente a con estado):
 - Tolerancia a fallos ante re arranque del servidor, posiblemente menos mensajes, no hay gastos de recursos en servidor por cada cliente (*escalabilidad*)

Sistemas Operativos Distribuidos
20

Fernando Pérez Costoya

Gestión de cache

- El empleo de cache permite mejorar el rendimiento
- Caches en múltiples niveles de un SD:
 - Cache en los servidores
 - Reducen los accesos a disco
 - Cache en los clientes
 - Reduce el tráfico por la red
 - Reduce la carga en los servidores
 - Puede situarse en discos locales (no permite nodos sin disco)
 - Más capacidad pero más lento
 - No volátil, facilita la recuperación
 - y/o en memoria principal
 - Menor capacidad pero más rápido
 - Memoria volátil

Uso de cache en clientes

- Empleo de cache de datos en clientes
 - Mejora rendimiento y capacidad de crecimiento
 - Introduce problemas de coherencia
- Otros tipos de cache
 - Cache de nombres
 - Cache de metadatos del sistema de ficheros
- Políticas de gestión de cache de datos:
 - Política de actualización
 - Política de coherencia
- Enfoque alternativo: Cache colaborativa (xFS)
 - Si bloque solicitado está en la cache de otro cliente, se copia de ésta

Política de actualización

- Escritura inmediata (*write through*)
 - Buena fiabilidad
 - Las escrituras son más lentas
 - Mayor fragmentación en la información transferida por la red
- Escritura diferida (*delayed write*)
 - Escrituras más rápidas
 - Se reduce el tráfico en la red
 - Los datos pueden borrarse antes de ser enviados al servidor
 - Menor fiabilidad
 - ¿Cuándo volcar los datos?
 - Volcado periódico
 - Volcado al cerrar (*Write-on-close*)

Coherencia de cache

- El uso de cache en clientes produce problema de coherencia
 - ¿es coherente una copia en cache con el dato en el servidor?
- Estrategia de validación iniciada por el cliente
 - cliente contacta con servidor para determinar validez
 - en cada acceso, al abrir el fichero o periódicamente
- Estrategia de validación iniciada por el servidor
 - servidor avisa a cliente (*callback*) al detectar que su copia es inválida
 - generalmente se usa *write-invalidate* (no *write-update*)
 - servidor almacena por cada cliente info. sobre qué ficheros guarda
 - implica un servicio con estado

C. de cache: semántica de sesión

- Validación iniciada por el cliente (usada en AFS versión 1):
 - En apertura se contacta con servidor enviando nº de versión (o fecha modificación) del fichero almacenado en cache local (si lo hay)
 - Servidor comprueba si corresponde con versión actual:
 - En caso contrario, se envía la nueva copia
- Validación iniciada por el servidor (usada en AFS versión 2):
 - Si hay copia en cache local, en apertura no se contacta con servidor
 - Servidor almacena información de qué clientes tienen copia local
 - Cuando cliente vuelca nueva versión del fichero al servidor:
 - servidor envía invalidaciones a clientes con copia
 - Disminuye nº de mensajes entre cliente y servidor
 - Mejor rendimiento y *escalabilidad*
 - Dificultad en la gestión de *callbacks*
 - No encajan fácilmente en modelo cliente-servidor clásico

C. de cache: semántica UNIX

- Validación iniciada por el cliente
 - Inaplicable
 - Hay que contactar con servidor en cada acceso para validar info.
- Validación iniciada por el servidor. 2 ejemplos de protocolos:
 - *Prot1*: control en la apertura con desactivación de cache
 - Basado en Sprite: SOD desarrollado en Berkeley en los 80
 - *Prot2*: uso de *tokens*
 - Basado en DFS, sistema de ficheros distribuido de DCE (*Open Group*)

C. de cache: semántica UNIX. *Prot1*

- Servidor guarda info. de qué clientes tienen abierto un fichero
- Si acceso concurrente conflictivo (1 escritor + otro(s) cliente(s))
 - se anula cache y se usa acceso remoto en nodos implicados
- En *open* cliente contacta con servidor especificando:
 - modo de acceso + nº de versión de copia en cache (si la hay)
 - Si no hay conflicto de acceso:
 - Si versión del cliente en cache es más antigua, se indica que la invalide
 - Si la petición produce un conflicto de acceso:
 - Se le envía a los clientes con el fichero abierto una orden de invalidación y desactivación de la cache para ese fichero
 - Si era un escritor se le pide un volcado previo
 - Se indica al cliente que invalide y desactive la cache para ese fichero
 - Si la petición se encuentra que ya hay conflicto
 - Se indica al cliente que invalide y desactive la cache para ese fichero

C. de cache: semántica UNIX. *Prot2*

- Para realizar operación se requiere *token* correspondiente
 - *Token* (de lectura o escritura) asociado a un rango de bytes
- Si cliente solicita operación y no está presente *token* requerido en su nodo, se solicita al servidor de ficheros
- Para una zona de un fichero, el servidor puede generar múltiples *tokens* de lectura pero sólo uno de escritura
- Si existen múltiples *tokens* de lectura y llega solicitud de escritura, servidor reclama los *tokens*
 - Cliente devuelve *token* e invalida bloques de cache afectados
 - Cuando todos devueltos, servidor manda *token* de escritura
- Si hay un *token* de escritura y llega solicitud de lectura o escritura, servidor reclama el *token*.
 - Cliente vuelca e invalida bloques de cache afectados

Sistemas Operativos Distribuidos

Servicio con estado basado en *leases*

- Semántica UNIX requiere servicio con estado
 - ¿Cómo lograr servicio con estado pero con buena recuperación?
- *Lease*: concesión con plazo de expiración
 - Necesita renovarse
 - Aplicable a otros servicios además de la coherencia
- *Prot2* ampliado: *token* tiene un plazo de expiración
 - Pasado el plazo cliente considera que *token* ya no es válido
 - Tiene que volver a solicitarlo
 - Permite tener servidor con estado pero fácil recuperación:
 - Cuando reanuncia servidor no entrega *tokens* hasta que haya pasado plazo de expiración con lo que todos los *tokens* están caducados

Sistemas Operativos Distribuidos
29

Fernando Pérez Costoya

Gestión de cerrojos

- SFD ofrecen cerrojos de lectura/escritura
 - múltiples lectores y un solo escritor
- Peticiones *lock/unlock* generan mensajes correspondientes
 - *lock*: si factible retorna OK; sino no responde
 - *unlock*: envía a OK a cliente(s) en espera
- Requiere un servicio con estado:
 - servidor almacena qué cliente(s) tienen un cerrojo de un fichero y cuáles están en espera
- Problema: cliente con cerrojo puede caerse
 - Solución habitual: uso de *leases*
 - Cliente con cerrojo debe renovarlo periódicamente

Sistemas Operativos Distribuidos
30

Fernando Pérez Costoya

Network File System (NFS) de Sun

- Especificación de un protocolo para acceso a ficheros remotos
- Estándar diseñado para entornos heterogéneos
 - Versión 3: RFC-1813 (descrita en esta presentación)
 - Versión 4: RFC-3010 (última versión; cambios en arquitectura)
- Independencia gracias al uso de RPC/XDR de ONC
- Seguridad basada en “RPC segura”
- Compartición: máquina monta directorio remoto en SF local
 - Espacio de nombres es diferente en cada máquina
- No da soporte a migración ni replicación
- Comprende dos protocolos: montaje y acceso a ficheros

Sistemas Operativos Distribuidos
31

Fernando Pérez Costoya

Protocolo de montaje

- Establece una conexión lógica entre el servidor y el cliente
- Cada máquina incluye una “lista de exportación”
 - qué “árboles” exporta y quién puede montarlos
- Petición de montaje incluye máquina y directorio remotos
 - Se convierte en RPC al servidor de montaje remoto
 - Si permiso en lista, devuelve un identificador “opaco” (*handle*)
 - Cliente no conoce su estructura interna
- La operación de montaje sólo afecta al cliente no al servidor
 - se permiten montajes NFS anidados, pero no “transitivos”
- Aspectos proporcionados por algunas implementaciones:
 - montajes *hard* o *soft*: en montaje, si servidor no responde...
 - espera ilimitada (*hard*) o plazo máximo de espera (*soft*)
 - automontaje: no solicita montaje hasta acceso a ficheros

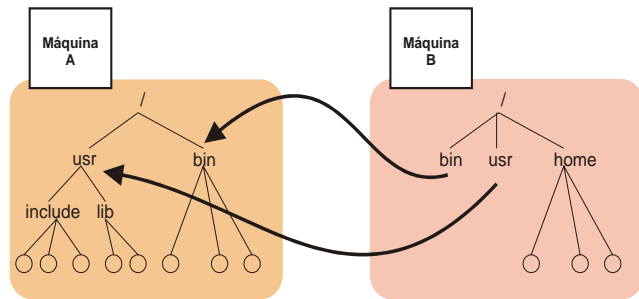
Sistemas Operativos Distribuidos
32

Fernando Pérez Costoya

Sistemas Operativos Distribuidos

Ejemplo de montaje en NFS

- La máquina A exporta `/usr` y `/bin`
- En la máquina B:
 - `mount máquinaA:/usr /usr`

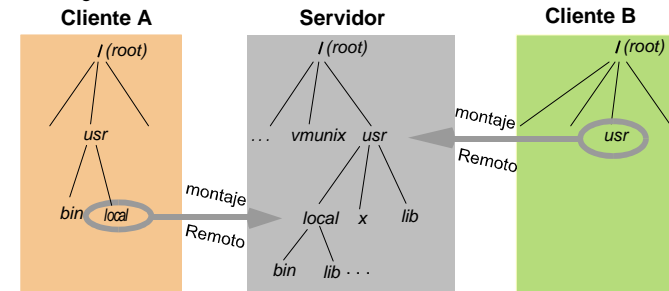


Sistemas Operativos Distribuidos
33

Fernando Pérez Costoya

Ejemplo de montaje en NFS

- Imagen diferente del sistema de ficheros



Sistemas Operativos Distribuidos
34

Fernando Pérez Costoya

Protocolo NFS

- Ofrece RPCs para realizar operaciones sobre ficheros remotos
 - Búsqueda de un fichero en un directorio (LOOKUP)
 - Lectura de entradas de directorio
 - Manipulación de enlaces y directorios
 - Acceso a los atributos de un fichero
 - Lectura y escritura. En vers. 2 escritura síncrona en disco de servidor
 - Versión 3 permite asíncrona (COMMIT fuerza escritura en disco)
- Servidores NFS sin estado (no en versión 4)
 - Operaciones autocontenidas
- OPEN reemplazado por LOOKUP (no hay CLOSE)
 - traducción iterativa componente a componente
 - LOOKUP(*handle* de directorio, fichero) → *handle* de fichero
- El protocolo no ofrece mecanismos de control de concurrencia
 - Procolo independiente de NFS: *Network Lock Manager*

Sistemas Operativos Distribuidos
35

Fernando Pérez Costoya

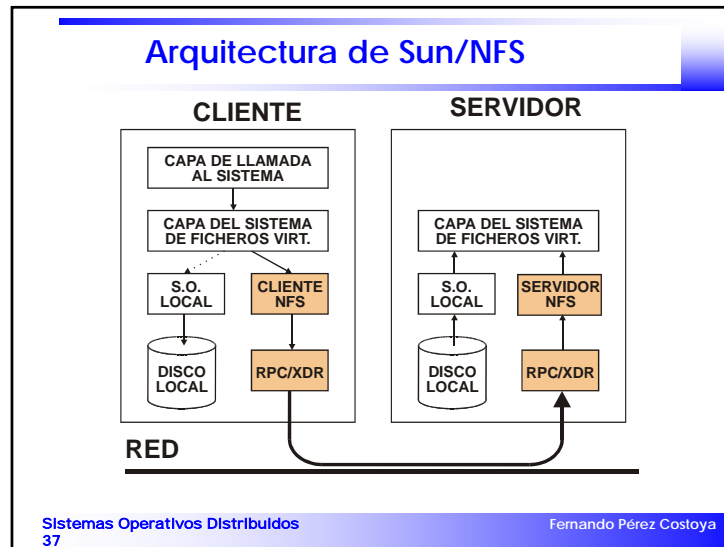
Implementación Sun/NFS

- Arquitectura basada en sistema de ficheros virtual (VFS)
- *Vnode* apunta a un *inode* local o a uno remoto (*Rnode*)
- Cada *Rnode* contiene *handle* del fichero remoto
- Contenido del *handle* depende de sistema remoto
- En sistemas UNIX se usa un *handle* con tres campos:
 - id. del sistema de ficheros
 - número de *inode*
 - número del versión del *inode* (se incrementa en cada reutilización)
- En montaje se obtiene *handle* de la raíz del subárbol montado
- Posteriores operaciones *lookup* obtienen sucesivos *handles*

Sistemas Operativos Distribuidos
36

Fernando Pérez Costoya

Sistemas Operativos Distribuidos



Acceso a ficheros en Sun/NFS

- Las transferencias se realizan en bloques de 8 KB
- Los bloques se almacenan en la cache de los clientes
- Los clientes realizan lecturas adelantadas de un bloque
- Las escrituras se realizan localmente.
- Los bloques se envían al servidor cuando se completan o se cierra el fichero
- Cache del servidor:
 - escritura síncrona o asíncrona según lo indicado por el cliente
- 3 tipos de cache en el cliente:
 - cache de nombres para acelerar las traducciones
 - cache de atributos de ficheros y directorios (fechas, dueño, ...)
 - cache de bloques de ficheros y directorios

Sistemas Operativos Distribuidos 38 Fernando Pérez Costoya

Coherencia de cache en Sun/NFS

- No asegura ninguna semántica
- Validación dirigida por el cliente:
 - Toda operación sobre un fichero devuelve sus atributos
 - Si los atributos indican que el fichero se ha modificado
 - se invalidan los datos del fichero en cache de bloques
 - Entradas de cache de bloques y atributos tienen un tiempo de vida
 - Si no se acceden es ese periodo se descartan
 - Valores típicos:
 - 3 segundos para ficheros
 - 30 para directorios

Sistemas Operativos Distribuidos 39 Fernando Pérez Costoya

Novedades de la versión 4 de NFS

- Servicio con estado basado en *leases* (hay OPEN y CLOSE)
- Diseñado para ser usado en Internet
- Integra protocolo de montaje y de cerrojos
 - Un solo puerto fijo (2049): facilita atravesar cortafuegos
- Empaquetamiento de operaciones:
 - Una llamada RPC (COMPOUND) con múltiples operaciones
- Operación LOOKUP puede resolver el camino completo
 - Montajes en el servidor visibles por el cliente
- Uso de listas de control de acceso (ACL)
- Atributos del fichero incluye un mecanismo de extensibilidad

Sistemas Operativos Distribuidos 40 Fernando Pérez Costoya

Sistemas Operativos Distribuidos

Andrew File System (AFS)

- SFD desarrollado en Carnegie Mellon (desde 1983)
 - Se presenta la versión AFS-2
- Actualmente producto de Transarc (incluida en IBM)
 - OpenAFS: versión de libre distribución para UNIX y Windows
- Sistemas distribuidos a gran escala (5000- 10000 nodos)
- Distingue entre nodos cliente y servidores dedicados
 - Los nodos cliente tienen que tener disco
- Ofrece a clientes dos espacios de nombres:
 - local y compartido (directorio /afs)
 - espacio local sólo para ficheros temporales o de arranque
- Servidores gestionan el espacio compartido
- Visión única en todos los clientes del espacio compartido

Sistemas Operativos Distribuidos
41

Fernando Pérez Costoya

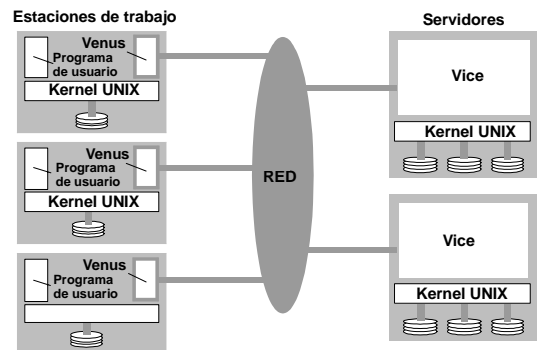
Estructura de AFS

- Dos componentes que ejecutan como procesos de usuario
- *Venus*:
 - ejecuta en los clientes
 - SO le redirecciona peticiones sobre ficheros compartidos
 - realiza las traducciones de nombres de fichero
 - resolución dirigida por el cliente
 - cliente lee directorios: requiere formato homogéneo en el sistema
- *Vice*:
 - ejecuta en los servidores
 - procesa solicitudes remotas de clientes
- Usan sistema de ficheros UNIX como almacén de bajo nivel

Sistemas Operativos Distribuidos
42

Fernando Pérez Costoya

Estructura de AFS



Sistemas Operativos Distribuidos
43

Fernando Pérez Costoya

Espacio de nombres compartido

- Los ficheros se agrupan en *volúmenes*
 - Unidad más pequeña que un sistema de ficheros UNIX
- Cada fichero tiene identificador único (UFID: 96 bits)
 - Número de volumen
 - Número de *vnodo* (dentro del volumen)
 - Número único: permite reutilizar números de vnodo
- Los UFID son transparentes de la posición
 - un volumen pueden cambiar de un servidor a otro.
- Soporte a la migración de volúmenes
- Estrategia de localización
 - número de volumen → servidor que lo gestiona
 - tabla replicada en cada servidor
 - cliente mantiene una cache de localización
 - si falla repite proceso de localización

Sistemas Operativos Distribuidos
44

Fernando Pérez Costoya

Acceso a ficheros

- Modelo de carga/descarga
 - En *open* servidor transfiere fichero completo al cliente
 - Versión actual: fragmentos de 64Kbytes
- *Venus* almacena el fichero en la cache local de los clientes
 - Se utiliza el disco local (la cache es no volátil)
- Lecturas/escrituras localmente sin intervenir *Venus*
 - Cache de UNIX opera aunque de manera transparente a AFS
- Cuando un proceso cierra un fichero (*close*)
 - Si se ha modificado se envía al servidor (*write-on-close*)
 - Se mantiene en cache local para futuras sesiones
- Modificaciones de directorios y atributos directamente al servidor

Coherencia de cache (1/2)

- Semántica de sesión
- Validación iniciada por servidor basada en *callbacks*
- Cuando cliente abre fichero del que no tiene copia local (o no es válida), contacta con el servidor
 - el servidor “anota” que el fichero tiene un *callback* para ese cliente
- Sigüientes aperturas del fichero no contactan con servidor
- Cuando cliente cierra un fichero que ha modificado:
 - Lo notifica y lo vuelca al servidor
 - Servidor avisa a los nodos con copia local para que la invaliden:
 - Eevoca el *callback*
 - Solicitud en paralelo usando una multiRPC

Coherencia de cache (2/2)

- Cuando llega una revocación a un nodo:
 - procesos con fichero abierto continúan accediendo a copia anterior
 - nueva apertura cargará el nuevo contenido desde el servidor
- Los clientes de AFS asumen que los datos en su cache son válidos mientras no se notifique lo contrario
- El servidor almacena por cada fichero una lista de clientes que tienen copia del fichero en su cache:
 - la lista incluye a todos los clientes que tienen copia y no sólo a los que tienen abierto el fichero

Coda

- Descendiente de AFS orientado a proporcionar alta disponibilidad mediante replicación de volúmenes
- Lectura de cualquier copia – Escritura en todas
- Si red “partida”: cliente sólo actualiza copias accesibles
 - Se mantienen contadores de versión en cada copia de fichero
- En reconexión: se comparan contadores de las copias
 - Si no conflicto → se propaga a todas las copias la última versión
 - Si conflicto → reconciliación automática o manual
 - Ejemplo de automática: reconciliación de directorio
- Permite operación desconectada del cliente
 - Usuario puede sugerir qué ficheros deberían estar en cache
 - En reconexión: conciliación entre cache del cliente y servidores
 - Aunque no concebido para ello, es aplicable a computación móvil

Sistemas Operativos Distribuidos

Sistemas de ficheros paralelos

Fernando Pérez Costoya

Índice

- Necesidad de E/S paralela
- Conexión de dispositivos
- Sistemas de ficheros distribuidos versus paralelos
- Técnicas de optimización de E/S
- *General Parallel File System* (GPFS)
- *Google File System* (GFS)

Sistemas Operativos Distribuidos
50

Fernando Pérez Costoya

Necesidad de E/S paralela

- Ciertas aplicaciones manejan repositorios masivos de datos
 - Requieren nº enorme de ops. E/S sobre disp. de almacenamiento
- Crecimiento muy significativo de capacidad discos
 - Pero no de sus prestaciones: ancho de banda y latencia
- Crisis E/S: desequilibrio entre capacidad procesamiento y E/S
 - Afecta a aplicaciones con fuerte componente de E/S
- Misma solución que para cómputo: uso de paralelismo
- Entrada/salida paralela
 - Distribución de datos entre múltiples dispositivos de almacenamiento
 - Acceso paralelo a los mismos
 - Debido alta latencia, mejor cuanto mayor sea tamaño accedido
 - Accesos paralelos pequeños mal rendimiento

Sistemas Operativos Distribuidos
51

Fernando Pérez Costoya

Conexión de dispo. almacenamiento

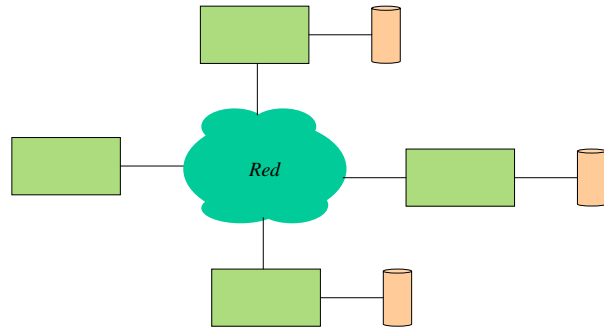
- DAS: *Direct attached storage*
 - Solución "clásica": disco asociado a nodo
- NAS: *Network attached storage*
 - Nodo que gestiona un conjunto de discos
- SAN: *Storage Area Networks*
 - Red dedicada al almacenamiento
 - Almacenamiento no vinculado a ningún nodo ("Discos de red")
 - Redes comunicación separadas para datos de aplicación y ficheros
 - Redes de almacenamiento incluyen *hubs*, *switches*, etc.
 - Tecnología más usada *Fibre Channel*
 - Conectividad total entre nodos y dispositivos:
 - Cualquier nodo accede directamente a cualquier dispositivo
 - Conectividad directa entre dispositivos
 - Copias directas entre dispositivos (agiliza *backups*, replicación, etc.)

Sistemas Operativos Distribuidos
52

Fernando Pérez Costoya

Sistemas Operativos Distribuidos

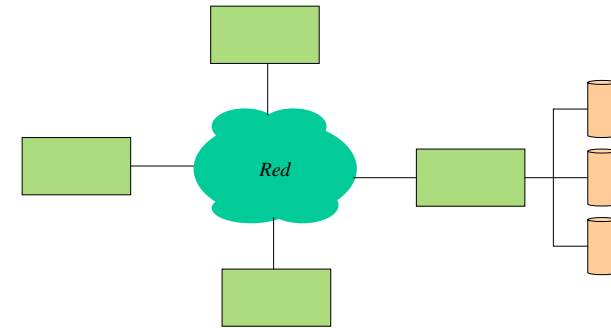
Direct-attached storage (DAS)



Sistemas Operativos Distribuidos
53

Fernando Pérez Costoya

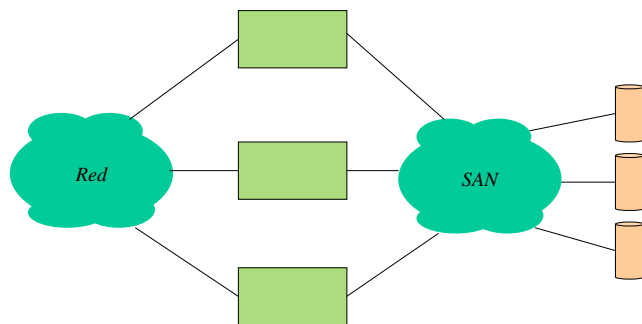
Network-attached storage (NAS)



Sistemas Operativos Distribuidos
54

Fernando Pérez Costoya

Storage Area Network (SAN)



Sistemas Operativos Distribuidos
55

Fernando Pérez Costoya

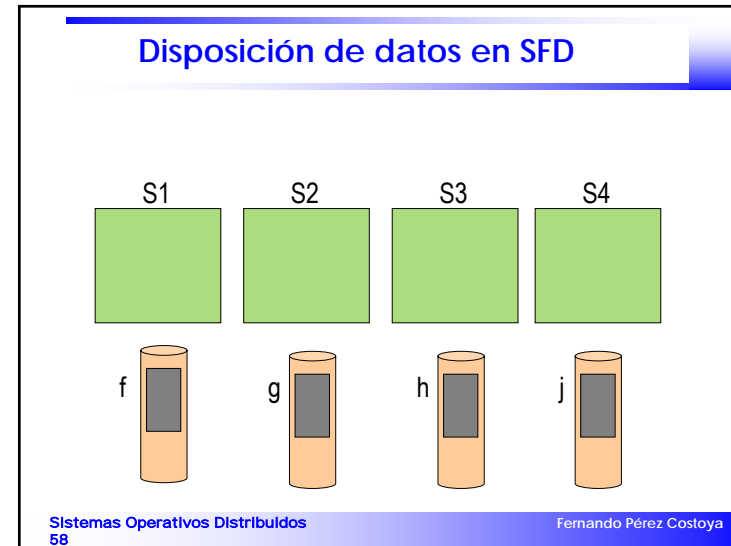
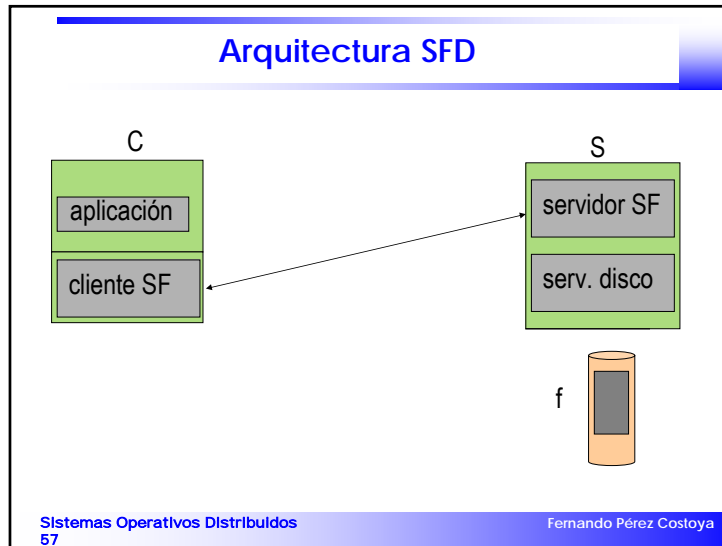
Sistemas de ficheros para E/S paralela

- ¿Por qué no son adecuados sistemas de ficheros distribuidos?
 - Como, por ejemplo, NFS o AFS
- Almacenan cada fichero en un solo servidor:
 - No hay paralelismo en accesos a fichero
 - “Cuello de botella”, falta de *escalabilidad* y punto único de fallo
- Demasiadas capas de software en el cliente y en el servidor
 - Separación de funcionalidad poco nítida (incluso redundancia)
- No aprovechan adecuadamente paralelismo de las SAN

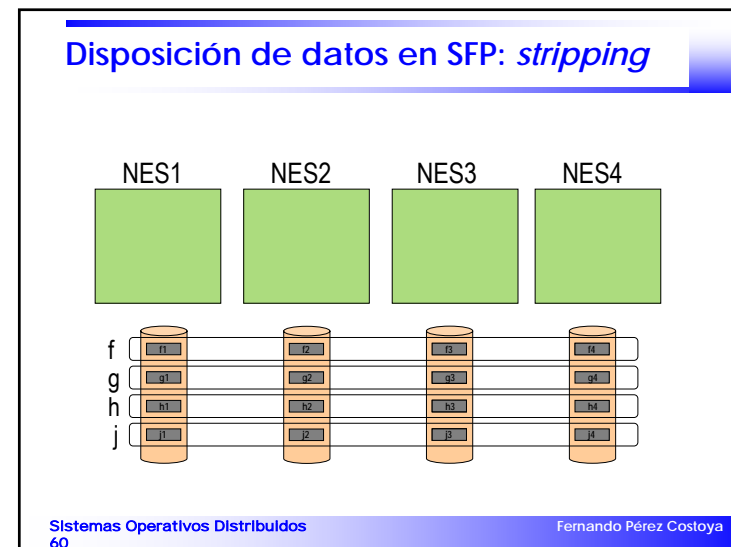
Sistemas Operativos Distribuidos
56

Fernando Pérez Costoya

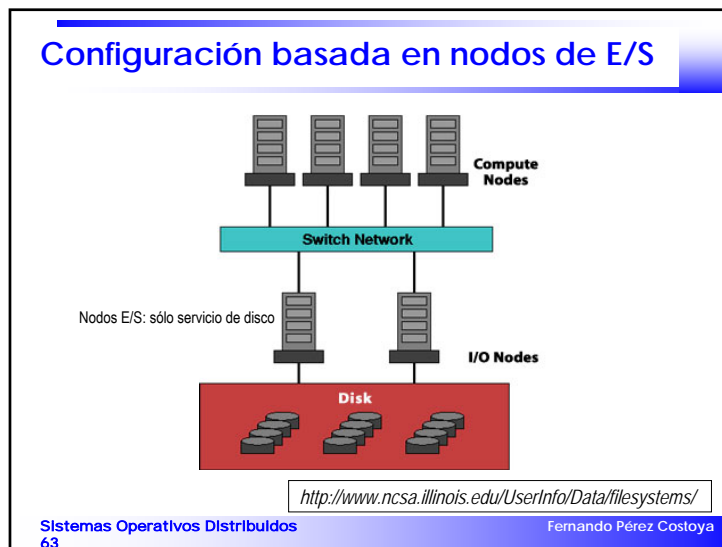
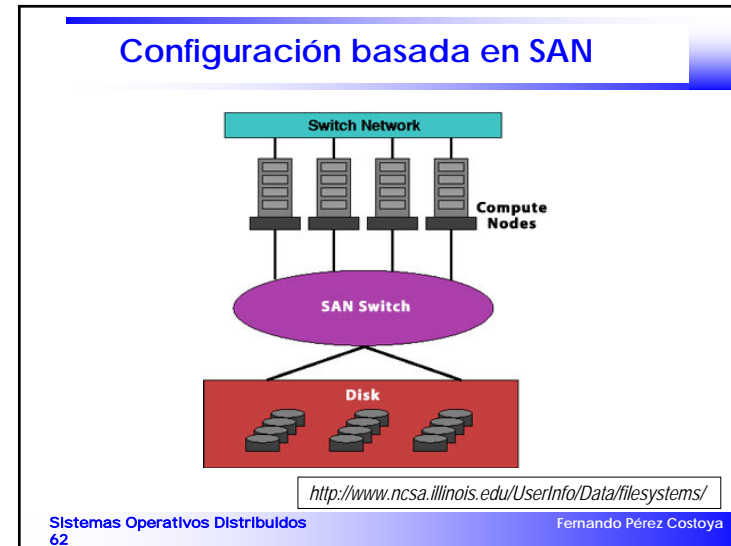
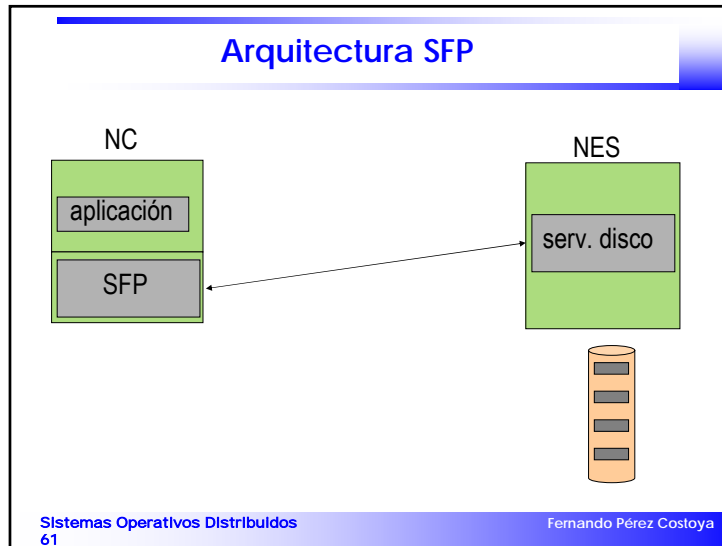
Sistemas Operativos Distribuidos



- ### Sistemas de ficheros paralelos
- Uso de *stripping*.
 - Datos de fichero distribuidos entre discos del sistema
 - Similar a RAID 0 pero por software y entre varios nodos
 - *Shared disk file systems*
 - Nuevo reparto de funcionalidad de SF en 2 niveles
 - Nivel inferior: servicio de almacenamiento distribuido
 - Proporcionado por la SAN
 - Si no SAN, módulo de servicio de disco en cada nodo E/S (NES)
 - Nivel superior: sist. ficheros en cada nodo de cómputo (NC)
 - Cada NC accede a los discos como si fueran locales
 - Cada NC gestiona la metainfo. de los datos que accede
 - Se requiere un mecanismo de cerrojos distribuido
 - Ejemplos: GPFS, PVFS, Lustre o Google File System
- Sistemas Operativos Distribuidos 59 Fernando Pérez Costoya

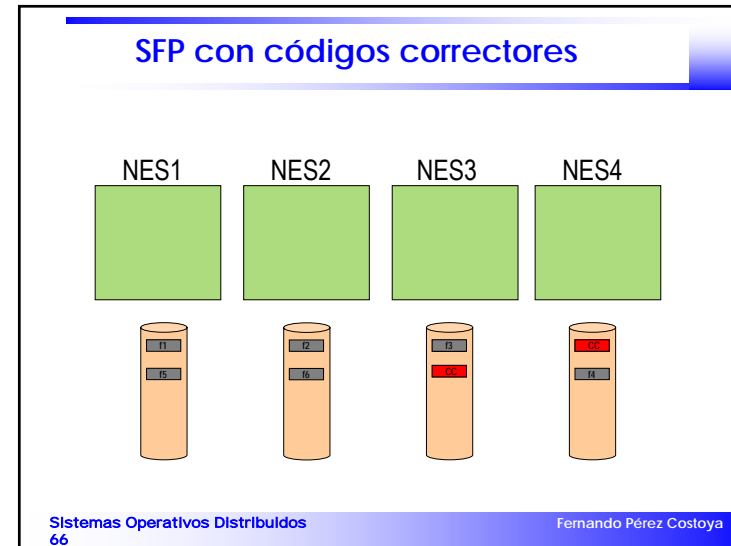
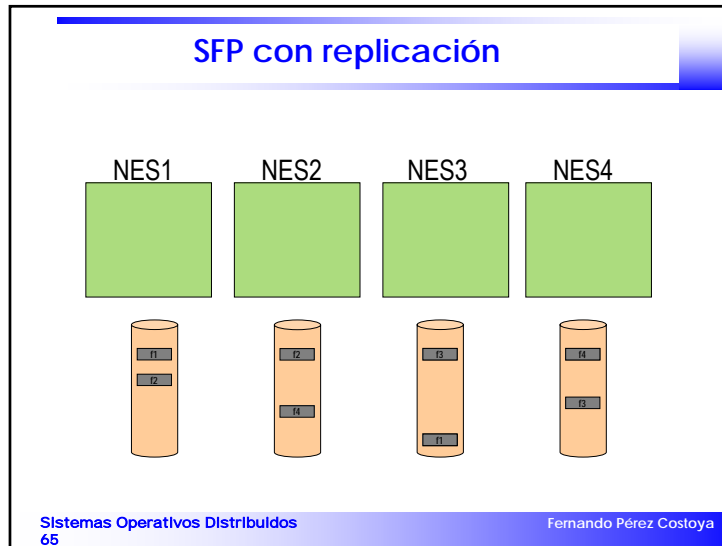


Sistemas Operativos Distribuidos



- ### Redundancia de datos
- *Striping* aumenta probabilidad de fallo al acceder a fichero
 - Necesidad de redundancia para mejorar disponibilidad
 - 2 alternativas:
 - Replicación (\approx RAID 1)
 - Códigos correctores (\approx RAID 5)
 - Comparativa:
 - Redundancia ocupa más espacio pero requiere menos cómputo
 - Códigos correctores: problema de las "escrituras pequeñas"
 - Escritura no abarca todos datos controlados por un código corrector
 - Lectura de datos a sobrescribir y código corrector previo
 - Cálculo del nuevo código corrector
 - Escritura de nuevos datos y nuevo código corrector
 - Hay que evitarlas
 - Tamaño escritura múltiplo tamaño zona controlada por código corrector
- Sistemas Operativos Distribuidos 64 Fernando Pérez Costoya

Sistemas Operativos Distribuidos



Técnicas de optimización de E/S

- Objetivo: Maximizar tamaño de accesos a dispositivo
- Optimización de:
 - Operaciones de un proceso
 - Operaciones independientes de varios procesos
 - Operaciones colectivas
- Diversas técnicas
 - Uso de caché de datos
 - Operaciones no contiguas
 - Operaciones colectivas

Sistemas Operativos Distribuidos 67 Fernando Pérez Costoya

Caché de datos

- Permite nodos clientes soliciten/vuelquen múltiples bloques
- Uso de *prefetching* en lecturas
 - Se piden por anticipado más bloques de los solicitados por lectura
 - SF detecta patrón de acceso de la aplicación:
 - P. ej. secuencial directo e inverso, con saltos regulares, ...
 - En caso de patrón de acceso irregular: aplicación puede especificarlo
- Uso de *write behind* (escritura diferida)
 - Nodo cliente vuelca simultáneamente múltiples bloques de datos
- Caché requiere algoritmo de coherencia
- Caché cooperativa:
 - Nodo cliente puede pedir datos a otro cliente en vez de a nodo E/S

Sistemas Operativos Distribuidos 68 Fernando Pérez Costoya

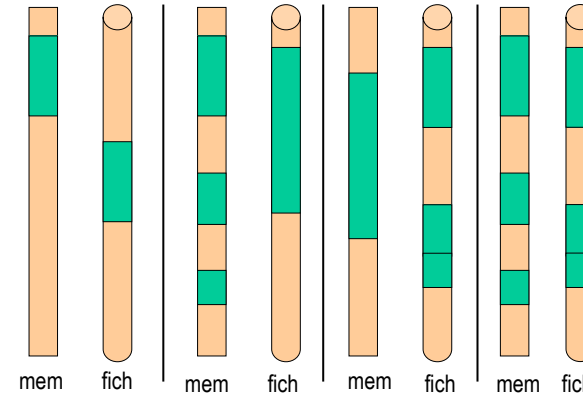
Accesos no contiguos

- Op. E/S implica dos zonas de almacenamiento
 - *Buffer* de usuario y zona del fichero afectada
 - Ambas pueden ser no contiguas
- No contigüedad más frecuente en aplicaciones paralelas
 - Ej. Proceso lee filas pares de matriz almacenada en fichero
- Si SF no da soporte a accesos no contiguos:
 - múltiples operaciones de E/S → peor rendimiento
- Soporte accesos no contiguos: *list I/O* vs. *datatypes I/O*
- *List I/O*: Op. con lista de *buffers* y zonas del fichero
- *Datatype I/O*: Definición de tipos de datos con posibles huecos
 - Para especificar no contigüedad en fichero *y/o* en *buffers*
 - Solución usada en MPI-IO

Sistemas Operativos Distribuidos
69

Fernando Pérez Costoya

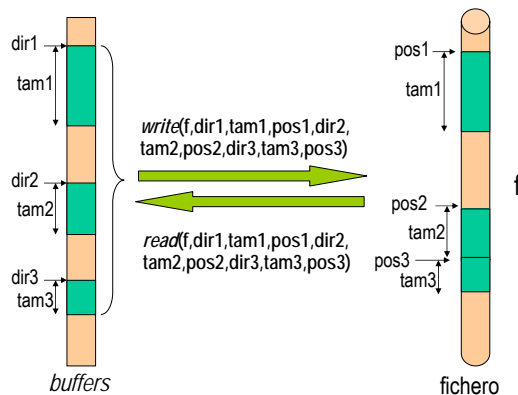
Accesos no contiguos



Sistemas Operativos Distribuidos
70

Fernando Pérez Costoya

Accesos no contiguos con *List I/O*



Sistemas Operativos Distribuidos
71

Fernando Pérez Costoya

Operaciones de E/S en sistemas UNIX

- Contiguas con posición de acceso implícita o explícita
 - *read/write* y *pread/pwrite*
- No contiguas en memoria (posición implícita o explícita)
 - *readv/writev* y *preadv/pwritev*
- No contiguas en memoria y fichero (posición explícita)
 - *lio_listio*
 - Especificadas en la extensión de tiempo real de POSIX
 - Pueden ser síncronas o asíncronas

Sistemas Operativos Distribuidos
72

Fernando Pérez Costoya

Ejemplo de uso de POSIX *List I/O*

```
#include <aio.h>...
char buf1[3], buf2[9];
int f = open("/etc/passwd", O_RDONLY);
struct aiocb * desc[2];
desc[0] = malloc(sizeof(struct aiocb));
desc[0]->aio_lio_opcode = LIO_READ;
desc[0]->aio_fildes = f;
desc[0]->aio_buf = buf1;
desc[0]->aio_nbytes = sizeof(buf1);
desc[0]->aio_offset = 7;
desc[1] = malloc(sizeof(struct aiocb));
desc[1]->aio_lio_opcode = LIO_READ;
desc[1]->aio_fildes = f;
desc[1]->aio_buf = buf2;
desc[1]->aio_nbytes = sizeof(buf2);
desc[1]->aio_offset = 22;
lio_listio(LIO_WAIT, desc, 2, NULL); ...
```

MPI-IO

- Desarrollado 1994 en laboratorio Watson de IBM
- Objetivo: dar soporte para E/S paralela en MPI
- Incorporado en MPI 2
- Integración uniforme en modelo de paso de mensajes de MPI
 - Escribir/leer en un fichero: enviar/recibir un mensaje
 - Operaciones con posición de acceso implícita o explícita
 - Soporte de varios procesos compartiendo puntero de posición
 - Soporte de operaciones asíncronas
 - Operaciones de E/S independientes y colectivas
 - Uso de tipos de datos de MPI, con posibles huecos, para
 - definir vistas (contiguas o no) sobre un fichero
 - definir *buffers* (contiguos o no) de una lectura/escritura

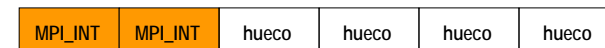
```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```

Operaciones de E/S en MPI-IO

- Lectura/escritura con posición de acceso implícita
 - Sincronas: *MPI_File_read*, *MPI_File_write*
 - Asíncronas: *MPI_File_iread*, *MPI_File_iwrite*
 - Cambio de puntero de posición de acceso: *MPI_File_seek*
- Lectura/escritura con posición de acceso explícita
 - Sincronas: *MPI_File_read_at*, *MPI_File_write_at*
 - Asíncronas: *MPI_File_iread_at*, *MPI_File_iwrite_at*
- Lectura/escritura con posición de acceso compartida
 - Sincronas: *MPI_File_read_shared*, *MPI_File_write_shared*
 - Asíncronas: *MPI_File_iread_shared*, *MPI_File_iwrite_shared*
- Operaciones colectivas
 - Posición implícita: *MPI_File_read_all*, *MPI_File_write_all*
 - Posición explícita: *MPI_File_read_at_all*, *MPI_File_write_at_all*

Ejemplo tipo no contiguo en MPI

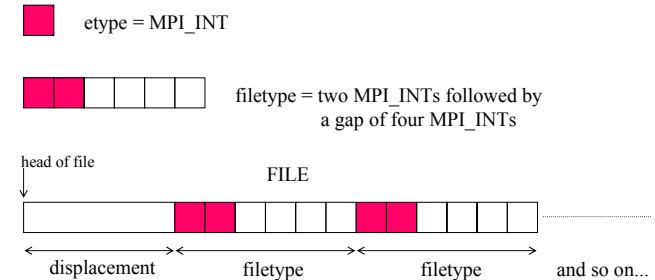
```
MPI_Aint base, extension;
MPI_Datatype tipo_cont, tipo_nocont;
MPI_Type_contiguous(2, MPI_INT, &tipo_cont);
base = 0; extension = 6 * sizeof(int);
MPI_Type_create_resized(tipo_cont, base, extension, &tipo_nocont);
MPI_Type_commit(&tipo_nocont);
```



Operaciones no contiguas en MPI-IO

- No contigüidad en *buffer*:
 - En op. lectura/escritura uso de un tipo de datos MPI que la refleje
 - Igual que con envío y recepción
- No contigüidad en fichero:
 - Proceso especifica vista del fichero que refleja su patrón de acceso
 - Para ello usa un tipo de datos MPI que defina patrón
 - Proceso sólo ve la parte del fichero que permite el patrón
 - *MPI_File_set_view(fh, disp, etype, filetype, datarep, info)*
 - Desplazamiento (*disp*)
 - Tipo de dato leído (*etype*)
 - Patrón de acceso (*filetype*)

Ejemplo de vista



Introduction to Parallel I/O and MPI-IO. Rajeev Thakur

Ejemplo de vista (código)

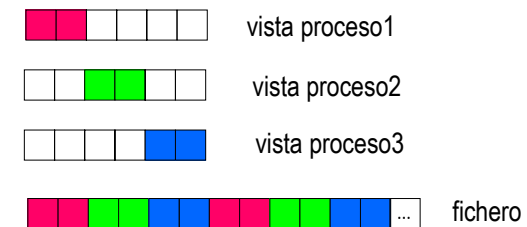
```

MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
    MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native", MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
    
```



Como en
ejemplo previo

Vistas no solapadas para varios procesos



Sistemas Operativos Distribuidos

Data Sieving

- Algunos SFP: todos los datos → *buffer* y de ahí se copian los pedidos
 - Acceso contiguo al fichero pero inútil si datos superfluos no se usan

buffers usuario copia parcial lectura completa (+ eficiente) fichero
 buffer temporal

Sistemas Operativos Distribuidos 81 Fernando Pérez Costoya

Operaciones colectivas de E/S

- Invocadas simultáneamente por todos los procesos
 - Similar a operaciones colectivas de paso de mensajes (MPI)
 - Todos los procesos especifican misma op. lectura/escritura colectiva
- SF conoce zonas de fichero y *buffers* de procesos afectados
 - Permite optimización y agrupamiento de accesos
 - Implementación con > rendimiento que usando ops. independientes
- 2 estrategias típicas: *server directed I/O* vs. *two phase I/O*
- Server directed I/O (SDIO) [Disk directed I/O]*
 - Cada NES recibe petición colectiva y realiza la parte que le afecta
 - Datos a *buffer* temporal y de ahí se distribuyen a nodos de cómputo
- Two phase I/O (TPIO)*
 - 1ª fase: Cada NC se ocupa de una parte contigua (*data sieving*)
 - 2ª fase: NCs se redistribuyen los datos
- SDIO más eficiente pero requiere NES más inteligentes

Sistemas Operativos Distribuidos 82 Fernando Pérez Costoya

SDIO (lectura)

Sistemas Operativos Distribuidos 83 Fernando Pérez Costoya

TPIO (lectura)

Sistemas Operativos Distribuidos 84 Fernando Pérez Costoya

Sistemas Operativos Distribuidos

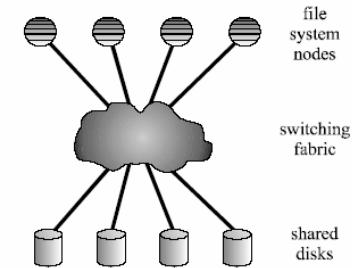
General Parallel File System de IBM

- Sistema de ficheros para clusters
 - Gran escala: decenas de miles de discos
 - Soporte grandes volúmenes, ficheros y directorios (*hashing extensible*)
 - Presente en la mayoría de los Top 500
- Soporte para SAN y nodos con discos: *Shared disk file system*
- Sistemas heterogéneos (versiones para AIX, Linux, Windows)
- Semántica POSIX (excepto *atime*)
 - Escrituras atómicas incluso aunque POSIX actual no lo requiera
- Facilidades para implementar biblioteca MPI
- Paralelismo en gestión de datos y metadatos
 - Optimiza acceso para 1 fichero/N procesos y N ficheros/1 proceso
 - Ops. administración también con paralelismo y “en caliente”
- Tolerancia a fallos en discos, nodos y comunicación

Sistemas Operativos Distribuidos
85

Fernando Pérez Costoya

Arquitectura Shared disk file system

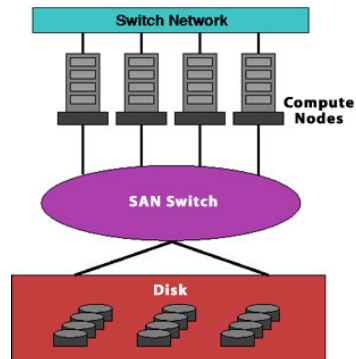


GPFS: A Shared-Disk File System for Large Computing Clusters
Frank Schmuck and Roger Haskin; *FAST '02. USENIX*

Sistemas Operativos Distribuidos
86

Fernando Pérez Costoya

Configuración basada en SAN

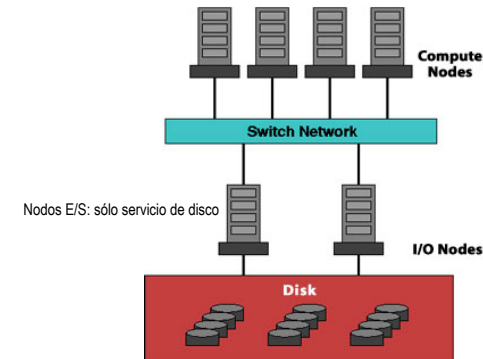


<http://www.ncsa.illinois.edu/UserInfo/Data/filesystems/>

Sistemas Operativos Distribuidos
87

Fernando Pérez Costoya

Configuración basada en nodos de E/S



Nodos E/S: sólo servicio de disco

<http://www.ncsa.illinois.edu/UserInfo/Data/filesystems/>

Sistemas Operativos Distribuidos
88

Fernando Pérez Costoya

Striping

- Bloques de fichero repartidos *round robin* en discos de un SF
- Tamaño bloque T entre 16K y 1M: típico 256K
 - Si SF formado por RAID: T múltiplo de tamaño franja de RAID
 - Ficheros pequeños y colas de ficheros: subbloques de hasta $T/32$
- Lecturas y escrituras de un nodo aprovechan paralelismo
 - Uso de *prefetching* en lecturas con detección de patrones de acceso:
 - secuencial directo e inverso y con saltos regulares
 - En caso de patrón de acceso irregular: aplicación puede especificarlo
 - Uso de *write-behind* para escrituras paralelas en discos
- Si discos de un SF no uniformes en tamaño y/o prestaciones
 - Configuración maximizando rendimiento o prestaciones
 - Reparto de bloques no uniforme

Paralelismo y control de coherencia

- SF gestiona diversos tipos de “objetos”
 - Datos de los ficheros
 - Metadatos del fichero: inodo y bloques indirectos
 - Metadatos del sistema de ficheros: información de espacio libre, etc.
- SF usa caché de “objetos” en nodo de cómputo (como en SFD)
 - Necesidad de coherencia en gestión de cachés
- Si “objeto” se extiende por varios dispositivos (no en SFD):
 - Necesidad de coherencia si se requiere actualización atómica
 - NOTA: si un SFP no usa caché ni requiere actualización atómica
 - No sería necesario esquema para mantener coherencia
- Solución basada en gestor de cerrojos distribuidos
 - NOTA: se trata de cerrojos internos del SF; No cerrojos de aplicación

Gestor de cerrojos distribuidos

- Gestor de *tokens* (GT) único en sistema ejecutando en un NC
 - Posible problema de escalabilidad y punto único de fallo
- Gestiona *tokens* lectura/escritura para distintos tipos de objetos
 - Rangos de bytes, inodos, mapas de reservas de bloques e inodos, ...
- Doble rol del *token*:
 - Control acceso paralelo a objeto + control de caché del objeto
 - Caché válida si se posee el token; volcado de cambios al perderlo
- Operación en NC requiere *token* para cierto objeto
 - Lo solicita a GT y lo mantiene para posteriores ops. en ese NC
 - Hasta que operación conflictiva en otro NC causa su revocación
- Escalabilidad GT: minimizar su intervención
 - Solicitud múltiples *tokens* en una sola petición
 - NC que requiere *token* solicita directamente revocación a NCs
 - Nuevo fichero reutiliza inodo manteniendo *tokens* asociados al mismo

Coherencia en acceso a datos

- Protocolo basado en *tokens* asociados a rangos de bytes
 - Solución similar a la usada en algunos SFD (*Prot2*)
- Optimización en la gestión de *tokens*
 1. Proceso lee/escrbe fichero usando N llamadas: 1 único *token*
 - En *Prot2*: N peticiones de *token* para rangos correspondientes
 2. M proc. escriben fich. ($1/M$ cada uno) con N llamadas/pr.: M tokens
 - En *Prot2*: $M*N$ peticiones de *token* para rangos correspondientes
- Solicitud de *token* incluye dos rangos:
 - Rango requerido: el especificado en operación *read/write*
 - Rango deseado: al que podría querer acceder en el futuro
 - Si acceso secuencial directo, hasta el infinito
- Resolución de solicitud:
 - Se revocan *tokens* que entran en conflicto con rango requerido
 - Se concede rango \subset deseado que no entre en conflicto

Optimización en gestión de *tokens*

1. En primera escritura/lectura a F, rango deseado $[0, \infty]$
 - Si ningún otro cliente accede a F, no más peticiones de *tokens*
 - Por tanto, el proceso sólo pide un *token*
2. Aplicación con M procesos crea fichero: $1/M$ parte cada uno
 - P1: $d=open(F); write(d,b,TAM_BLOQ)$
 - P1 obtiene *token* $[0, \infty]$
 - P2: $d=open(F); lseek(d,SEEK_SET,1/N); write(d,b,TAM_BLOQ)$
 - P2 obtiene *token* $[1/N, \infty]$; P1 ajusta su *token* $[0, 1/N-1]$
 - P3: $d=open(F); lseek(d,SEEK_SET,2/N); write(d,b,TAM_BLOQ)$
 - P3 obtiene *token* $[2/N, \infty]$; P2 ajusta $[1/N, 2/N-1]$; P1 *token* $[0, 1/N-1]$
 - Y así sucesivamente.
 - Por tanto, cada proceso sólo pide un *token*

Data shipping

- *Tokens* rangos de bytes: OK si accesos alineados a bloques
- Si accesos de grano fino (sub bloque):
 - Tráfico elevado por coherencia de caché
 - Sobrecarga de gestión de *tokens*
 - Volcados a disco frecuentes y de poco volumen al perder el *token*
 - Conflictos incluso aunque operaciones no solapadas (*False sharing*)
- Alternativa: *data shipping* (DS)
 - Activación DS sobre fichero F: especifica conjunto de NC
 - Bloques de F asignados *round-robin* a NC del conjunto
 - Lectura/escritura de bloque se redirige a NC asociado a ese bloque
 - Sólo ese NC lee y escribe ese bloque del disco
 - Cada NC encargado de DS vuelca periódicamente datos a discos
 - Desventajas: No uso de caché y no atomicidad en escrituras
 - Biblioteca MPI-IO usa DS (disponible para cualquier aplicación)

Coherencia acceso metadatos fichero

- Modificaciones concurrentes a metadatos de fichero
 - Directas (*chmod*)
 - Indirectas: *write* → fecha modificación, tamaño y punteros a bloques
- Uso de *token* de acceso exclusivo por inodo no es eficiente
 - Solicitud *token* de inodo por cada escritura aunque no solapadas
- Idea: actualización de inodo en paralelo y mezcla de cambios
- Solución: *token* de escritura compartida y exclusiva
 - Escrituras usan *token* de escritura compartida
 - Cada nodo modifica su copia del inodo (fecha mod., tamaño y punteros)
 - Ciertas ops. requieren *token* escritura exclusiva (*stat, utime, truncate...*)
 - Se revocan *tokens* de escritura compartida
 - Cada NC afectado vuelca su copia del inodo y se mezclan
 - ¿Quién se encarga de esta operación de mezcla?
 - Metanodo del fichero

Metanodo de un fichero

- NC elegido como metanodo (MN) de un fichero
 - Es el único que lee/escrive su inodo al disco
- Primer acceso a fichero F en un NC contacta con GT
 - Además de pedirle *tokens* de bytes de rangos y de inodo
 - Le solicita *token* de metanodo (se ofrece como MN para F)
 - Si todavía no hay MN para F, GT lo asigna a solicitante
 - Primer nodo que accede a fichero actúa como MN
 - Si ya lo hay, GT le informa de su identidad
 - NOTA: Se piden todos los *tokens* con un único mensaje a GT
- Periódicamente y cuando se revoca *token*, envía su copia a MN
 - MN mezcla cambios concurrentes a inodos:
 - $Fecha\ mod = \max(Fecha\ mod\ copias)$ (igual para tamaño y punteros)
- NC deja rol de MN para F cuando deja de usar ese inodo
 - Rechaza volcados para ese inodo → elección de nuevo MN para F

Sistemas Operativos Distribuidos

Coherencia acceso metadatos sistema

- Modificación concurrentes de información de espacio libre
 - Uso de cerrojo sobre metadatos de sistema no es eficiente
 - Cada *write* sobre fichero nuevo consulta y modifica esa información
- Solución: se divide espacio libre de un SF en R regiones
 - Cada región controlada por un cerrojo
 - Región guarda información de 1/R estado de cada disco
- Cada SF tiene un gestor de asignación (GA)
 - Mantiene estadísticas aproximadas de uso de cada región
 - NC pide región a GA y la usa para asignar espacio
 - Cuando se agota espacio, pide otra
 - GA asigna regiones intentando evitar conflictos
- Operaciones intentan evitar conflictos en uso de regiones
 - Borrado de fichero en NC: no hace directamente liberación espacio
 - La retransmite a NC que tienen asignadas las regiones afectadas

Sistemas Operativos Distribuidos
97

Fernando Pérez Costoya

Administración escalable y en línea

- Algunas operaciones de administración son “pesadas”
 - Redistribución de datos al incluir nuevos dispositivos
 - GPFS permite añadir y eliminar discos “en caliente”
 - Desfragmentación de discos
 - Comprobación/repación del SF
- Deberían aprovechar el paralelismo del sistema
 - Y hacerse en línea (si procede: no para una reparación de SF)
- Solución: Gestor del sistema de ficheros (GSF)
 - Cada SF tiene un GSF
 - Actúa de iniciador y coordinador de ops. de administración
 - Modelo maestro-trabajador
 - Va asignando a cada NC un rango de inodos que debe procesar
 - Cuando ese NC termina trabajo, le envía un nuevo rango

Sistemas Operativos Distribuidos
98

Fernando Pérez Costoya

Tolerancia a fallos en nodos

- Monitorización de estado de los nodos
- Cada nodo usa un *log* de metadatos por cada SF que usa
- Cuando se detecta caída de un nodo *N*
 - Otro NC procesa el *log*
 - No posible conflicto pues *N* tenía *tokens* requeridos
 - Fin de procesado de *log*, GT libera *tokens* que tenía *N*
- Si cae GT, se asigna función a otro nodo
 - Recupera estado preguntando a otros nodos qué *tokens* poseen
- Tratamiento similar para caída de otros gestores (p.e. GA)

Sistemas Operativos Distribuidos
99

Fernando Pérez Costoya

Tolerancia a fallos en comunicación

- Infraestructura gestiona protocolo de pertenencia a grupo
- Partición de la red:
 - Sólo grupo mayoritario puede seguir accediendo al SF
 - En caso contrario, se puede corromper SF
 - Para evitar NC fuera de grupo mayoritario tengan acceso a SF
 - Se notifica a SF que les impida acceso (se “vallen” esos nodos)

Sistemas Operativos Distribuidos
100

Fernando Pérez Costoya

Tolerancia a fallos en almacenamiento

- Distribución de datos/metadatos entre varios dispositivos
 - Fallo de un disco puede afectar gravemente a un SF
 - Especialmente si hay pérdida de metadatos
- Configuración de GPFS típica:
 - Uso de múltiples RAID con conexión dual
- Además (o en su lugar) se puede activar replicación
 - SF se encarga de gestionar dos copias
 - Tanto de datos como de metadatos
 - Mayor sobrecarga pero mejor tolerancia a fallos
 - O sólo de metadatos
 - Pérdida de algunos datos de ficheros pero no ficheros inaccesibles

Google File System

- Caldo de cultivo (≈ 2000)
- Almacenamiento datos en Google crítico: ¿usar SFP existente?
 - NO: por características de plataforma y de aplicaciones previstas
- Plataforma basada en *commodity* HW
 - Fallos HW/SW son norma y no excepción: Tolerancia a fallos SW
 - No altas prestaciones pero gran paralelismo (>1000 nodos almacen.)
 - 1 op. no es muy rápida pero se pueden hacer muchas en paralelo
- Perfil de aplicaciones previstas:
 - Modo de operación *batch* (ancho de banda importa más que latencia)
 - Millones de ficheros grandes (>100 MB) y muy grandes (>1 GB)
 - Patrones de acceso típicos
 - Escritor genera fichero completo inmutable
 - Múltiples escritores añaden datos a un fichero en paralelo
 - Con gran paralelismo, que no debe “estropear” el SF

Por la especialización hacia el éxito

- ¿SFP especializado para aplicaciones/plataforma Google?
 - Generalización de componentes clave en desarrollo informática
 - Tensión entre especialización y generalización
- Google juega con ventaja
 - Controla desarrollo de SFP y de aplicaciones
 - SFP no ideado para usarse fuera de Google
 - GFS \rightarrow NFS (*Not For Sale*)
 - Reparto de funcionalidad SFP y aplicaciones ajustable a discreción
 - Puede imponer API, modelos coherencia,... “extraños”, no estándar
- Especialización: sí pero no demasiada
 - Cobertura a mayoría de aplicaciones en Google
 - Prueba de éxito: numerosos clones de libre distribución (Hadoop FS)

Carga de trabajo prevista y API

- Perfil de aplicaciones previsto implica:
 - Mayoría lecturas grandes (>1 MB) y secuenciales
 - Algunas lecturas pequeñas aleatorias
 - Mayoría escrituras grandes (>1 MB) y secuenciales
 - Agregando datos y no sobrescribiendo (ficheros inmutables)
 - Habitual escrituras pequeñas simultáneas al final del fichero
 - Escrituras pequeñas aleatorias no previstas (pero soportadas)
- API, y modelo de coherencia, no estándar
 - Afectará a productividad pero Google manda...
 - Además de clásicas *create, open, read, write, close* y *delete*
 - *record append*
 - *snapshot*: Copia perezosa de fichero usando COW

Una primera aproximación a GFS

- Receta para diseñar de una manera simple un nuevo SFP:
 - Tomar como base un SF convencional (nodo maestro)
 - Añadir: cada trozo de fichero almacenado en nodo distinto
 - Nodo Linux convencional: trozo fichero GFS → fichero local
 - Datos repartidos en discos: problema de fiabilidad → réplicas
 - No usar caché en nodos cliente: no algoritmos de coherencia
 - Carga típica de Google ($\approx read/write\ once$) apoya esta simplificación
- ¡Único nodo maestro gestiona toda la información del SF!
 - Ha primado sencillez sobre escalabilidad y tolerancia a fallos
 - Con el tiempo lo pagaremos...
- Escalabilidad del SF: la del nodo maestro
 - Minimizar trabajo y gasto de memoria de maestro
 - Nodo maestro más potente y con más memoria
 - ¡Ambas soluciones contrarias a la filosofía Google!

Striping

- Trozos fichero repartidos entre nodos de almacenamiento (NA)
- Tamaño de trozo/*chunk/strip*: ¡64MB!
 - Respaldado por patrón típico de aplicaciones:
 - Grandes ficheros accedidos con lecturas/escrituras grandes
- Ventajas:
 - Clásicas: mejor aprovechamiento discos y red
 - Escalabilidad del maestro:
 - Menos gasto de memoria
 - Menos trabajo
- Desventajas:
 - Clásicas: relacionadas con fragmentación
 - Aunque fichero que representa *chunk* en NA puede tener tamaño justo
 - Menos paralelismo (fichero de 64MB en un único NA)
 - Pero mayoría ficheros previstos son muy grandes

El nodo maestro

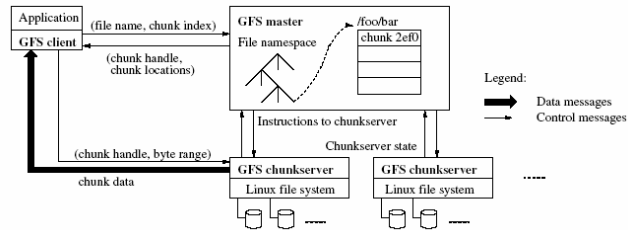
- Gestiona el SF: todo excepto accesos a trozos en NA
 - Además de información habitual de SF almacena:
 - *ChunkID* de cada trozo de un fichero
 - Datos asociados a *ChunkID*: localización actual de réplicas y nº versión
 - Por escalabilidad, SF "peculiar": en memoria y basado en prefijos
- SF en memoria
 - Escalabilidad: bueno por rendimiento; malo por gasto de memoria
 - Minimiza metainformación por fichero y por trozo (64 bits/trozo)
 - Persistencia: *log* de ops. replicado con *checkpoints* periódicos
 - *checkpoints* diseñados para poca sobrecarga y re-arranques rápidos
- SF basado en prefijos (similar a SO Sprite)
 - No inodos, ni ficheros directorio, ni enlaces, ...
 - Tabla *hash*: ruta → metadatos del fichero (usa compresión)
- *Shadow master* (maestro replicado): acceso sólo lectura a SF

Lectura

1. C: fichero+nºtrozo → M: *ChunkID*+versión+ubicación réplicas
 - C: guarda esa información hasta expiración o reapertura
 - M: puede enviar información de trozos siguientes
 2. C: Pide directamente trozo a réplica más cercana
- Minimiza intervención de maestro en operación de lectura

Sistemas Operativos Distribuidos

Arquitectura + operación de lectura



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung; SOSP '03

Sistemas Operativos Distribuidos
109

Fernando Pérez Costoya

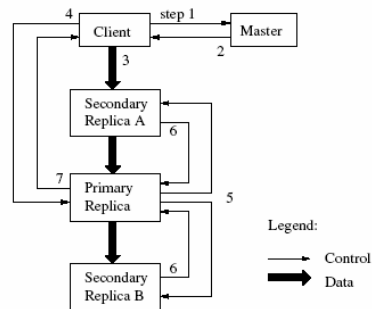
Escritura

- Debe mantener coherencia de réplicas (3 por defecto)
 - Clientes deben leer misma información de réplicas
 - Pero pueden estar mezclados datos de escrituras (no atómicas)
- Uso de *leases* para controlar qué réplica actúa como primaria
 1. (2) Igual que en lectura pero con info. propietario del *lease*
 - Si no lo hay, maestro lo selecciona en ese instante
 3. Propagación de datos a réplicas (*pipeline* de datos)
 4. Primaria: asigna orden a escritura, escribe y versión++
 5. Secundaria: escribe siguiendo orden asignado por primaria
 6. Réplicas secundarias confirman escritura a primaria
 7. Respuesta a cli.: error si falló alguna escritura → reintento
- Escrituras *multi-chunk* no atómicas

Sistemas Operativos Distribuidos
110

Fernando Pérez Costoya

Operación de escritura



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung; SOSP '03

Sistemas Operativos Distribuidos
111

Fernando Pérez Costoya

Record append

- Agrega datos (registro) sin necesidad de sincronización
- *Offset* lo elige GFS
- Limitado a tamaños $\leq \frac{1}{4}$ tamaño trozo
- Operación similar a escritura excepto en:
 - Si registro no cabe en trozo, primaria/secundarias rellenan resto
 - Informan a cliente de que reintente en siguiente trozo
 - Si cabe y no errores de escritura, similar a escritura
 - Pero *offset* lo selecciona el primario
 - Si error escritura en alguna réplica, cliente repite operación hasta OK
 - Pero en cada reintento datos se escriben a continuación
 - Contenido de réplicas distinto (escrituras erróneas, regs. duplicados)
 - Sólo asegura que se ha escrito bien al menos una vez en cada réplica
- ¡Aplicación maneja huecos, registros erróneos y duplicados!

Sistemas Operativos Distribuidos
112

Fernando Pérez Costoya

Gestión de réplicas

- Maestro no mantiene info. persistente de ubicación de réplicas
 - En arranque interroga a NA
- Maestro responsable de crear réplicas:
 - Cuando se crea el trozo, política de ubicación guiada por
 - Uso de discos y NA, y aspectos geográficos (rack-aware)
 - Cuando nº réplicas debajo de umbral
 - Por nodos caídos: *Heartbeats* entre maestro y NA con info. de estado
 - Errores de disco: uso de *checksums* en NA
 - Para equilibrar la carga entre NA
- Liberación de réplicas mediante *garbage collection*
 - En *Heartbeats* maestro detecta réplicas huérfanas y obsoletas

Finalmente se nos ha quedado pequeño

- Charla esclarecedora de Quinlan (Google) y McKusick (BSD)
 - <http://queue.acm.org/detail.cfm?id=1594206>
- Evolución de las necesidades
 1. Almacenamiento de centenares TB a decenas de PB
 2. Aumento proporcional de número de clientes
 3. Nuevas aplicaciones que manejan ficheros pequeños
 4. Nuevas aplicaciones donde latencia es importante
- Problemas (relación directa/indirecta con maestro único)
 1. Más metadatos en maestro: requiere más proceso y memoria
 2. Maestro recibe más operaciones (*open, close, ...*)
 3. Tamaño bloque (TB) menor (¿1MB?): más metadatos en maestro
 4. GFS usa TB grande y agrupa todo tipo de ops. siempre que puede
 - Además, tiempo de recuperación fallo maestro del orden de minutos

GFS II/Colossus

- GFS entra en la era de los “múltiples maestros”
 - *Sharding* de metadatos entre maestros
- GFS II/Colossus: reescritura completa
- Todavía poca información: se han filtrado aspectos adicionales
 - Tamaño de bloque 1MB
 - Tiempo de recuperación de pocos segundos
 - Uso de códigos correctores vs. replicación
 - Más especialización: soporte de Google Caffeine
 - Diseñado para trabajar conjuntamente con Bigtable
 - Almacenamiento de metadatos de Colossus en Bigtable