

Sistemas Distribuidos

Sistemas de ficheros paralelos

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)

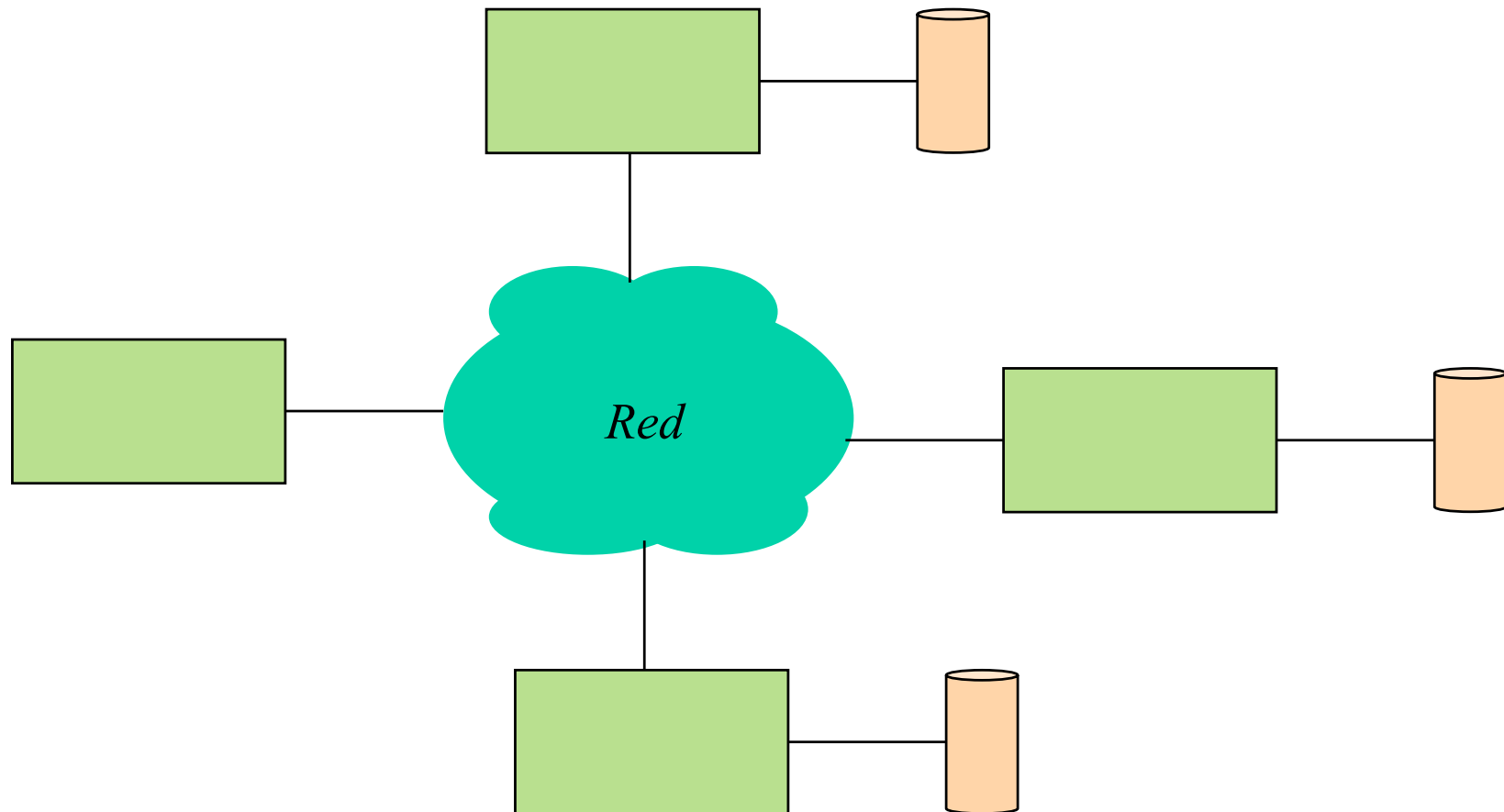
Necesidad de E/S paralela

- Ciertas aplicaciones manejan repositorios masivos de datos
 - Requieren n° 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

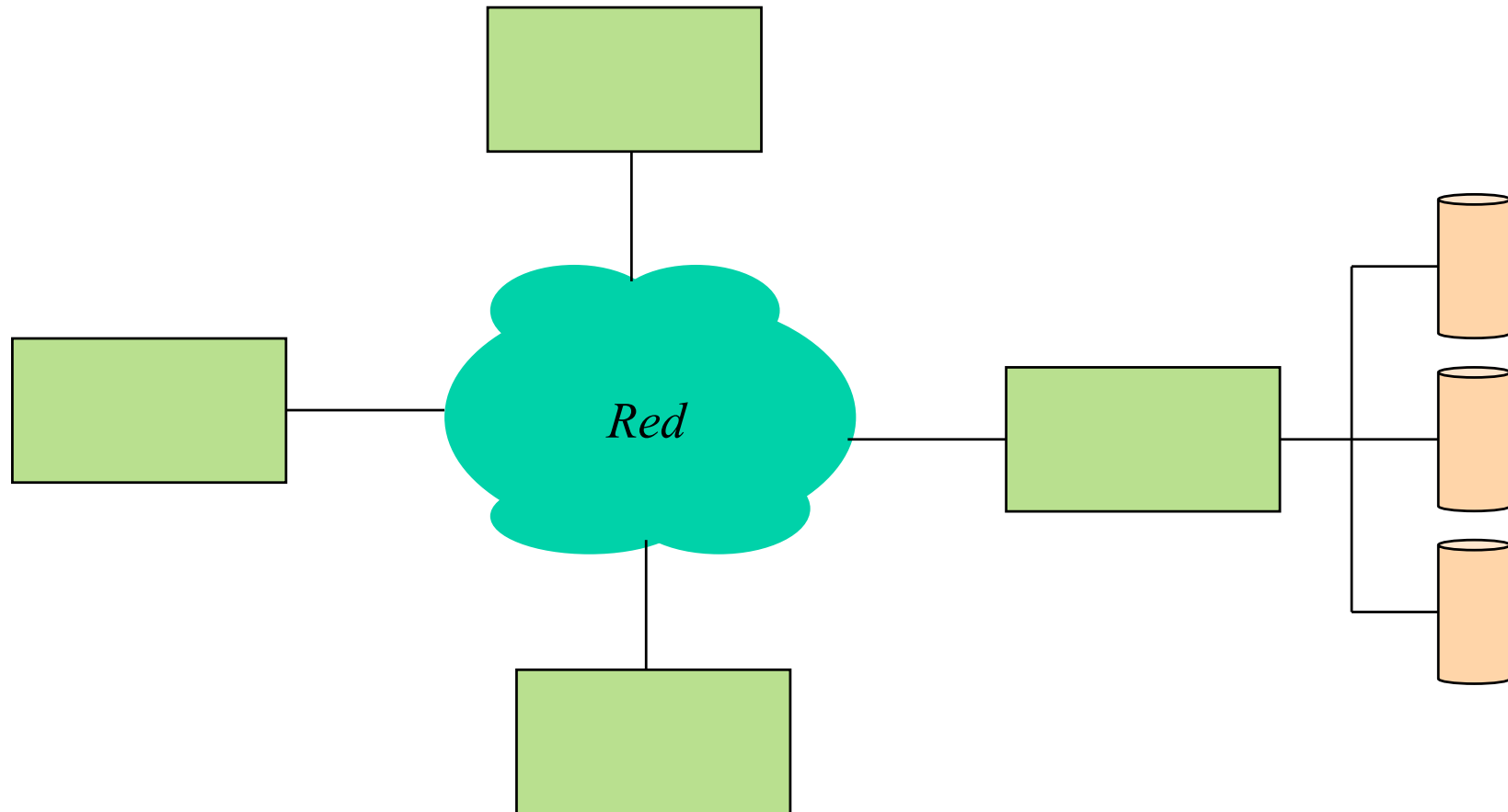
Conexión de dispos. almacenamiento

- **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.)

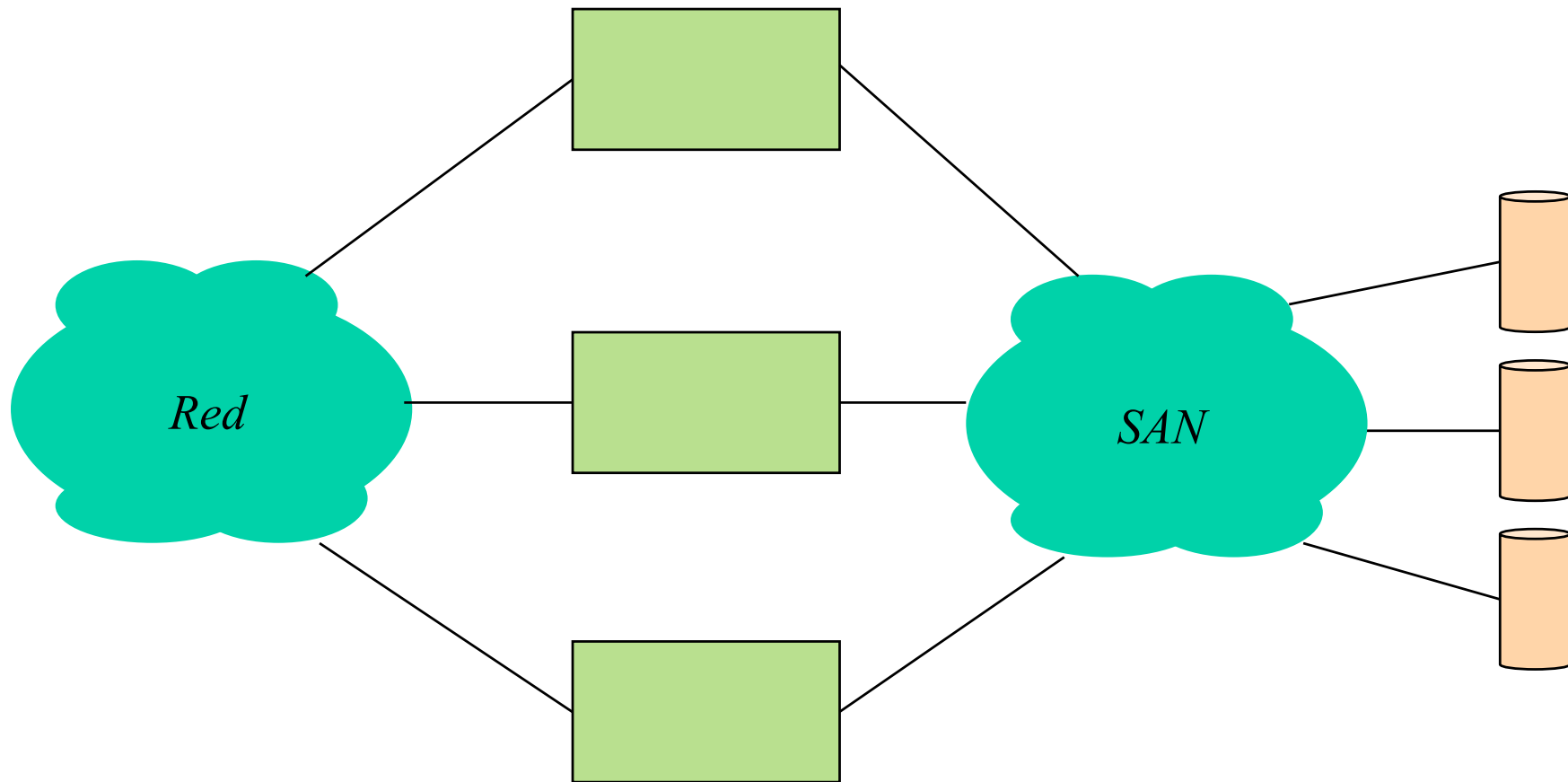
Direct-attached storage (DAS)



Network-attached storage (NAS)



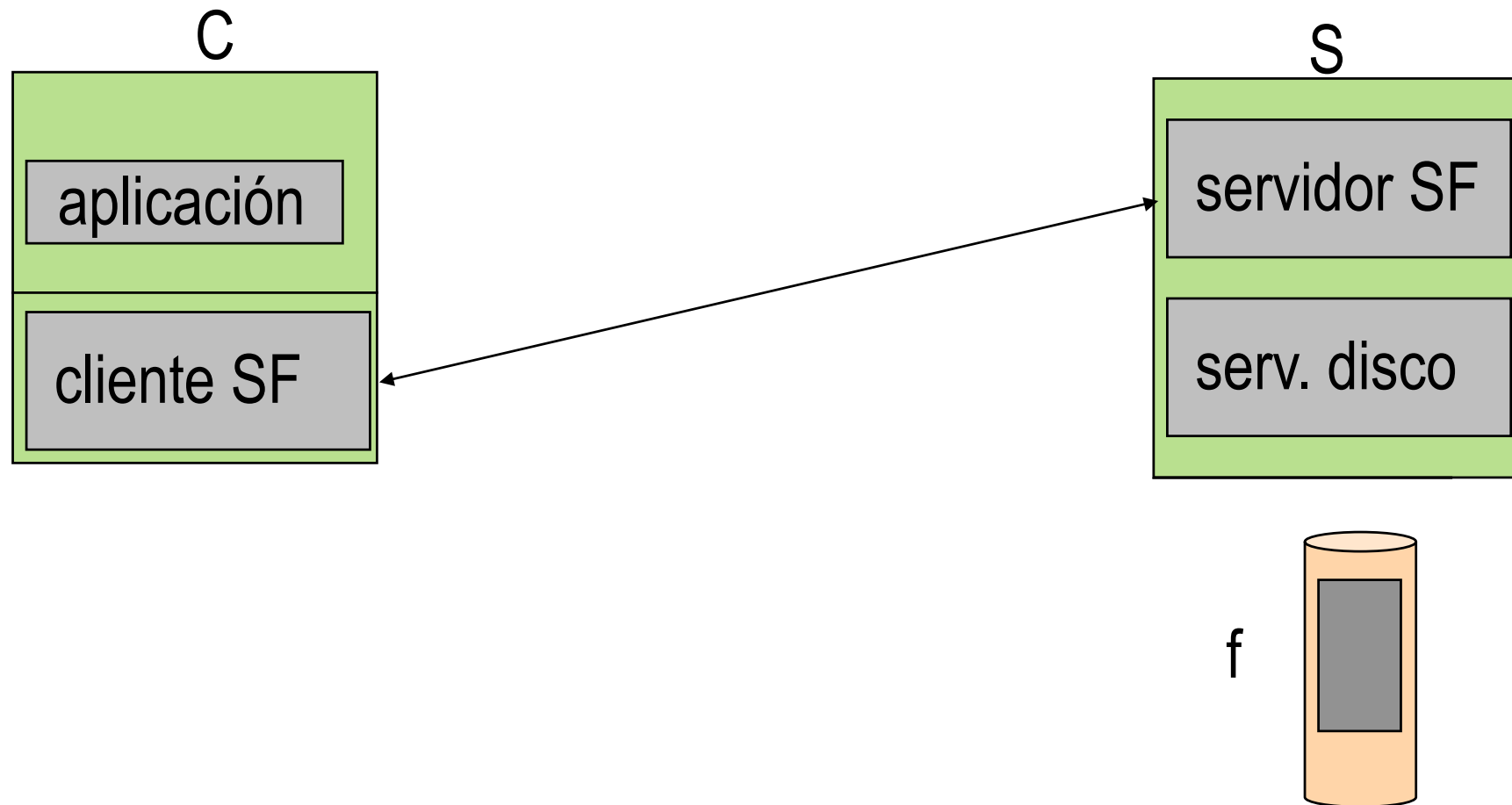
Storage Area Network (SAN)



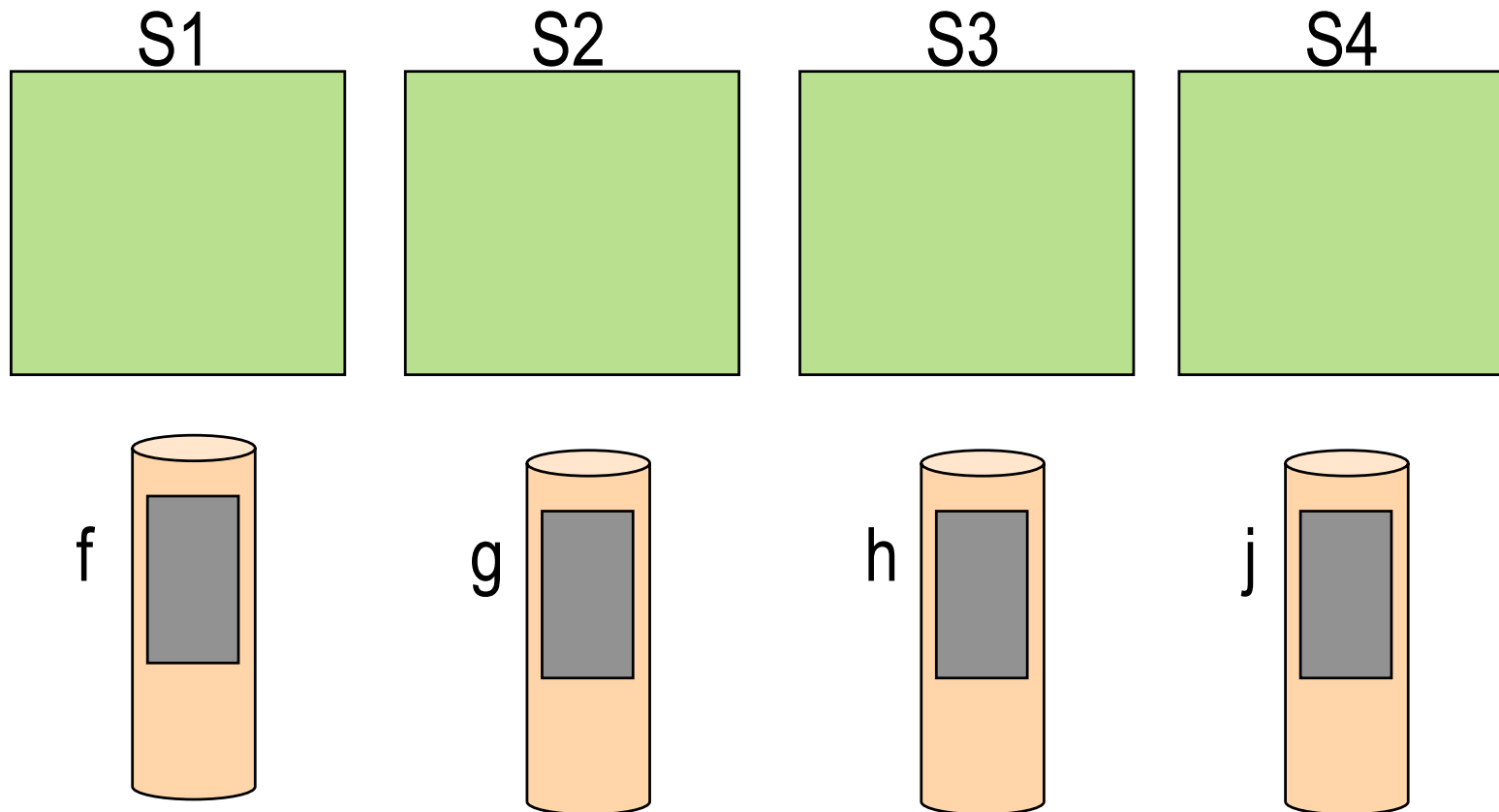
Sistemas de ficheros para E/S paralela

- ¿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

Arquitectura SFD



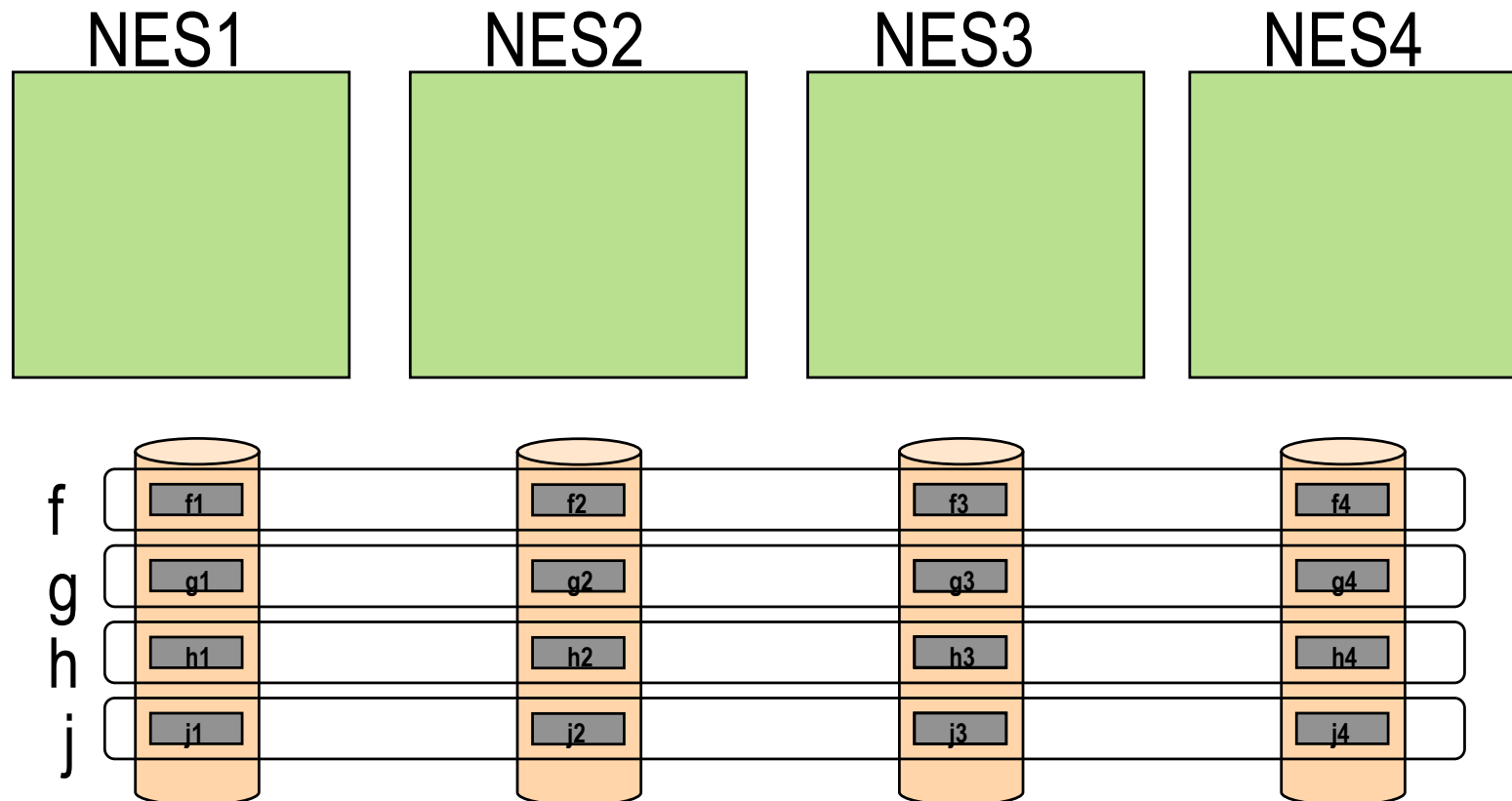
Disposición de datos en SFD



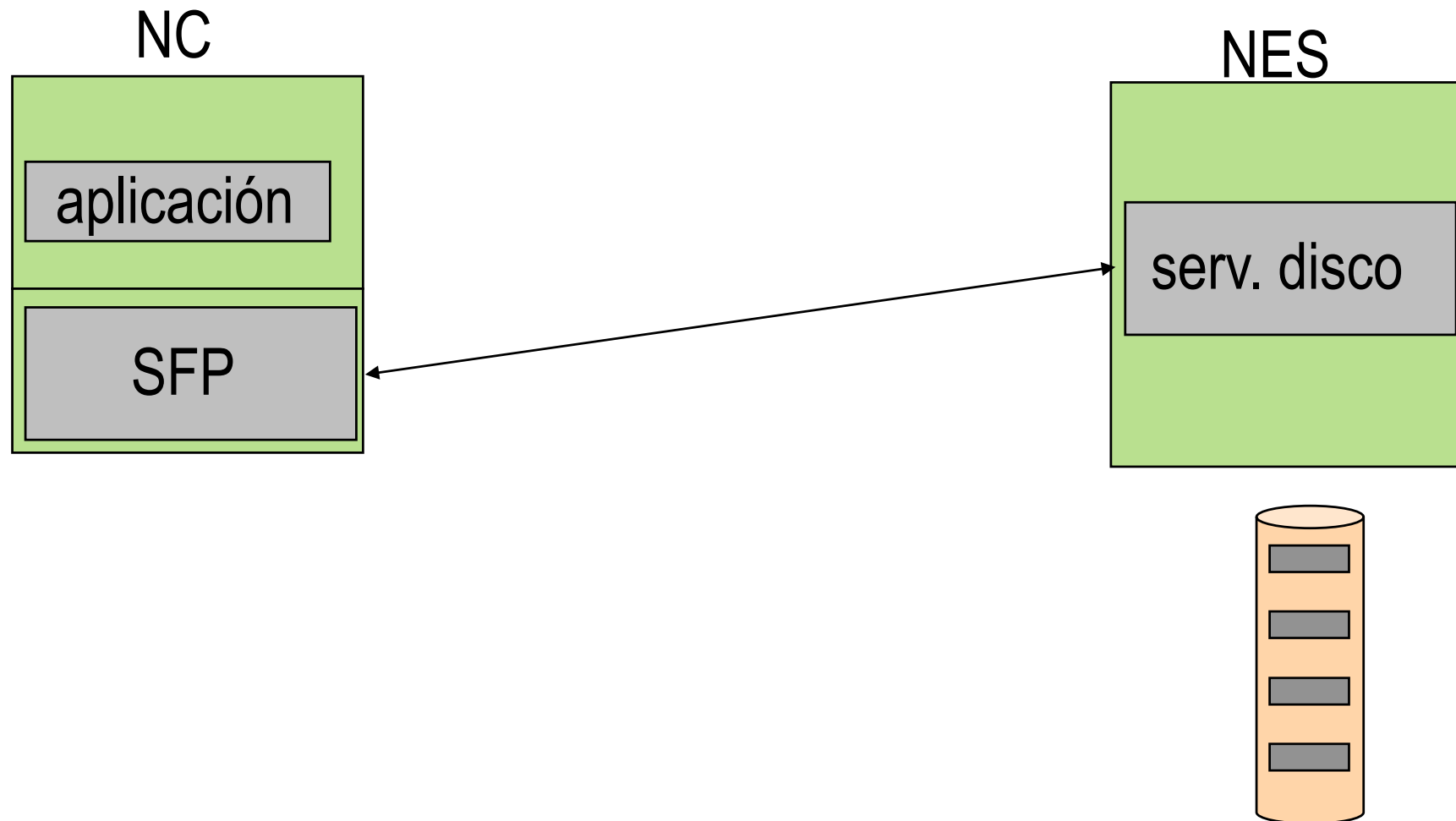
Sistemas de ficheros paralelos

- 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

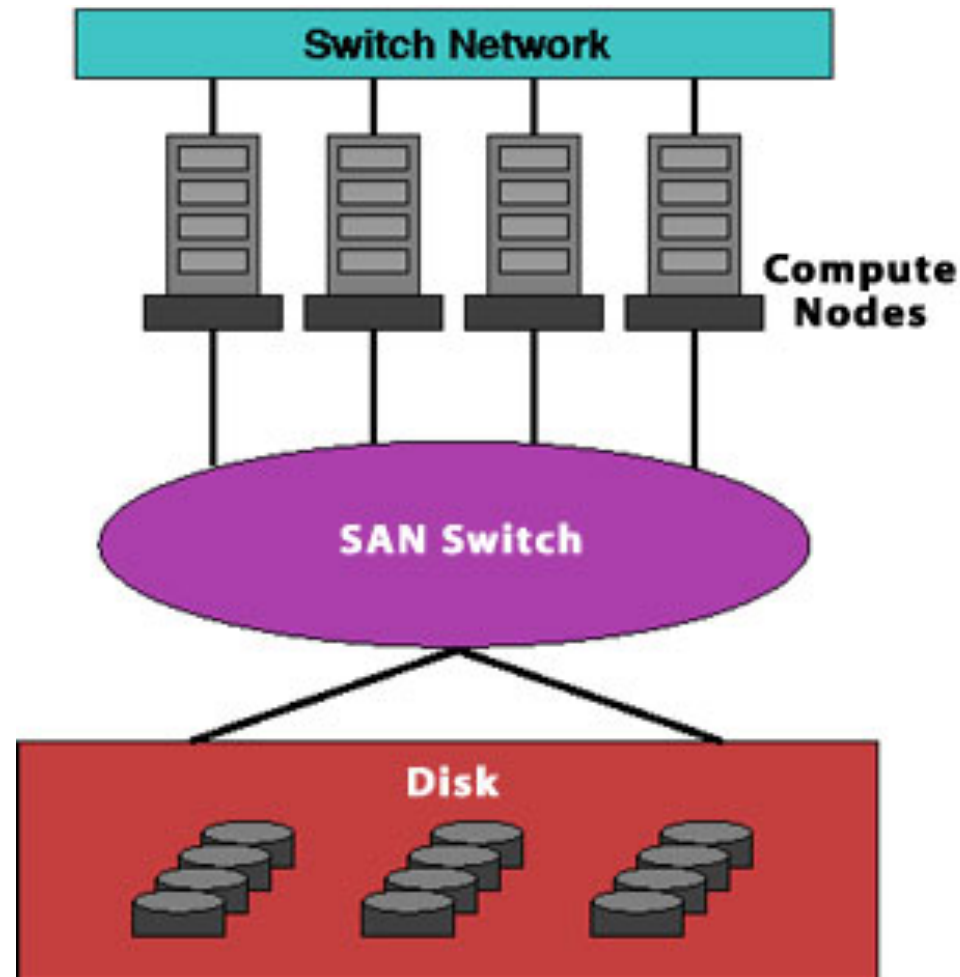
Disposición de datos en SFP: *stripping*



Arquitectura SFP

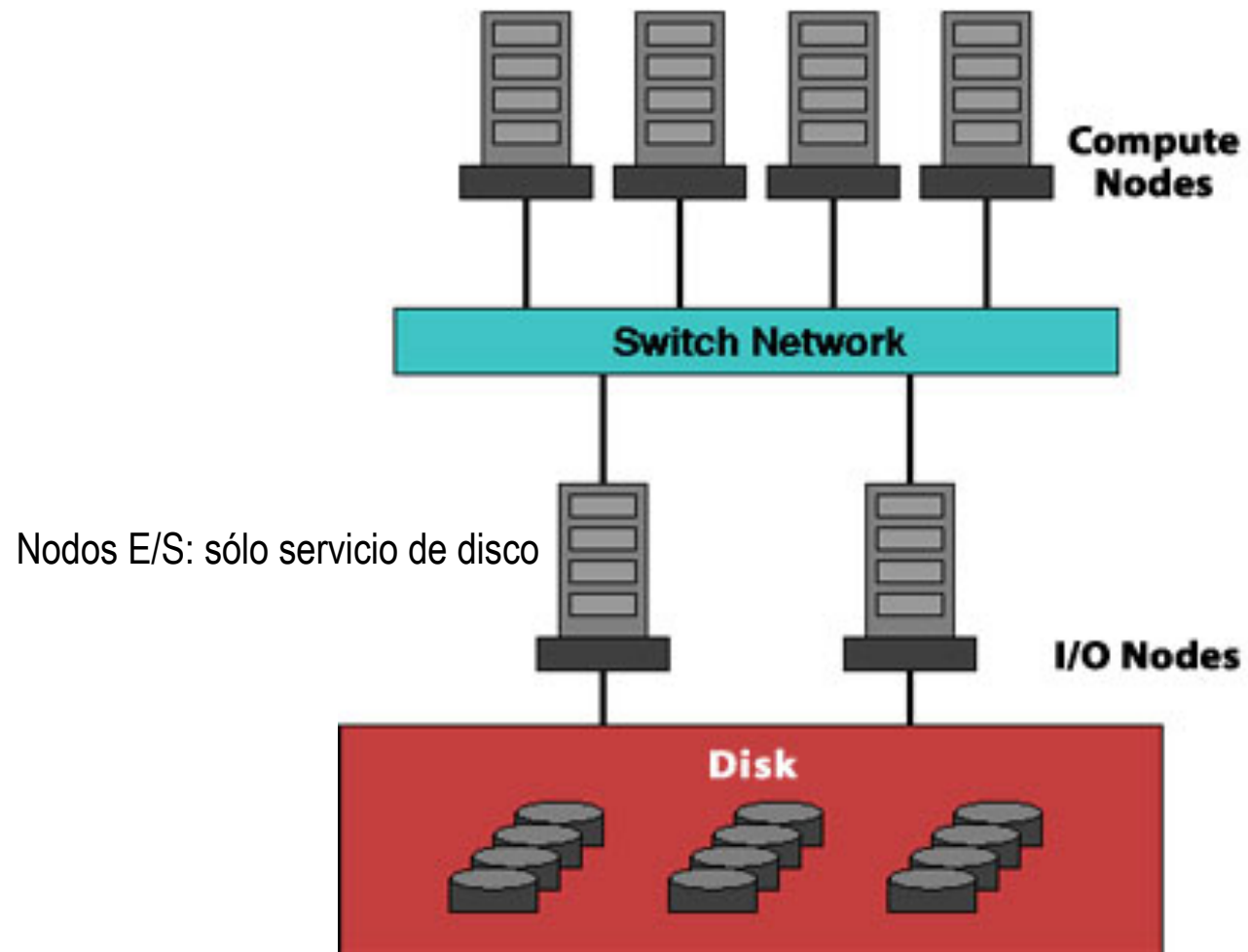


Configuración basada en SAN



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

Configuración basada en nodos de E/S

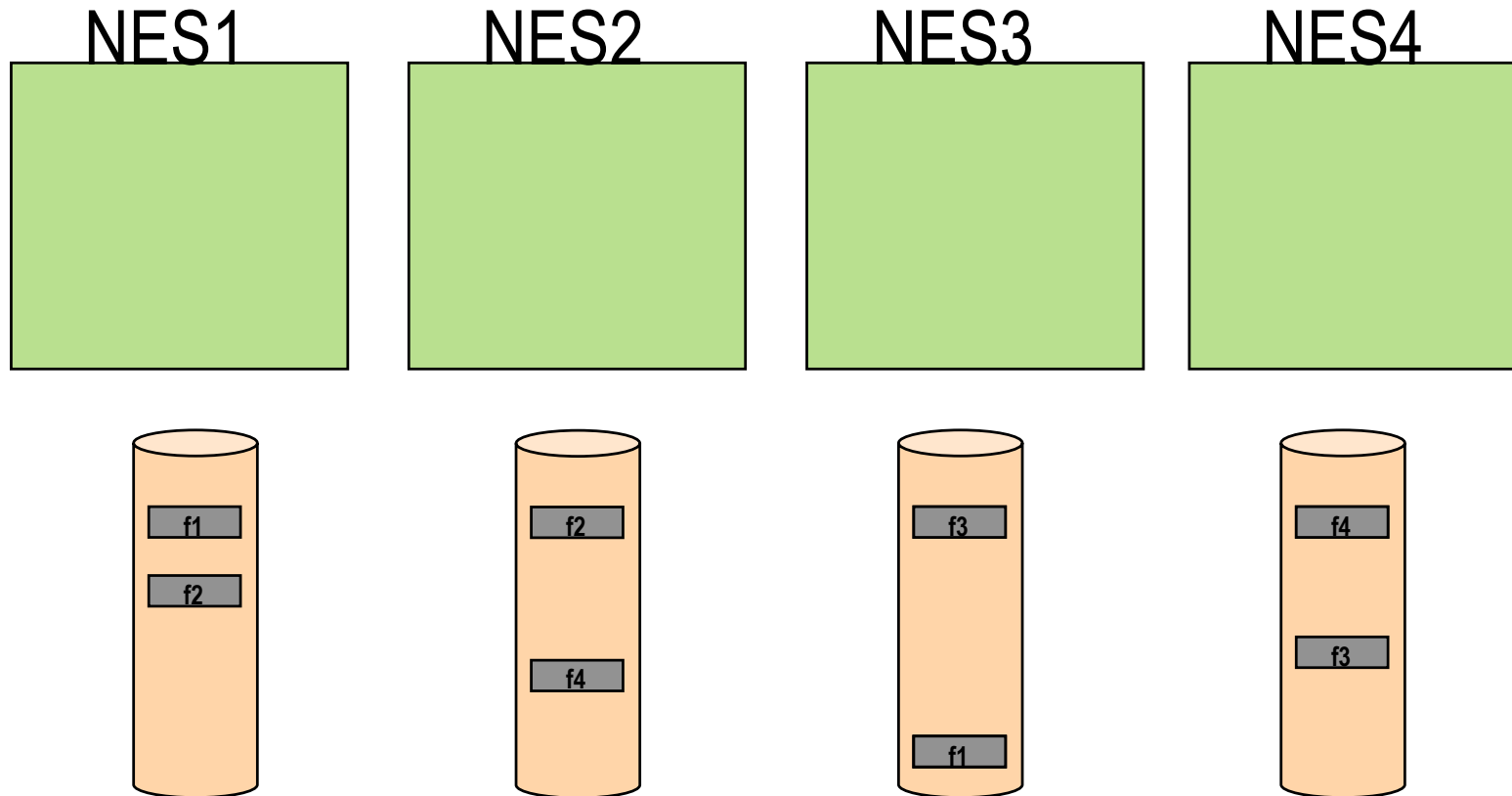


<http://www.ncsa.illinois.edu/UserInfo/Data/filesystems/>

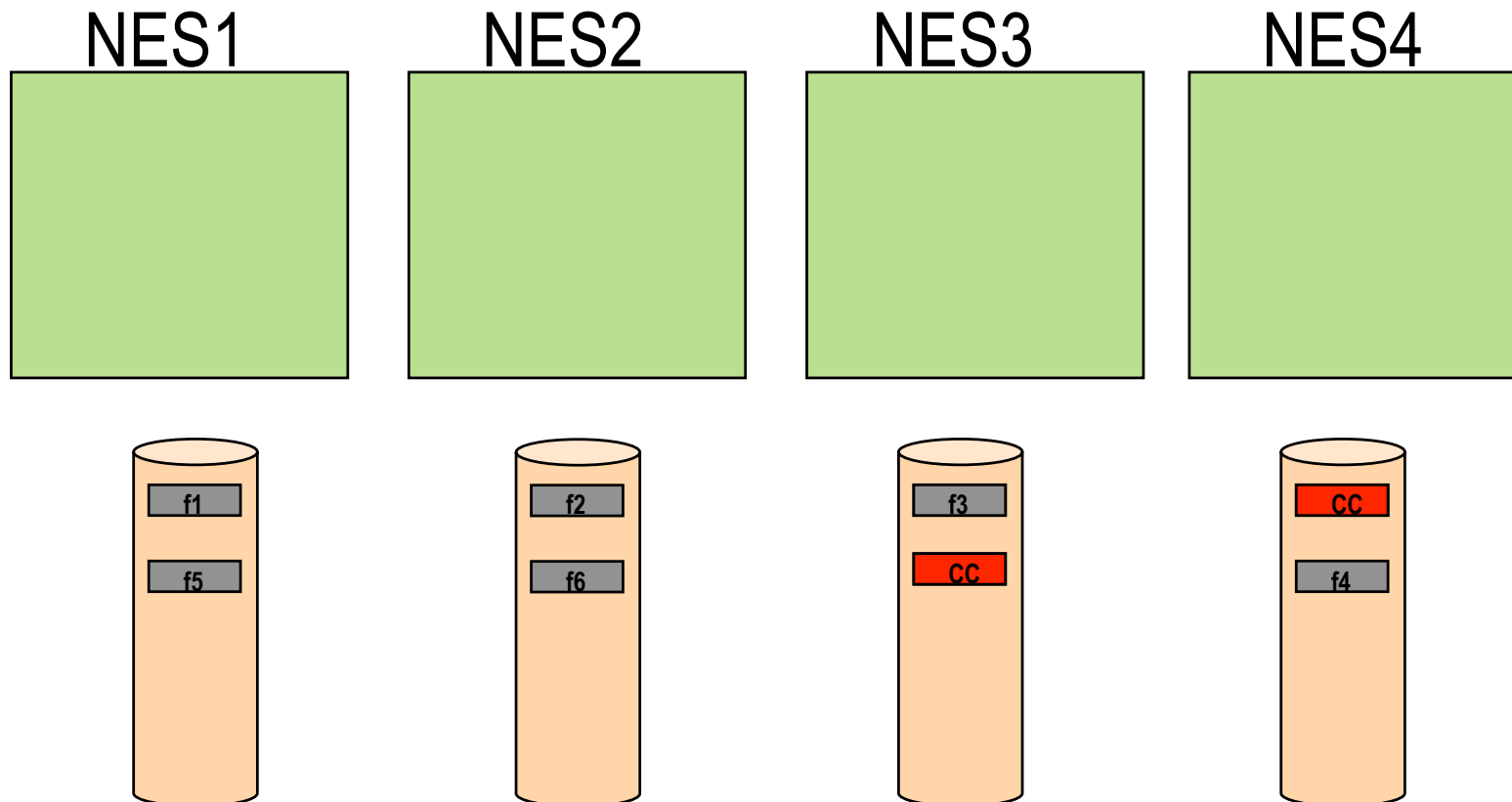
Redundancia de datos

- *Stripping* 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

SFP con replicación



SFP con códigos correctores



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

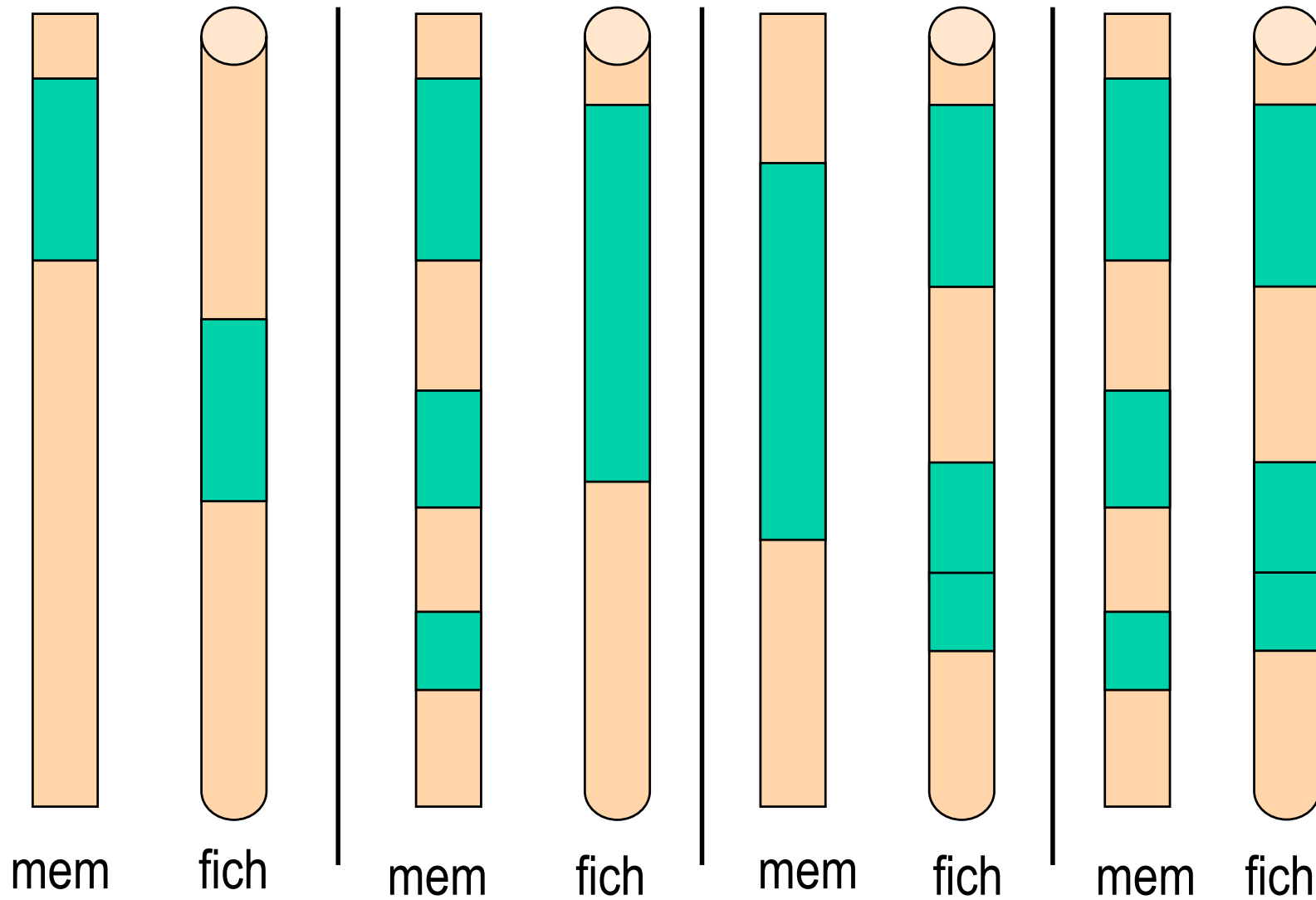
Caché de datos

- 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

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 contigüidad 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 contigüidad en fichero y/o en *buffers*
 - Solución usada en MPI-IO

Accesos no contiguos



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)`

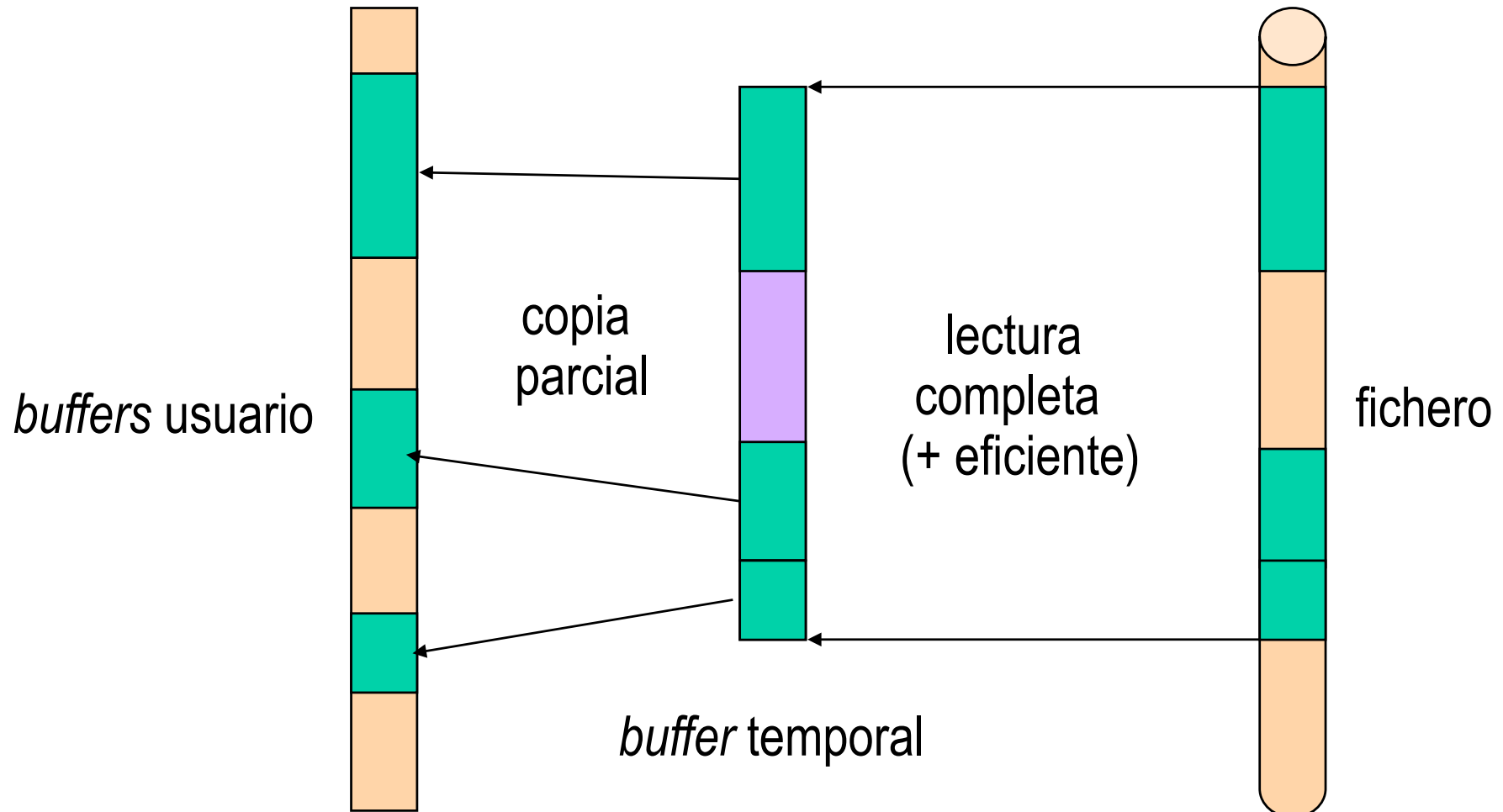
`int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`

Operaciones de E/S en MPI-IO

- Lectura/escritura con posición de acceso implícita
 - Síncronas: *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
 - Síncronas: *MPI_File_read_at*, *MPI_File_write_at*
 - Asíncronas: *MPI_File_iread_at*, *MPI_File_iwrite_at*
- Lectura/escritura con posición de acceso compartida
 - Síncronas: *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*

Data Sieving

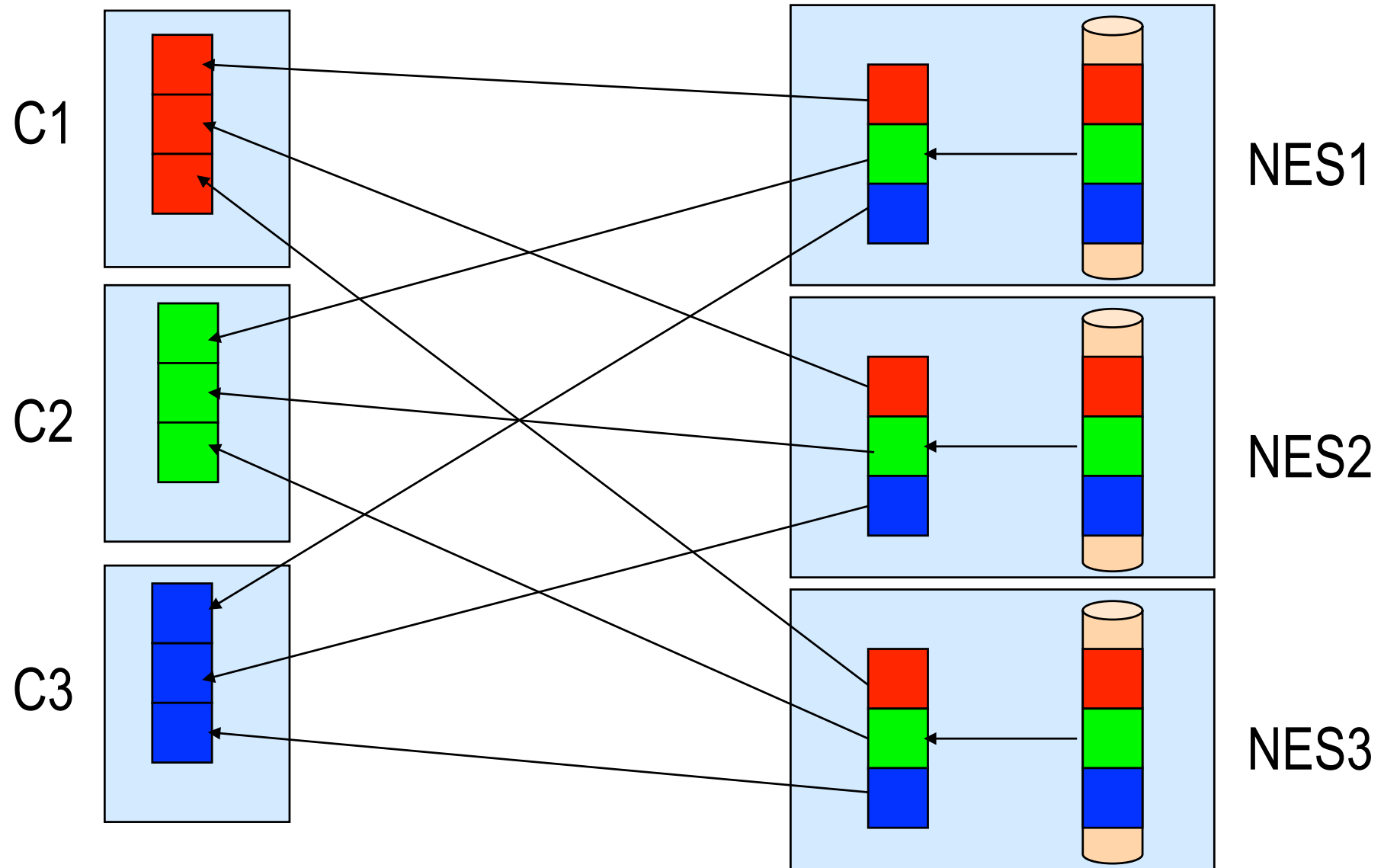
- Algunos SFP: todos los datos \rightarrow *buffer* y de ahí se copian los pedidos
 - Acceso contiguo al fichero pero inútil si datos superfluos no se usan



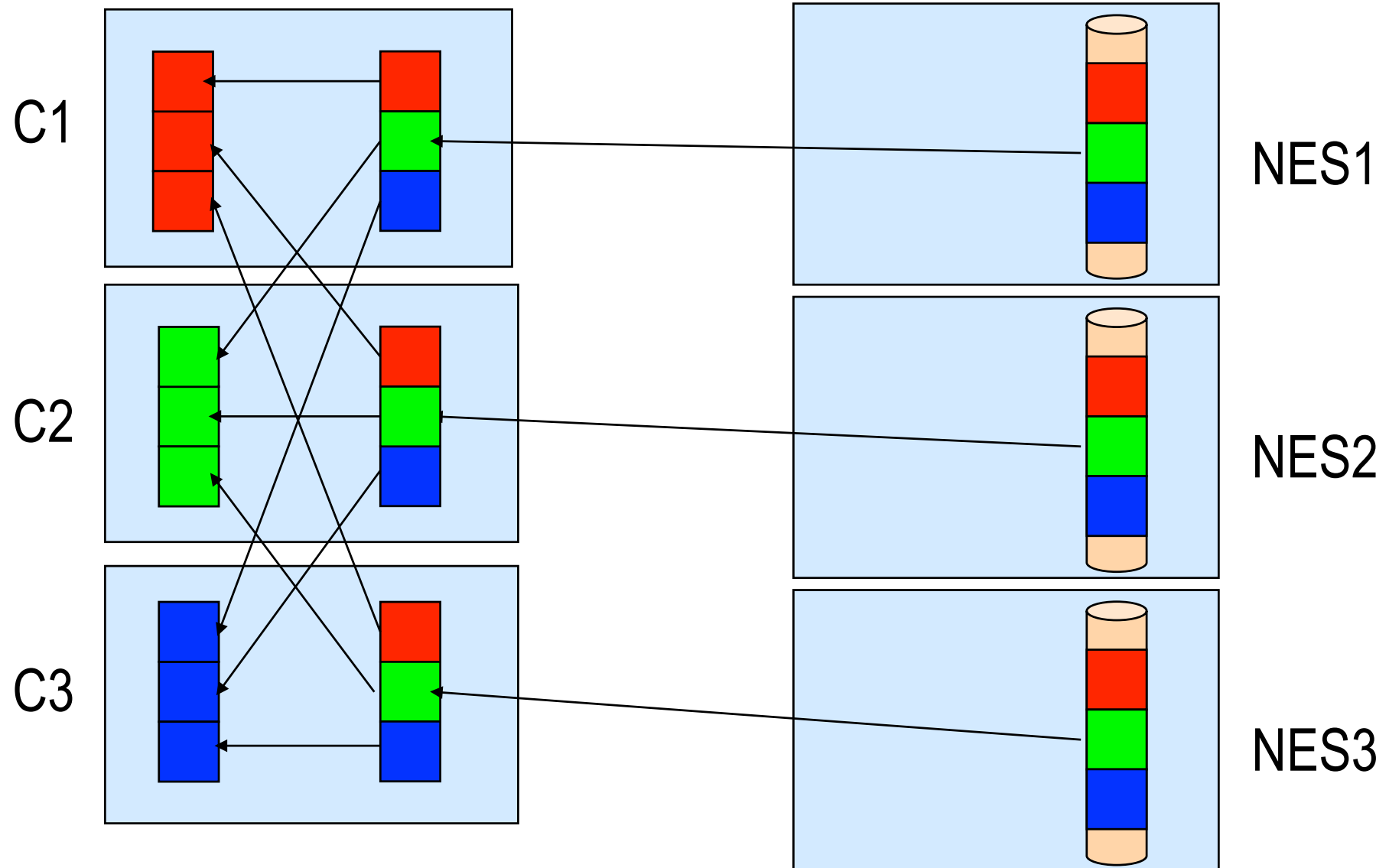
Operaciones colectivas de E/S

- 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

SDIO (lectura)



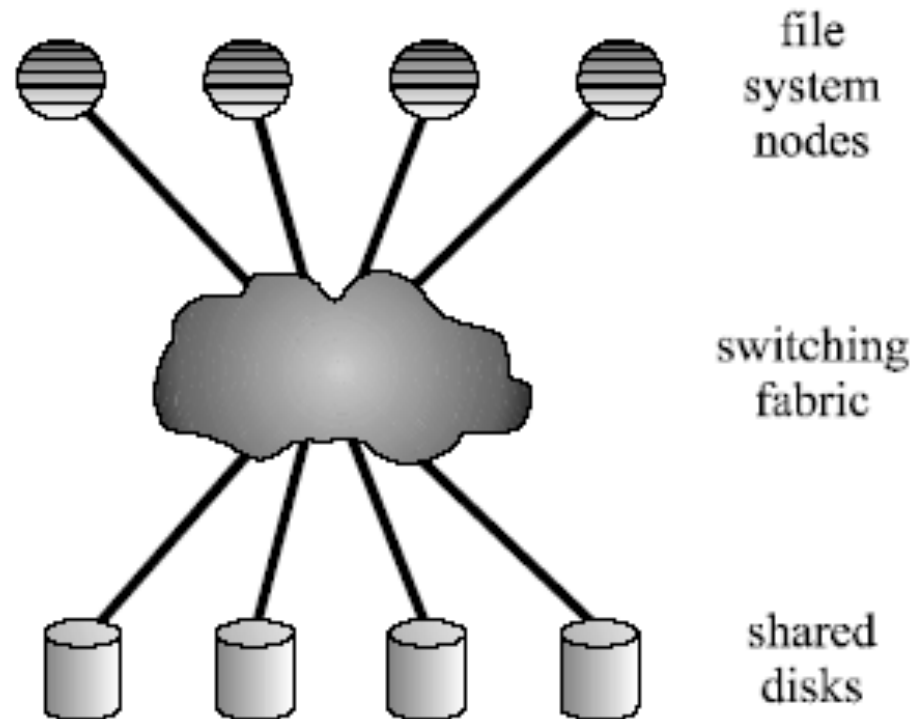
TPIO (lectura)



General Parallel File System de IBM

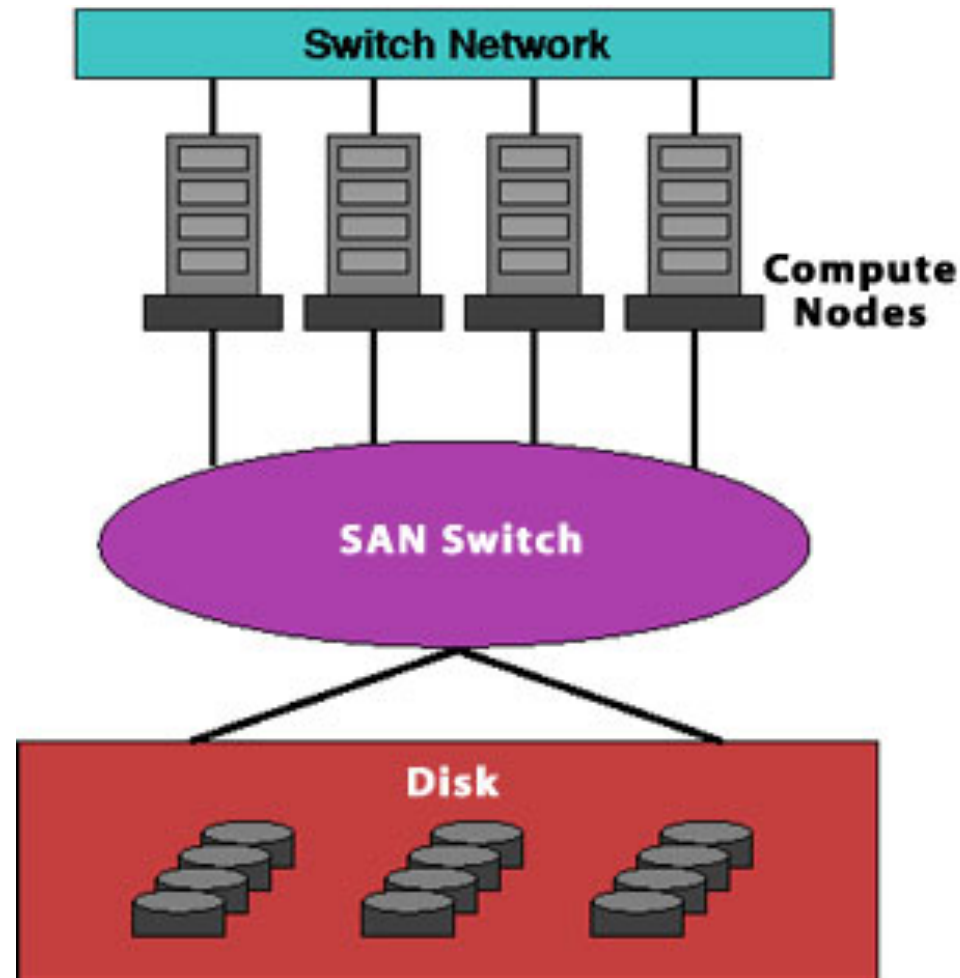
- Sistema de ficheros para clusters
 - Gran escala: decenas de miles de discos
 - Soporte grandes volúmenes, ficheros y directorios (*hashing* extensible)
 - Presente en la mayoría de los Top 500
- Soporte para SAN y nodos con discos: *Shared disk file system*
- Sistemas heterogéneos (versiones para AIX, Linux, Windows)
- Semántica POSIX (excepto *atime*)
 - Escrituras atómicas incluso aunque POSIX actual no lo requiera
- Facilidades para implementar biblioteca MPI-IO
- 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

Arquitectura *Shared disk file system*



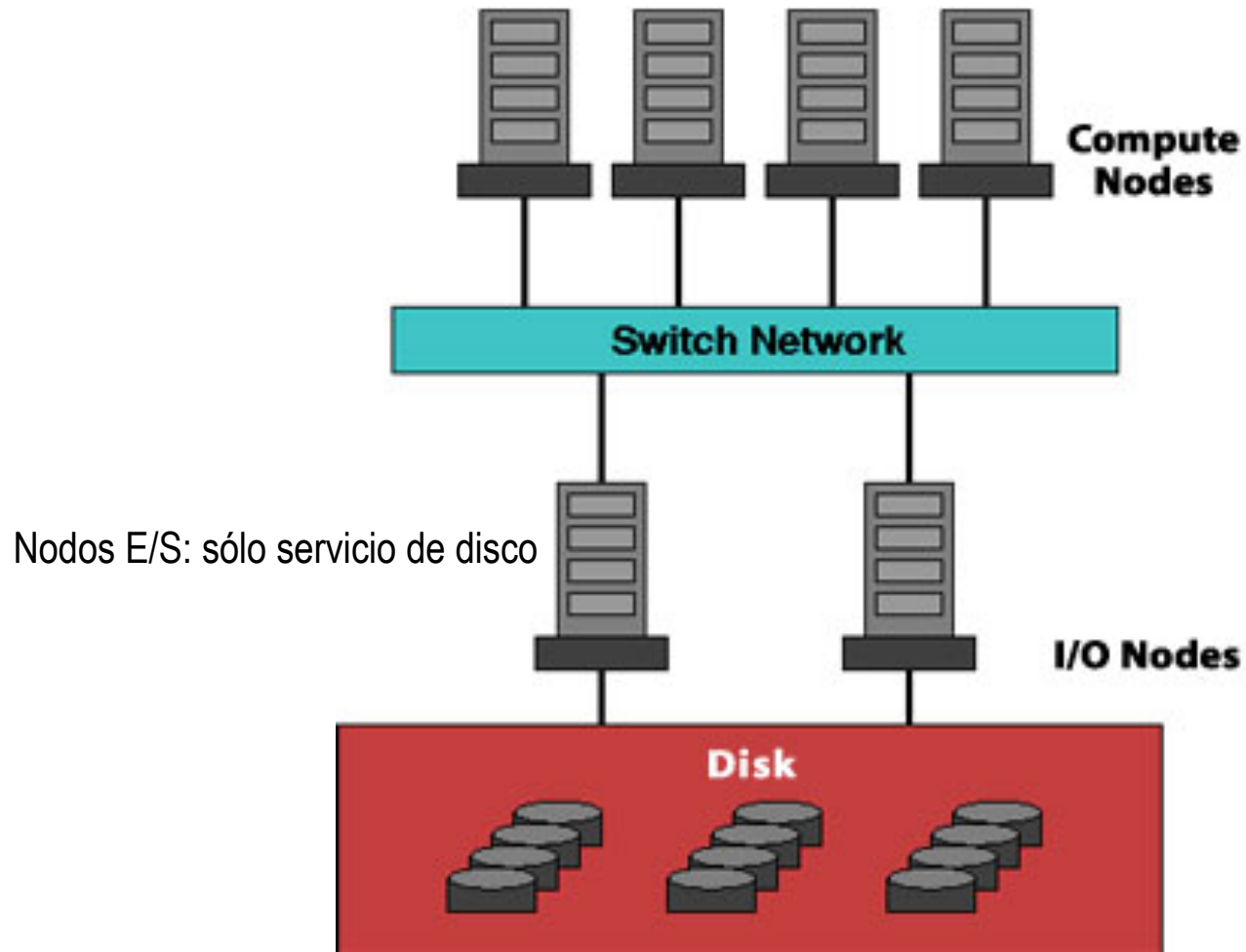
GPFS: A Shared-Disk File System for Large Computing Clusters
Frank Schmuck and Roger Haskin; *FAST '02. USENIX*

Configuración basada en SAN



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

Configuración basada en nodos de E/S



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

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 RAIDs: 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/escribe 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*

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, ftruncate, ..*)
 - 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/escibe 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:
 - $Fecha\ mod = máx(Fecha\ mod\ copias)$ (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

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 (>100MB) y muy grandes (>1GB)
 - 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 → 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 (>1MB) y secuenciales
 - Algunas lecturas pequeñas aleatorias
 - Mayoría escrituras grandes (>1MB) 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 (*≈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!

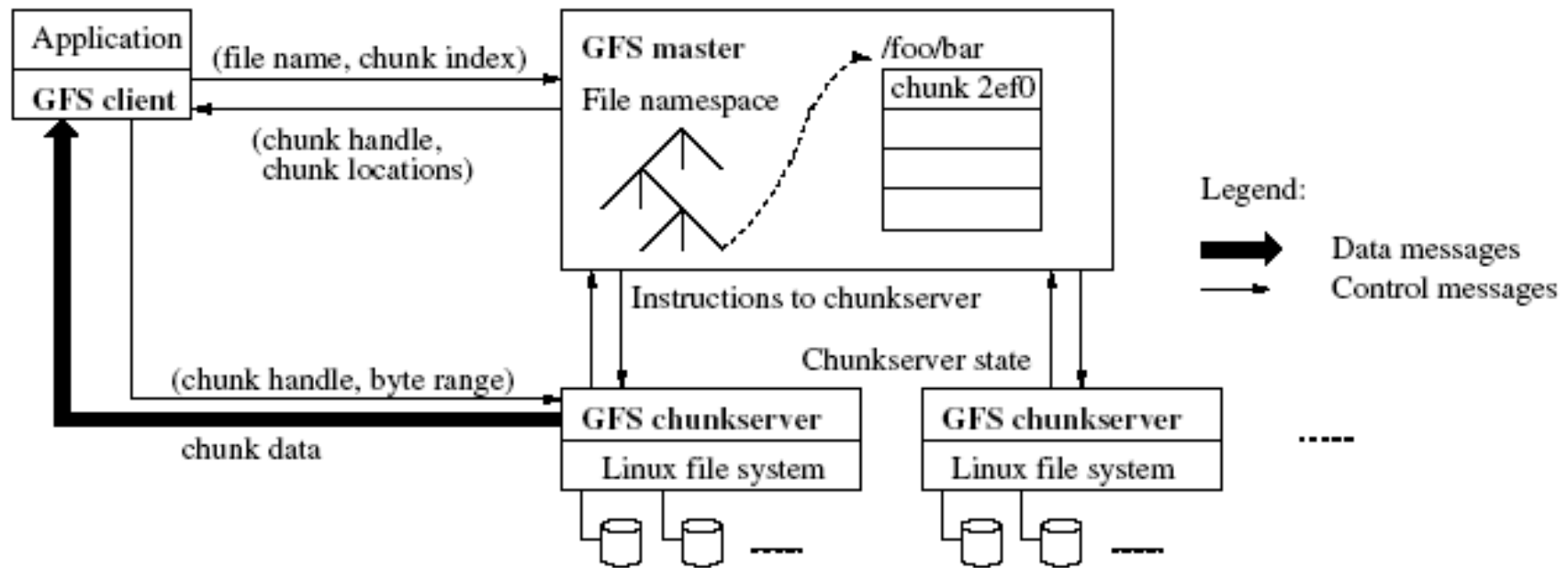
Striping

- Trozos fichero repartidos entre nodos de almacenamiento (NA)
- Tamaño de trozo/*chunk*/*stripe*: ¡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

Arquitectura de GFS



The Google File System

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

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 n° 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