Universidad De Granada

E.T.S de Ingenierías Informática y de Telecomunicación
Departamento de Arquitectura y Tecnología de Computadores

Incremento de la localidad de datos en Sistemas de Ficheros

Editor: Editorial de la Universidad de Granada
Autor: Hugo Eduardo Camacho Cruz
D.L.: GR 214-2013
ISBN: 978-84-9028-322-6

Universidad De Granada

E.T.S de Ingenierías Informática y de Telecomunicación
Departamento de Arquitectura y Tecnología de Computadores

Incremento de la localidad de datos en Sistemas de Ficheros

Memoria presentada para optar al grado de Doctor en Arquitectura de Computadores: Perspectivas y Aplicaciones por:

D. Hugo Eduardo Camacho Cruz

Directores:

Dra. Mancia Anguita López
Dr. Antonio Francisco Díaz García

Firma del alumno:

Mancia Anguita López y **Antonio Francisco Díaz García**. Profesores titulares del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada en España.

CERTIFICAN:

Que la memoria titulada **"Incremento de la localidad de datos en Sistemas de Ficheros"**, ha sido realizada **por D. Hugo Eduardo Camacho Cruz** bajo su dirección en el Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada, para optar al grado de Doctor.

Granada, España a 11 de junio de 2012

Dra. Mancia Anguita López
Director de Tesis

Dr. Antonio Francisco Díaz García
Director de Tesis

*A **Linda** y **Hugo Jesús** por su comprensión y apoyo a diario.*

Índice de Contenido

CA**PÍTULO 4**

Implementaciones de Memoria Cache en PVFS2

CAPÍTULO 5

Implementaciones de Memoria Cache en AbFS

CAPÍTULO 6

Evaluación de las Implementaciones de Memoria Cache propuestas en PVFS2 y AbFS

CAPÍTULO 7

Conclusiones y Trabajos Futuros

Apéndice

Bibliografía

Índice de Figuras

Prólogo/Introducción

CAPÍTULO 1

Sistemas de Ficheros Distribuidos y Paralelos

CAPÍTULO 2

Implementación de memoria Cache en Sistemas de Ficheros Distribuidos y Paralelos

CAPÍTULO 3

Sistemas de Ficheros PVFS2 y AbFS

CAPÍTULO 4

Implementaciones de Memoria Cache en PVFS2

CAPÍTULO 5

Implementaciones de Memoria Cache en AbFS

CAPÍTULO 6

Evaluación de las Implementaciones de Memoria Cache propuestas en PVFS2 y AbFS

Índice de Tablas

CAPÍTULO 1

Sistemas de Ficheros Distribuidos y Paralelos

CAPÍTULO 2

Implementación de memoria Cache en Sistemas de Ficheros Distribuidos y Paralelos

CAPÍTULO 3

Sistemas de Ficheros PVFS2 y AbFS

CAPÍTULO 4

Implementaciones de Memoria Cache en PVFS2

CAPÍTULO 5

Implementaciones de Memoria Cache en AbFS

Índice de Listados de Código

CAPÍTULO 4

Implementaciones de Memoria Cache en PVFS2

Agradecimientos

Al iniciar este trabajo una de mis principales motivaciones fue poder seguir formándome como profesional, ya que con esfuerzo y dedicación he podido llegar hasta aquí. Durante este trayecto han sido tanto las personas que han ayudado a mi formación que sería imposible nombrarlas a todas, por lo cual espero y me perdonen si no pongo sus nombres, además de que llenaría mi trabajo únicamente de agradecimientos. Primordialmente quiero agradecer a Dios por ser mi guía en todo momento, ya que él me brinda las fuerzas cada mañana para poder seguir adelante, el encontrarme lejos de mi familia y amigos ha sido una etapa muy difícil, pero la cual me ha enseñado a valorar a cada una de las personas que conforman ese círculo.

Quiero agradecer de manera especial a mis padres (Jesús Camacho Abundis y María Guadalupe Cruz de Camacho) y hermanos (Edgar, Jorge y Leslie), que siempre están ahí cuando los necesito y el hecho de poder escuchar sus voces es para mí un motivo más por el cual debo seguir esforzándome.

También quiero agradecer a mis tutores, porque a lo largo de este tiempo, han dejado en mi algo de ellos, Mancia Anguita, Antonio Díaz y Julio Ortega, que han hecho espacio para irme guiado durante la realización de este trabajo.

Prólogo/Introducción

Las investigaciones en sistemas de ficheros siguen en auge porque los sistemas de ficheros influyen decisivamente en las prestaciones de los accesos al almacenamiento en un sistema de cómputo y se demandan más prestaciones. La necesidad de poder compartir datos a grandes velocidades impulsa que se busquen soluciones que brinden altos rendimientos a un bajo costo en sistemas distribuidos. El sueño de muchos diseñadores es el poder implementar sistemas para que los usuarios obtengan el máximo beneficio y así cubrir sus necesidades principales.

Los sistemas de ficheros permiten guardar información dentro de los dispositivos de almacenamiento; por ejemplo, discos duros, discos flexibles, CDs, etc; Los sistemas operativos necesitan acceder a programas, datos de usuarios, configuración que se almacena en ficheros. Se encargan de obtener las propiedades físicas de los distintos dispositivos de almacenamiento proporcionando una interfaz única que es visible para los usuarios.

Cada Sistema Operativo tiene un sistema de archivo montado por defecto; por ejemplo, NTFS o VFAT en Windows y, ext2, ext3 ó actualmente ext4 en Linux, UFS en Solaris o EFS en IRIX; estos son llamados sistemas de ficheros de discos o sistemas de ficheros locales.

Existen otros tipos de sistemas de ficheros además de los mencionados en el párrafo anterior: los sistemas de ficheros en red o NAS (*Network-Attached Storage*). Estos sistemas de ficheros permiten compartir ficheros y directorios entre distintas máquinas que se encuentran conectadas a una red, de manera que aunque se acceda a un disco conectado a otra máquina se tiene la impresión de trabajar de forma local. Los sistemas de ficheros en red se clasifican en: sistemas de ficheros distribuidos y sistemas de ficheros paralelos. La diferencia principal en estos sistemas es que el segundo permite a los clientes acceder a un único fichero en paralelo por parte de varios nodos clientes, reduciendo así el tiempo de acceso al mismo.

El incremento en prestaciones de los computadores ha incrementado la necesidad de usar caches en los clientes en sistemas de ficheros distribuidos y en sistemas de ficheros paralelos. El uso de caches en los clientes mejora la latencia media de acceso a datos almacenados en disco principalmente porque es probable que se encuentren los datos en la cache y, por tanto, no haya que ir a requerirlos al servidor generando tráfico en la red.

Además la latencia media de acceso al servidor también mejora debido a que se reduce el número de accesos a la red y a los servidores y, por tanto, reduce la penalización por esperas. Esto es muy importante para las aplicaciones que requieren que los datos estén disponibles lo antes posible; es decir, que requieren bajas latencias de acceso a los datos almacenados en disco.

En este trabajo, se presenta la implementación de una cache en clientes para PVFS2 (*Parallel Virtual File System*) y la implementación de cache en clientes y servidores para AbFS. PVFS2 es un sistema de ficheros paralelo de código abierto que no implementa cache en los clientes. Sucesor original del sistema de ficheros PVFS [Philip00] desarrollado en la Universidad de Clemson, en la década de los 90´s. PVFS distribuye los datos a través de los distintos servidores de Entrada/Salida, permitiendo con esto accesos concurrentes para múltiples tareas de aplicaciones paralelas que se ejecutan en distintos nodos. AbFS es un sistema de ficheros distribuido que se está desarrollando en el Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada. Se ha implementado con el objetivo de conseguir unas buenas prestaciones en el acceso a datos y metadatos. Se ha conseguido este objetivo distribuyendo los metadatos entre varios servidores de metadatos y haciendo uso de cache de metadatos en el servidor y en los clientes. AbFS dispone de caches en clientes y servidores mejorando tanto la latencia local que observan las aplicaciones como la latencia global de acceso al sistema de ficheros.

La Figura 1 [Anguita11] relaciona distintas clases o tipos de sistemas de ficheros y distintos criterios de clasificación de sistemas de ficheros. Los tipos de sistemas de ficheros que aparecen son sistema de ficheros local, sistema de ficheros global, sistema de ficheros simétrico, sistema de ficheros asimétrico, sistema de ficheros SAN (*Storage Area Network*), sistema de ficheros NAS (*Network-Attached Storage*), etc. Los criterios de clasificación que relaciona la figura son: espacio de nombres (*namespace*) único para todos los nodos de cómputo del sistema o múltiple (uno para cada nodo), gestores de metadatos dedicados o no,

protocolo de acceso a nivel de bloque o usando un protocolo de red, distribución de ficheros o bloques de fichero y número de servidores de datos (uno o múltiples).

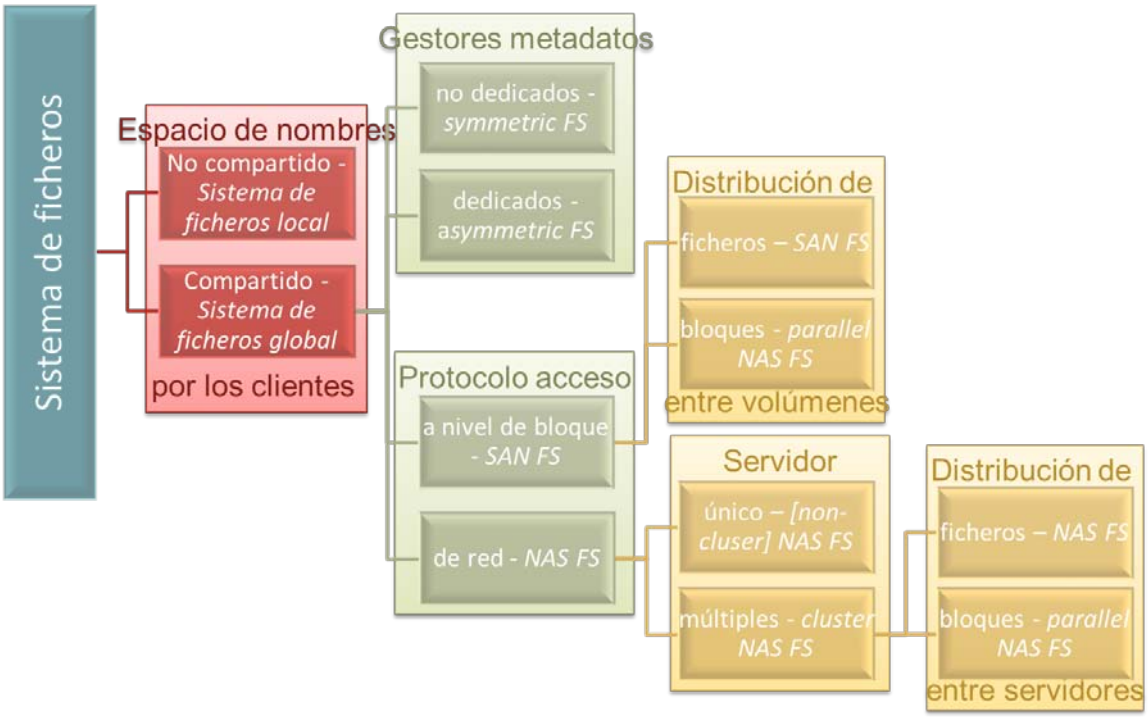


Figura 1. Relación entre distintas clases o tipos de sistemas de ficheros y distintos criterios de clasificación de sistemas de ficheros (espacio de nombres, gestores de metadatos, protocolo de acceso, distribución de datos y número de servidores de datos) (FS: File System)

Espacio de nombres

- No compartido: *local file systems (ext3, ext4, XFS …)*
- Compartido: *global file systems (NFS, **PVFS**, Lustre, OCFS, pNFS, Ceph, **AbFS**, …)*

Protocolo de acceso

- Nivel de bloque (FC, iSCSI): *SAN (or shared disk) file systems (GPFS, GFS, CXFS, OCFS, …)*
- De red (TCP/IP): *NAS (or network) file systems (NFS,**PVFS**, Lustre, Ceph, **AbFS**,…)*

Nº de servidores

- Uno: *NAS file systems (Network-Attached Storage) (NFS, AFS,…)*
- Múltiple : *cluster NAS file systems (**PVFS**, Lustre, pNFS, Ceph, **AbFS**, …)*

Desacopla acceso a metadatos del acceso a datos

- Sí: *asymmetric file systems (CXFS, Lustre, Ceph, pNFS…)*
- No: *symmetric file systems (GPFS, **PVFS2**, **AbFS**, …)*

Figura 2 Resumen de clasificación de sistemas de ficheros con ejemplos. Se ha destacado AbFS en negrita

Por su parte la Figura 2 [Anguita11] [Díaz12] muestra ejemplos de las distintas clases de sistemas de ficheros. Esta figura permite caracterizar la versión actual de AbFS y PVFS2, los sistemas de ficheros para los que se ha implementado cache, dentro del conjunto de sistemas de ficheros.

A continuación se presenta una breve reseña del contenido de cada uno de los capítulos descrito en este trabajo.

El primer Capítulo hace mención al objetivo del trabajo. Presenta los sistemas de ficheros distribuidos y sistemas de ficheros paralelos y comenta sus características, ventajas y desventajas.

El segundo Capítulo hace referencia a las implementaciones de memoria cache en los clientes para sistemas de ficheros distribuidos y sistemas de ficheros paralelos. Tras una breve aproximación al tema, se introducen algunas facetas relevantes: uso de cache, qué es una cache de datos, qué es una cache de metadatos, aspectos a tener en cuenta en el diseño de una cache de datos (ubicación, tamaño, políticas de reemplazo y políticas de actualización), qué es una cache colaborativa. Se incluyen además varios ejemplos. Algunos de los sistemas que implementan cache de datos son los sistemas de ficheros distribuidos AFS, Sprite, NFS y GFS, y los sistemas de ficheros paralelos Parfisys y GPFS.

En el tercer Capítulo presenta el sistema de ficheros virtual paralelo en sus diferentes versiones (PVFS y PVFS2). Se inicia con una pequeña introducción de sus orígenes, así como una breve descripción del sistema. Continuando con aspectos relacionados: características, componentes y arquitectura. Además se muestra la diferencia que existe entre versiones. Posteriormente se detallan algunas de las implementaciones realizadas de memoria cache para datos en los clientes de PVFS. Por último, se introduce el sistema de ficheros distribuidos AbFS (*Abierto File System*).

El cuarto y quinto capítulo presentan de forma detallada las implementaciones de memoria cache propuestas sobre los sistemas de ficheros PVFS2 y AbFS.

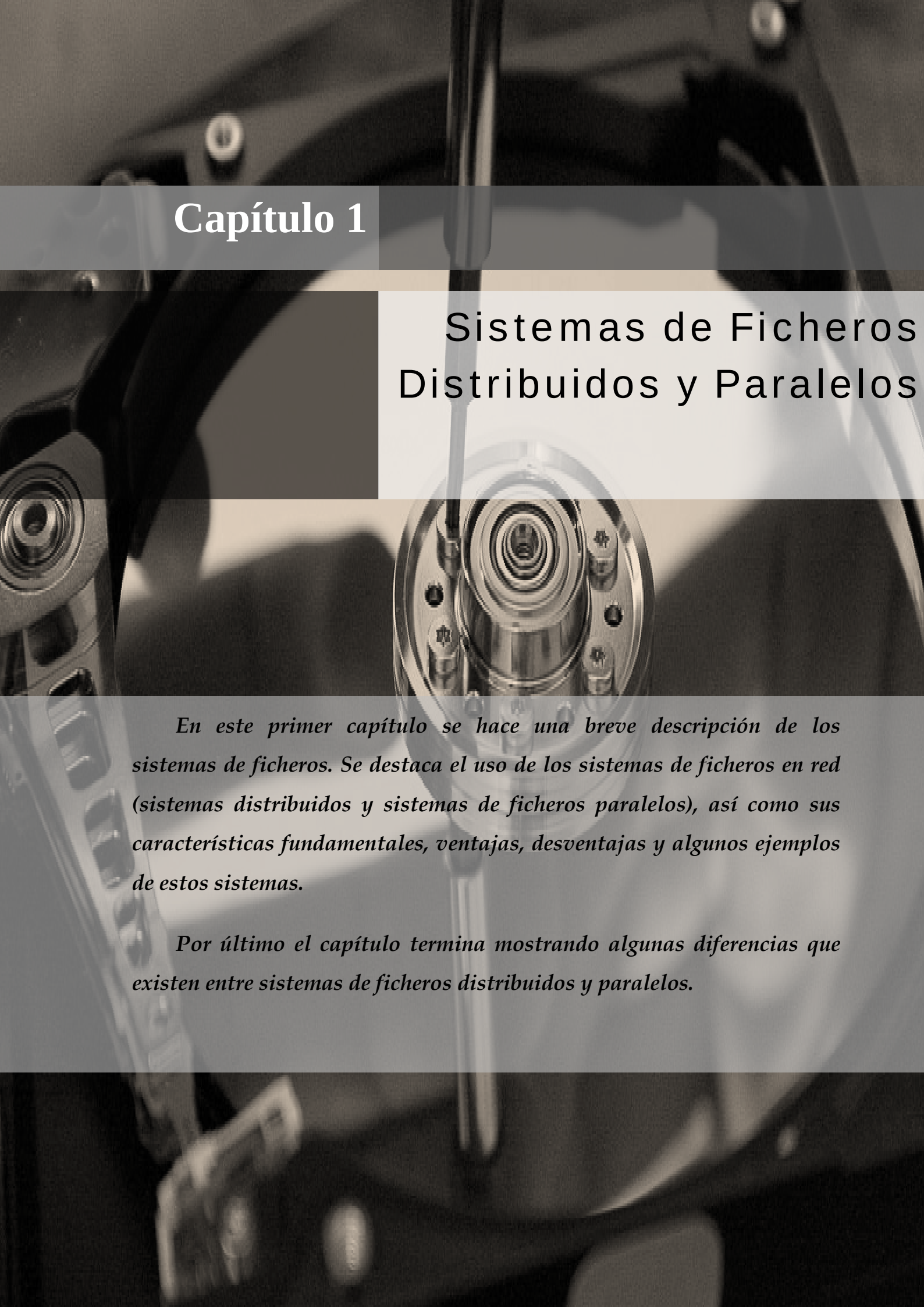
El sexto capítulo muestra la evaluación de las implementaciones de memoria cache en los sistemas de ficheros PVFS2 y AbFS. Inicialmente se da conocer el hardware empleado, así como las herramientas de medición, configuraciones utilizadas y resultados obtenidos.

El séptimo capítulo detalla las conclusiones que se hacen en base al desarrollo de este trabajo de investigación, así como trabajos futuros que se pretenden abordar.

Capítulo 1

Sistemas de Ficheros Distribuidos y Paralelos

SUMARIO



Capítulo 1

Sistemas de Ficheros Distribuidos y Paralelos

En este primer capítulo se hace una breve descripción de los sistemas de ficheros. Se destaca el uso de los sistemas de ficheros en red (sistemas distribuidos y sistemas de ficheros paralelos), así como sus características fundamentales, ventajas, desventajas y algunos ejemplos de estos sistemas.

Por último el capítulo termina mostrando algunas diferencias que existen entre sistemas de ficheros distribuidos y paralelos.

1.1 Introducción

Un sistema de ficheros es un software que organiza y gestiona los datos almacenados en dispositivos de almacenamiento y los presenta a los usuarios o programas como ficheros o directorios (carpetas) lógicamente organizados en una jerarquía de directorios (estructura o árbol de directorios). Establece una relación entre los ficheros, directorios y dispositivos de almacenamiento (discos duros, usb, discos flexibles, cd´s, etc). Sus principales funciones son las de:

- ✓ Organizar
- ✓ Almacenar
- ✓ Recuperar
- ✓ Gestión de nombres y atributos
- ✓ Coutilización o control de cuotas
- ✓ Protección de los ficheros

La figura 1.1 muestra la jerarquía en la que se sitúa un sistema de ficheros. Como se puede ver éste se localiza entre la parte física (particiones y volúmenes) y la estructura de ficheros y directorios vista por las aplicaciones.

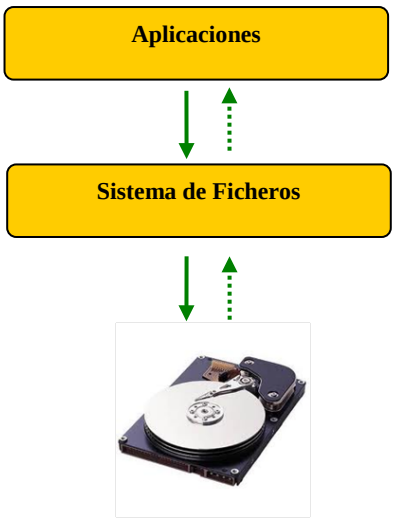


Figura 1. 1 Ubicación de Sistema de Ficheros

Existen diferentes tipos de sistemas de ficheros, que ofrecen distintas prestaciones y características. Se pueden clasificar como:

- ✓ Sistemas de Ficheros Locales
- ✓ Sistemas de Ficheros Distribuidos
- ✓ Sistemas de Ficheros Paralelos

Los Sistemas de Ficheros Locales son aquellos con los que utilizan en el día a día la mayoría de las personas que hacen uso de un ordenador, móvil (teléfono celular), reproductor de música, etc. Estos equipos requieren un sistema de ficheros local para administrar los datos que almacenamos en ellos. Algunos ejemplos de los sistemas de ficheros más conocidos son:

- ✓ NTFS (*New Technology File System*), FAT (*file allocation table*), para Microsoft Windows [Microsoft05],
- ✓ ext2,ext3,ext4 (*Extended File System*) para LINUX [Linux, ext11],
- ✓ HSF (*Hierarquical File System*) para MacOS [HFS04].

Hay que mencionar que conforme avanza el tiempo la necesidad de compartir información entre distintos usuarios va en crecimiento. Investigaciones realizadas en los últimos años sobre sistemas de ficheros han permitido programar aplicaciones que requieren una gran cantidad de datos. Estos avances en sistemas de ficheros también han aumentado las prestaciones de los sistemas de cómputo percibidas por los usuarios.

Inicialmente estas investigaciones fueron impulsadas por la necesidad de compartir información entre los distintos nodos conectados a través de red de interconexión, dando como resultado el diseño de sistemas en ficheros en red. Todo esto permitió que se implementaran sistemas de ficheros distribuidos.

1.2 Sistemas de Ficheros Distribuidos

Un sistema de ficheros distribuido se caracteriza por estar formado por un número de ordenadores que de forma coordinada trabajan por un fin en común: que múltiples procesos pueden compartir datos durante un periodo de tiempo de manera segura y fiable [Tanenbaum02]. Inicialmente lo que hace la diferencia entre los sistemas de ficheros distribuidos sobre los locales, es que los primeros ofrecen un acceso a ficheros remotos y además brinda y facilita la compartición de información entre los distintos usuarios o clientes. La figura 1.2 ilustra el modelo cliente/servidor utilizado en un sistema de ficheros distribuidos.

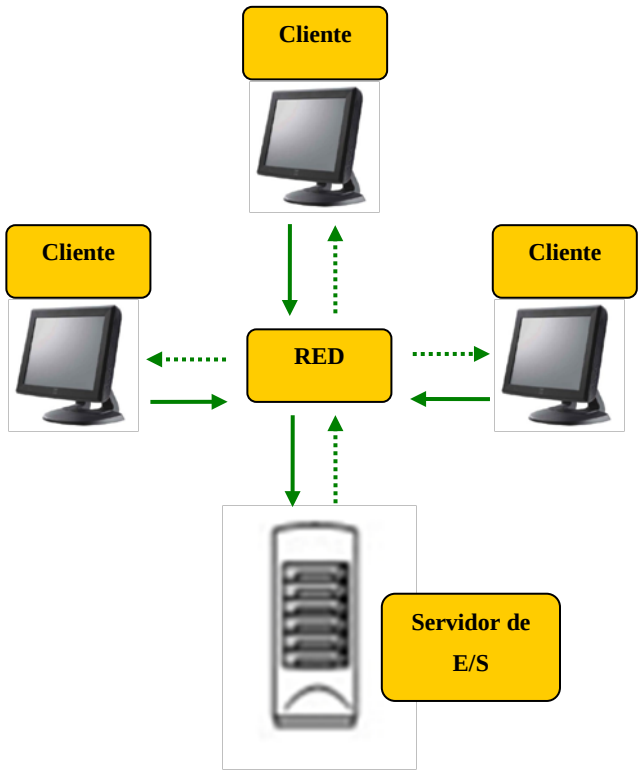


Figura 1. 2 Modelo de Sistema de ficheros distribuidos (cliente/servidor)

Un sistema de fichero distribuido requiere cumplir una serie de características para su buen funcionamiento. El siguiente apartado enumera algunas de las características más importantes de este tipo de sistemas.

1.2.1 Características de los Sistemas de Ficheros Distribuidos

Algunas de las características importantes de los sistemas de ficheros distribuidos son: [Lafuente11]

- ✓ Proporcionar almacenamiento
- ✓ Identificar los ficheros en un espacio de nombres
- ✓ Permite accesos concurrentes

Otras características en los sistemas de ficheros distribuidos es que algunos de ellos son sistemas de código abierto, lo que facilita su manipulación y crecimiento, además puedan implementarse en diversas arquitecturas. Esto último hace que estos sistemas pueden ser heterogéneos. También los sistemas de ficheros distribuidos buscan proporcionar:

- ✓ **Transparencia**. Permitir que se ejecuten las mismas acciones sobre los ficheros sin importar que estos estén de forma local o remota. Es decir, que el usuario no sea consciente de si está o no accediendo a un fichero remoto.

- ✓ **Robustez ante fallos.** El servidor no debe verse afectado por los fallos de los clientes, la interfaz ofrecida a los clientes debe proporcionar en lo posible operaciones que garanticen la corrección ante peticiones repetidas (por sospecha de error) al servidor.

- ✓ **Disponibilidad y tolerancia a fallos.** Implica alguna forma de replicar datos.

- ✓ **Escalabilidad.**

1.2.2 Ejemplos de Sistemas de Ficheros Distribuidos

Un ejemplo de sistema de fichero distribuido es NFS (*Network File System*) [Russel85], que se ha diseñado para permitir accesos remotos a ficheros. Es fácilmente portable a otros sistemas operativos y a distintas arquitecturas. El sistema NFS se caracteriza por un diseño básico, que consiste de un protocolo, un servidor y uno o más clientes, los cuales pueden actuar como ambos a la vez. Los clientes acceden de manera remota a los datos que se encuentran almacenados en el servidor. De esta forma el computador del cliente necesita menos espacio de disco debido a que los datos se encuentran centralizados en un único lugar, pero pueden ser accedidos y modificados por varios usuarios, de tal forma que no es necesario tener que replicar la misma información para que puedan acceder varios a la vez.

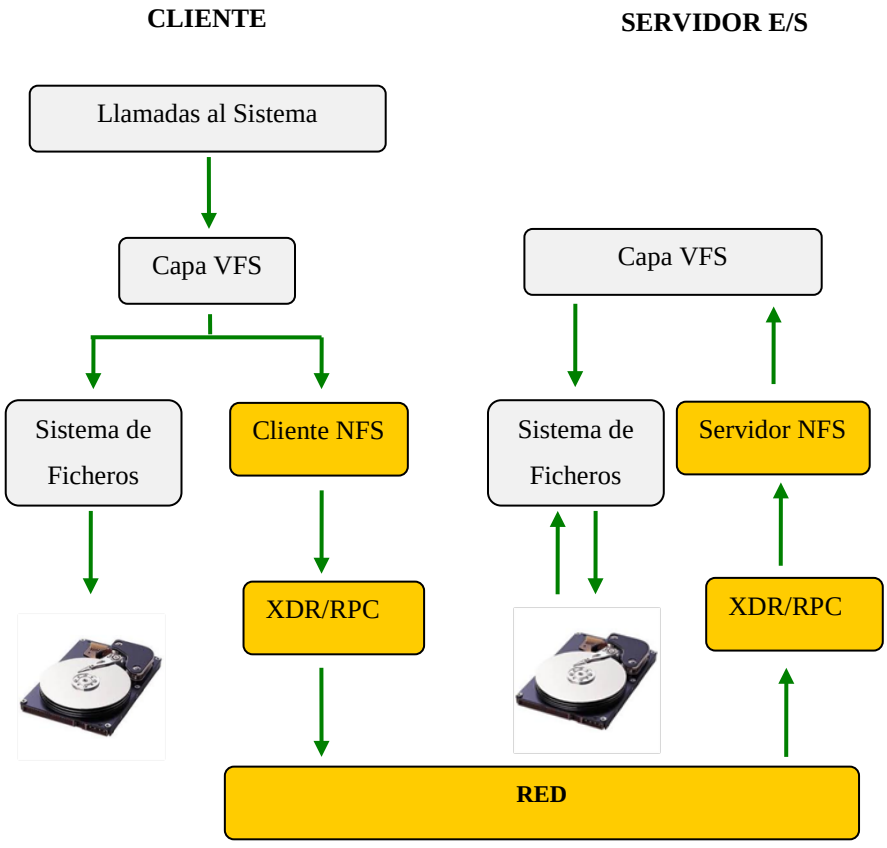


Figura 1. 3 Arquitectura de NFS

La figura 1.3 muestra la arquitectura de NFS. Cuando una petición sobre un fichero se realiza por parte del cliente, dicha petición es enviada al sistema virtual de ficheros (VFS), quien mantiene una tabla de ficheros abiertos. A cada entrada se le indica si el fichero es local o remoto. Si el fichero es local hay un puntero al sistema de fichero local (ejemplo, ext3), si el fichero es remoto, la petición es enviada al cliente de NFS que interpreta las llamadas al sistema de ficheros y genera las peticiones hacia el servidor mediante llamadas de procedimiento remoto (RPC, *Remote Procedure Call*). Una vez que la petición es recibida por el servidor de NFS a través del protocolo RPC, ésta se envía al sistema virtual de ficheros, y posteriormente se accede a los datos del sistema de ficheros local

Otros ejemplos de sistemas de ficheros distribuidos son:

Ceph File System [Weil07], diseñado con la finalidad de mejorar el rendimiento y la confiabilidad. Ceph mejora el rendimiento y la confiabilidad a través de la separación de datos y metadatos, es decir que las operaciones de metadatos (abrir, re-nombrar, etc.) son gestionadas por un servidor de metadatos, mientras que los clientes interactúan directamente con los dispositivos de almacenamiento para ejecutar las operaciones de entrada/salida (lectura / escritura).

Google File System [Google03], es un sistema de ficheros distribuido escalable diseñado para cargas masivas de datos.

CODA [Satyanarayanan89], fue desarrollado con el fin de mantener alta disponibilidad en los datos, y emplear replicación utilizando grupos de almacenamiento.

SPRITE [Ousterhout87]. Los principales objetivos fueron que soportara alto rendimiento y una buena semántica en la compartición de ficheros. Por lo que implementó la semántica de UNIX. Sprite realiza llamadas similares a las que se proveen en UNIX, sin embargo agrega características adicionales. Una de estas características es, el uso de memoria cache tanto en los servidores como en los clientes del sistema.

1.2.3 Ventajas de los Sistemas de Ficheros Distribuidos

Algunas de las ventajas de estos sistemas son:

- ✓ Accesibilidad de los ficheros, los ficheros están más accesibles, debido a que puede accederse a ellos (a los servidores) a través de varios clientes.

- ✓ Los servidores pueden ofrecer una gran capacidad de almacenamiento, es costoso y poco práctico suministrar a cada cliente esa capacidad.

- ✓ Fiabilidad, es un sistema que puede ser tolerante a fallo. En el caso de sufrir algún fallo en un componente, el sistema no deja de trabajar, es decir permite que se distribuya la aplicación entre varios nodos al tener redundancia. Esto se logra porqué los datos se encuentran en otros nodos.

- ✓ Incrementa el rendimiento en la ejecución de procesos, debido a que emplea técnicas de procesamiento distribuido. Es decir cuenta con varios ordenadores disponibles para la ejecución de un problema.

- ✓ Compartición de dispositivos.

1.2.4 Desventajas de los Sistemas de Ficheros Distribuidos

Al igual que ventajas, existen inconvenientes que reducen el buen desempeño de estos sistemas, entre ellos están los siguientes:

- ✓ Uno de los inconvenientes presentados en estos sistemas en ocasiones es la latencia del envío de información

- ✓ En algunos casos la falta de replicación de datos, es decir, al mantener toda la información en un solo servidor de E/S en lugar de distribuirla por los diferentes servidores, puede provocar pérdida de información.

- ✓ Su escasa optimización en accesos concurrentes. Es decir, como tratar con el hecho de que pueda haber varios clientes trabajando sobre un mismo fichero o un fragmento de éste.

- ✓ Una desventaja para este tipo de sistemas es que al haber un único servidor de ficheros, es decir, un único punto de entrada, según aumenta el número de clientes que acceden al mismo, se produce un cuello de botella que impide su utilización de una manera adecuada. Dicho de otra forma, se trata de una estructura que para algunos sistemas de ficheros distribuidos no es escalable, por ejemplo el caso de NFSv1 [Russel85].

El incremento en prestaciones de los procesadores y de las redes de interconexión está aumentando el número de peticiones de E/S que reciben por unidad de tiempo los servidores de E/S en sistemas de ficheros distribuidos. Normalmente, aplicaciones que requieren de grandes cantidades de datos, por ejemplo, aplicaciones de cálculo, simulaciones, etc., se pueden ejecutar en sistemas distribuidos, sin embargo se provoca una carga excesiva de trabajo en los servidores de E/S. Es decir, se presenta un cuello de botella en los servidores de E/S que disminuye el rendimiento de los sistemas de cómputo. Este cuello de botella empeora en sistemas que ejecutan aplicaciones en las que todos los clientes acceden al mismo fichero.

Este problema es denominado por Patterson como crisis de Entrada/Salida (E/S) [Patterson88]. Una de las principales soluciones a este problema es:

- ✓ La implementación de paralelismo en los servidores de entrada/salida. Se logra, gracias a la implementación de sistemas de ficheros paralelos. Esto es debido a la distribución de los bloques de un fichero por los distintos servidores de E/S, permitiendo que se pueda acceder a todos los bloques o a grupos de ellos a la vez. Por consecuencia se reduce los tiempos de accesos a ficheros y mejora las prestaciones.

1.3 Sistemas de Ficheros Paralelos

En semejanza con los sistemas distribuidos, un sistema de ficheros paralelos está montado sobre un modelo cliente/servidor cuyo objetivo es trabajar de forma conjunta con el fin de optimizar el tiempo de acceso a los datos. La diferencia con respecto a los sistemas de ficheros distribuidos está en que, en los paralelos, los datos se distribuyen por todos los servidores de E/S; de esta forma permite acceder de manera paralela a distintos datos o a los mismo datos de un fichero. La figura 1.4 muestra la arquitectura de un sistema de ficheros paralelo. Como se ve puede ver en esta imagen, existe un gran número de servidores de entrada/salida, por lo que se produce un incremento en el ancho de banda, reduciendo así el tiempo de acceso a los datos. Como se mencionó anteriormente supone la resolución del cuello de botella que se presenta normalmente en los sistemas de ficheros distribuidos.

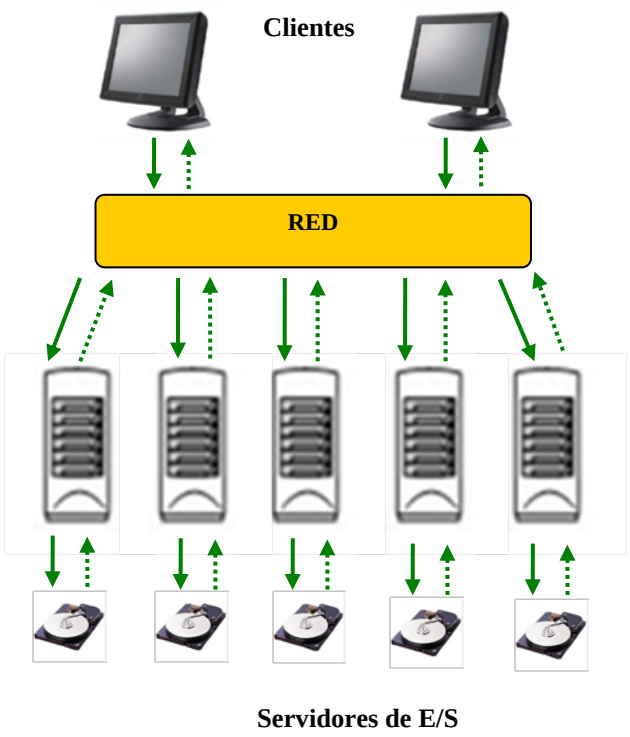


Figura 1. 4 Arquitectura de Sistema de fichero paralelo

1.3.1 Características los Sistemas de Ficheros Paralelos

Además de las características de los sistemas de ficheros distribuidos mencionadas en el Apartado 1.2.1 (Características de los Sistemas de Ficheros Distribuidos), los sistemas de ficheros paralelos, permiten ejecutar las aplicaciones paralelas con mayores prestaciones que los sistemas de ficheros distribuidos. Esto se hace mediante la distribución de un fichero entre los diferentes servidores de E/S, es decir, que puede repartirse incluso en diferentes trozos o fragmentos de un mismo fichero entre los servidores de E/S.

También hacen uso de interfaces de E/S paralela como es el caso de MPI-I/O [MPICH2] lo que contribuye al aumento del rendimiento de estos sistemas. Algunos ejemplos de sistemas de ficheros paralelos son descritos en el siguiente apartado.

1.3.2 Ejemplos de Sistemas de Ficheros Paralelos

Existe una gran cantidad de sistemas de ficheros paralelos. Entre ellos están:

PVFS (*Parallel Virtual File System*) [Ross02], Al igual que otros sistemas de ficheros su diseño se centra en el uso de clusters de computadores (grupo de ordenadores trabajando de manera conjunta para un fin en común), permite el acceso de operaciones de lectura y escritura concurrentes realizadas desde múltiples clientes a un fichero. Su distribución es como software libre y no se requiere de ningún hardware especial para que funcione. Es decir, trabaja sobre cluster heterogéneos. El capitulo 3 describe de manera más detallada las características y funcionamiento sobre PVFS.

GPFS (*General Parallel File System*)[Barkes98]. El objetivo principal en el diseño de GPFS, así como el de la mayoría de los sistemas de ficheros paralelos, es lograr un gran ancho de banda para acceder a ficheros desde cualquier cliente. GPFS está optimizado para los accesos secuenciales. Los ficheros se dividen en bloques iguales que se distribuyen por todos los discos logrando así un balanceo de carga. GPFS, soporta sin ningún problema los accesos

paralelos a los datos y metadatos de un fichero, ya que los datos son distribuidos por en los servidores de E/S a través de un modelo round-robin [Schmuck02].

Otros ejemplos de sistemas de ficheros paralelos son: Expand (*Expandable Parallel File System*) [García03], que es un sistema basado en servidores NFS, cuyo fin es crear una partición distribuida para almacenar los datos que han sido previamente divididos, es decir todos los ficheros en el sistema son distribuidos a través de todos los servidores NFS. Por último también están los sistemas de ficheros ParFySys [Carretero96] y Galley [Nils96].

1.3.3 Ventajas de los Sistemas de Ficheros Paralelos

Un sistema de ficheros paralelos además de contar con las ventajas mencionadas en los sistemas de ficheros distribuidos (Apartado 1.2.3), adicionalmente brinda:

- ✓ Mejora el tiempo de ejecución de las aplicaciones, disminuyendo el tiempo de acceso a los datos.
- ✓ Permite además de accesos secuenciales, accesos paralelos a los ficheros.
- ✓ Es altamente escalable, es decir cuanto mayor sea el número de servidores de E/S mejores son las prestaciones del sistema.

1.3.4 Desventajas de los sistemas de ficheros paralelos

En este caso también se pueden mencionar algunas de las desventajas de los sistemas de ficheros distribuidos. Por ejemplo:

- ✓ La carga continúa de peticiones en paralelo puede provocar cuellos de botella en los servidores E/S.

- ✓ La falta de memoria cache de datos. La mayoría de los sistemas no lo implementan por defecto ya que el mantenimiento de coherencia de los datos en

ocasiones resulta costoso y muy problemático, debido a que es necesario mantener los datos consistentes en las caches.

No obstante, una forma de resolver esta desventaja es la implementación de cache en los clientes. La utilización de la memoria caché puede almacenar en disco o memoria principal una copia de los datos más utilizados del sistema de ficheros, lo que permite que la ejecución de las operaciones de E/S se realice más rápidamente. Cabe destacar que el uso de cache ofrece ventajas las cuales se mencionan a continuación:

- ✓ Reduce la carga de trabajo al servidor.
- ✓ Disminuye el tráfico de la red.

El objetivo principal de esta memoria es analizar las distintas técnicas y métodos utilizados en la implementación de cache en entornos de sistemas de ficheros distribuidos y paralelos y proporcionar nuestra propia implementación de cache. En el Capítulo 2 se muestran algunas de las implementaciones de memoria cache en los clientes para sistemas de ficheros distribuidos y sistemas de ficheros paralelos.

La siguiente parte de este capítulo se dedica a describir de manera general los sistemas de ficheros distribuidos y paralelos, así como sus ventajas y desventajas al momento de ser implementados.

1.4 Diferencias de sistemas de ficheros Distribuidos y Paralelos

Normalmente los sistemas de ficheros distribuidos y paralelos tienen características muy similares ya que ambos acceden a datos que se encuentran distribuidos en una red; Sin embargo tienen propiedades que los diferencian, a continuación en la Tabla 1.1 se hace mención a algunas de estas diferencias:

Tabla 1.1 Diferencias de sistemas de ficheros Distribuidos y Paralelos.

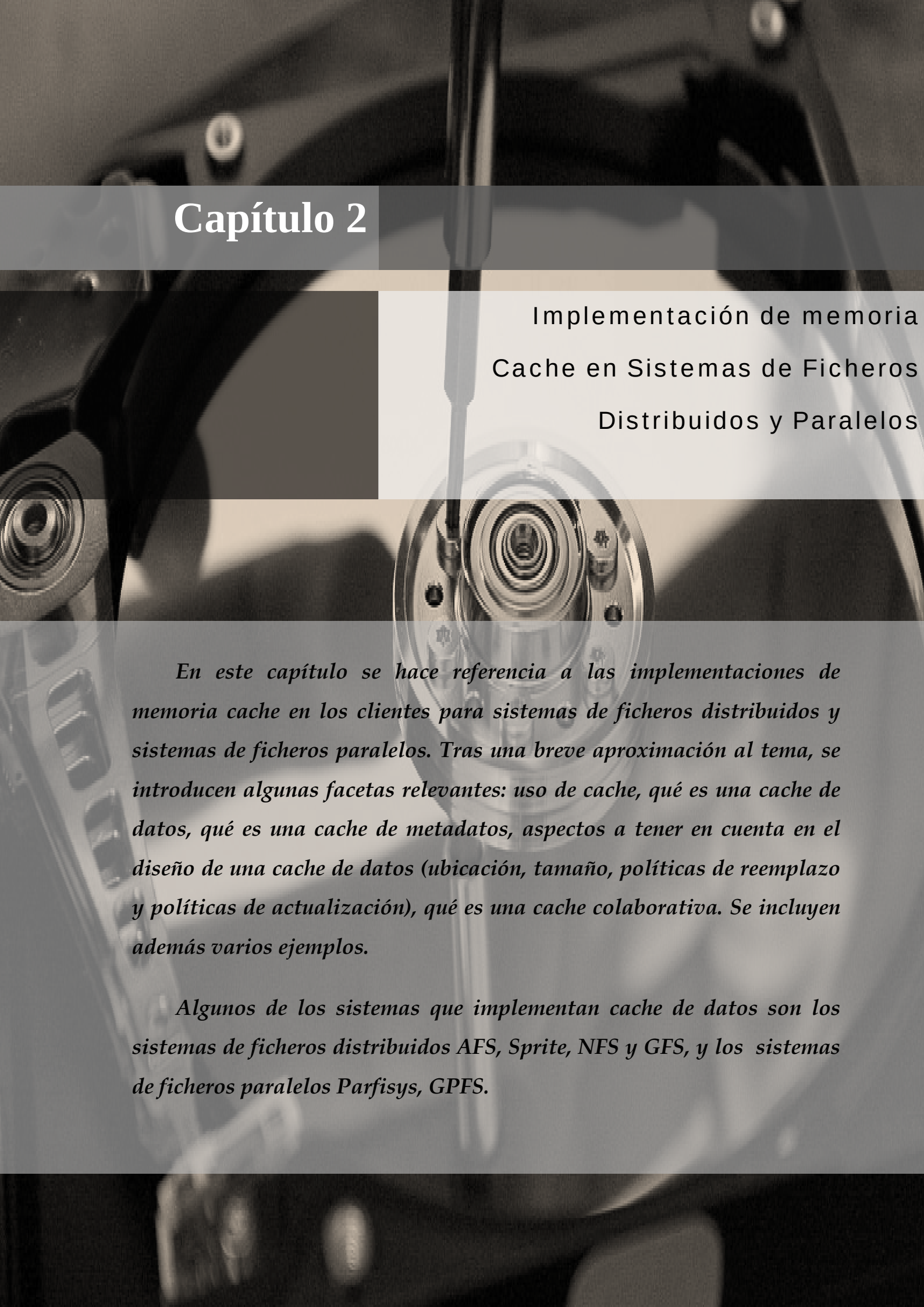
Sistemas de Ficheros Distribuidos	✓ El objetivo principal de los sistemas de ficheros distribuidos es permitir a múltiples usuarios trabajar de manera colaborativa en la resolución de problemas. ✓ Permite la ejecución de distintas aplicaciones a la vez. ✓ Normalmente los sistemas distribuidos cuentan con arquitectura heterogénea.
Sistemas de Ficheros Paralelos	✓ El objetivo principal de los sistemas paralelos es, poder alcanzar una velocidad mayor al momento de realizar una operación de E/S. Esto se realiza mediante accesos paralelos a los datos. ✓ Ejecutan una aplicación a la vez. ✓ Es más conveniente trabajar sobre arquitecturas homogéneas. Esto es porque de esta forma las características de los ordenadores pueden ser explotadas al máximo, otorgando mejores prestaciones. En caso contrario. ¿Qué sucede si los servidores de E/S tienen diferentes discos de almacenamiento? Puede presentarse el caso en el que uno de los discos llegue al máximo de su capacidad antes que otros; viéndose comprometidas las prestaciones del sistema por este detalle.

De esta manera concluimos el primer capítulo de este trabajo, con un breve bosquejo de lo que es un sistema de ficheros tanto para entornos distribuidos y paralelos. Además se presentan sus características principales, así como ventajas y desventajas. Como se ha mencionado anteriormente el siguiente capítulo trata sobre las implementaciones de memoria cache realizadas en entornos distribuidos y paralelos.

Capítulo 2

Implementación de memoria Cache en Sistemas de Ficheros Distribuidos y Paralelos

SUMARIO



Capítulo 2

Implementación de memoria Cache en Sistemas de Ficheros Distribuidos y Paralelos

En este capítulo se hace referencia a las implementaciones de memoria cache en los clientes para sistemas de ficheros distribuidos y sistemas de ficheros paralelos. Tras una breve aproximación al tema, se introducen algunas facetas relevantes: uso de cache, qué es una cache de datos, qué es una cache de metadatos, aspectos a tener en cuenta en el diseño de una cache de datos (ubicación, tamaño, políticas de reemplazo y políticas de actualización), qué es una cache colaborativa. Se incluyen además varios ejemplos.

Algunos de los sistemas que implementan cache de datos son los sistemas de ficheros distribuidos AFS, Sprite, NFS y GFS, y los sistemas de ficheros paralelos Parfisys, GPFS.

2.1 Introducción

Como se mencionó en el capítulo anterior, aplicaciones que requieren de grandes cantidades de datos, por ejemplo, aplicaciones de cálculo, simulaciones, etc., se ejecutan de manera más eficiente con:

- ✓ La implementación de paralelismo en los servidores de Entrada/Salida. Es decir, distribuyendo los datos entre los distintos servidores de E/S para que se pueda acceder en paralelo a los datos por parte de un cliente o de múltiples clientes.

- ✓ Otra propuesta es la utilización de memoria cache en los sistemas de ficheros. Esta última propuesta es la que analizaremos en este capítulo.

La implementación de cache en los clientes presenta las siguientes ventajas:

- ✓ Reducción de tiempos de accesos. Ya que los datos se pueden encontrar en la cache.

- ✓ Reducción del tráfico de la red que conecta los servidores o los discos a los clientes. Disminuye la necesidad de realizar transferencias de datos entre clientes y servidores de E/S.

- ✓ Reducción de la carga de los servidores de E/S. Disminuye la necesidad de realizar peticiones de bloques a servidores porque los puede encontrar en la cache.

El resto de este capítulo introduce algunos puntos importantes referentes al uso y tipos de cache, así como aspectos que debemos tomar en cuenta en el diseño de una cache de datos como, por ejemplo, ubicación, tamaño, políticas de reemplazo y políticas de actualización. Además se mencionan algunos ejemplos.

2.2 Cache en Sistemas de Ficheros Distribuidos y Paralelos.

Con este análisis se pretende dar una visión más detallada del uso de memoria cache en sistemas de ficheros distribuidos (SFD) y paralelos (SFP). Primero se describe el uso de cache en sistemas de ficheros distribuidos y paralelos así como sus características. Además, en esta sección se muestran algunos ejemplos de implementaciones de memoria cache para entornos distribuidos y paralelos.

2.2.1 Uso de cache en SFD y SFP

Como se ha mencionado anteriormente, para mejorar prestaciones en sistemas de ficheros distribuidos y paralelos se puede usar cache en los clientes.

Una memoria cache es un espacio de almacenamiento reservado que se encarga de guardar los datos más reciente en disco o memoria principal con la finalidad de que puedan ser accedidos posteriormente de manera rápida por las aplicaciones que hacen uso de la información. Estos datos son accedidos a través de copias que se encuentran almacenadas en las cache en lugar de acceder directamente a los dispositivos de almacenamiento de los servidores de E/S, por lo que se pretende que el incremento de prestaciones se vea incrementado en estos sistemas.

La figura 2.1 muestra la diferencia entre un sistema sin cache y con cache [García98]. La función T(x) representa el tiempo de acceso a los datos localizados en los dispositivos de almacenamiento y T´(x) representa el tiempo de accesos a los datos almacenados en cache. El uso de cache permite que T´(x) sea menor que T(x). La diferencia entre estos dos términos dependerá de la tasa de aciertos de la cache.

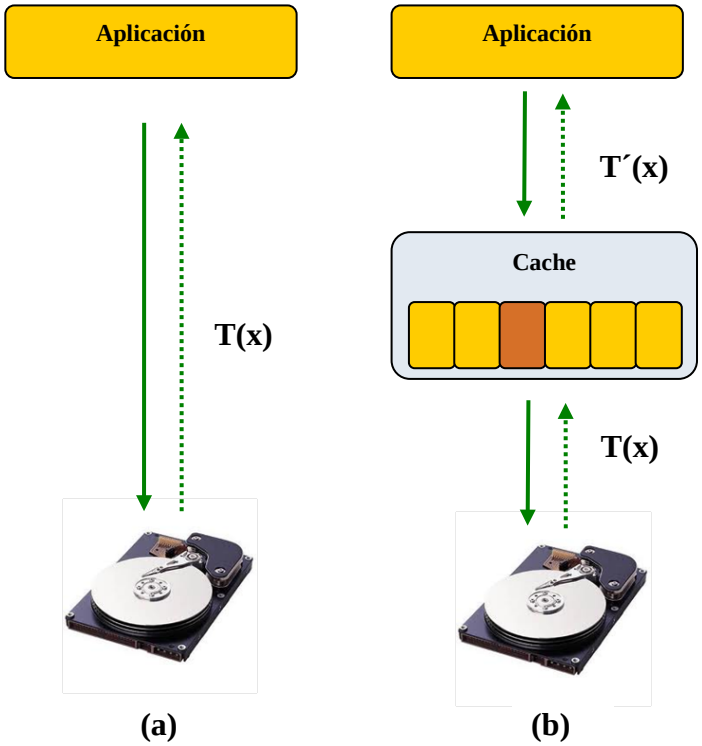


Figura 2. 1 Jerarquía de un sistema Sin Cache (a) y Con Cache(b)

Hay que destacar que el término de cache en sistemas de ficheros distribuidos o sistemas de ficheros paralelos hace referencia a una cache de datos o a una cache de metadatos. En el siguiente apartado se describe lo que es una cache de datos para sistemas de ficheros distribuidos y paralelos, definiendo algunos aspectos como la ubicación, tamaño y políticas de actualización, entre otras.

2.2.2 Cache de datos en SFD y SFP

La utilización de una cache de datos en sistemas de ficheros distribuidos y paralelos permite el almacenamiento de datos de un fichero, lo cual se traduce en un mayor rendimiento al momento de realizar un acceso a la información que ha sido previamente almacenada en cache. Esta mejora se logra debido a que:

1. Se aprovecha la localidad temporal y localidad espacial de los accesos. Generalmente en las aplicaciones se accede repetidamente a los mismos datos (*localidad temporal*) o se accede a datos cercanos a los que previamente se han accedido (*localidad espacial*).

2. Permite implementar lecturas adelantadas (*prefetching*) de datos. Consiste en aprovechar el acceso a los servidores para traer datos adicionales a los solicitados expresamente. Así, si estos datos son requeridos posteriormente por la aplicación, se accederán más rápidamente por estar en la cache y evitar realizar una petición a los servidores de E/S.

3. El empleo de políticas de escritura retardada. Este tipo de políticas permite hacer las modificaciones de un bloque en la cache en lugar de en los servidores de E/S, lo que reduce el número de acceso a estos servidores. El acceso a los servidores de E/S se puede retardar incluso hasta cuando se produzca el reemplazo del bloque por otro en la cache o se sincronicen los ficheros (fsync y sync).

Hay que mencionar que existen diversas facetas del diseño de las caches de datos que afectan a sus prestaciones como por ejemplo, la ubicación, su tamaño y las políticas de reemplazo y actualización. A continuación se describen cada una de estas facetas.

2.2.2.1 Ubicación de la cache

Dentro de un sistema de ficheros distribuido o paralelo una cache de datos puede ubicarse en el lado de los clientes y también en los servidores de E/S. Una cache de datos en los servidores, evita los problemas de mantener coherencia en los datos, sin embargo el sistema es menos escalable y existe un tráfico continuo en la red de interconexión.

A diferencia de la cache en los servidores de E/S, la cache de datos en el lado del cliente brinda una solución más efectiva, se evita tráfico en la red y los accesos, no sólo a los dispositivos de almacenamiento de los servidores de E/S, también a los propios servidores de E/S.

La cache de datos puede almacenarse en discos locales o en memoria principal. Algunas de las ventajas de almacenar la cache en discos son:

- ✓ Mayor capacidad de almacenamiento.

- ✓ Un alto grado de confiabilidad, es decir en el caso que existiera un bloqueo del sistema, no es necesario traer de nuevos los datos a la cache, ya que durante la recuperación los datos siguen estando almacenados.

Pero también tiene sus desventajas. Una de ellas es que supone mayor tiempo de acceso a datos.

Algunas de las ventajas de implementar cache en memoria principal son:

- ✓ la velocidad de procesamiento de la memoria es 5 o 10 veces más rápido que la de un disco. Por lo que permite disminuir el tiempo de espera de un bloque.

- ✓ Además que el implementar una cache en memoria principal, supone en el caso de los clientes, poder funcionar sin discos.

- ✓ Proveen suficiente porcentaje de acierto en lecturas, y se implementa un mismo mecanismo de cache tanto para los servidores como para los clientes.

Algunos ejemplos de ubicación de una cache de datos en sistemas de ficheros distribuidos y paralelos pueden verse en la tabla 2.1.

Tabla 2.1 Ubicación de cache en SFD y SFP.

Cache en Disco	✓ AFS [AFS84]. Cada cliente reserva un espacio en el disco para la cache, permitiendo almacenar los datos de forma temporal.
Cache en Memoria Principal	✓ Sprite [Ousterhout87] El objetivo principal de implementar cache en Sprite viene dado por la reducción de tráfico a los discos. Por lo cual implementa la cache en memoria principal [Nelson88]. ✓ NFS [Brados99], Principalmente el uso de cache en clientes se localiza en la memoria principal. ✓ ParFiSys [García04], hace uso de distintas cache en servidores y clientes las cuales almacenan los datos en memoria principal, con la finalidad de incrementa el rendimiento del sistema por medio de la optimización de operaciones de E/S. ✓ GPFS [Barkes98], La implementación de cache en el cliente se localiza en un espacio reservado de memoria principal en cada uno de los nodos.

2.2.2.2 Tamaño de la cache

El espacio reservado para una cache de datos en sistemas de ficheros distribuidos o paralelos dependerá de la frecuencia con la que acceden las aplicaciones a los datos. Es decir, para aplicaciones en donde la reutilización de datos es muy poco probable se puede definir un tamaño de cache pequeño [García98]. Para aplicaciones que acceden de manera repetida a los mismos datos puede ser necesario una cache de gran tamaño, para almacenar así mayor cantidad de datos que serán reutilizados. Si se cumplen los principios de localidad, cuanto mayor sea la cache mayor será la tasa de aciertos. Pero un gran tamaño es contraproducente si las aplicaciones no cumplen los principios de localidad porque reduce la memoria dedicada a la ejecución de las aplicaciones a consta de una cache que no va a tener aciertos. Por ello, es necesario conocer de manera adecuada el comportamiento de las aplicaciones para así determinar el tamaño óptimo de la cache.

Por ejemplo en GPFS [Schmuck02]. La cache se localiza en un espacio reservado de memoria, con un tamaño que es ajustado dinámicamente, dependiendo de la cantidad de memoria que se necesite; el rango es de 4MB a 512MB. Por defecto el tamaño definido es de 20MB, sin embargo normalmente este tamaño se establece alrededor de los 50MB.

2.2.2.3 Tamaño de bloques de la cache

Hay que indicar que una cache de datos, puede almacenar desde partes de un fichero (bloques) hasta el fichero completo. La tabla 2.2 muestra algunos sistemas de ficheros y el tamaño de bloque de cache que implementan para almacenar sus datos.

Tabla 2.2 Tamaño de bloque de cache en sistemas de ficheros distribuidos y paralelos

Tamaño Ficheros	✓ AFS-1, AFS-2 y Coda [Satyanarayanan89] trabajan con ficheros enteros.
Tamaño Bloques	✓ AFS-3 con bloques de 64 KB. ✓ NFS [Brados99], el tamaño de bloque de cache se define en 8KB almacenándose de forma local en la cache del cliente. ✓ Al igual que NFS, Parfisys [García04] y Sprite [Ousterhout87], emplean el almacenamiento en bloques. La forma en que Sprite organiza los bloques, es fijando un tamaño de 4 KB por bloque. ✓ En GPFS [Schmuck02], para minimizar la búsqueda de los bloques se define un tamaño de 256KB, sin embargo pueden configurarse entre 16KB y 1MB. Los bloques de mayor tamaño, permiten escribir o leer más datos en una simple operación, por lo que se reduce las solicitudes de bloques pequeños.

2.2.2.4 Politicas de reemplazo

Una cache de datos inicialmente se encuentra vacía. Conforme se van realizando accesos a los datos, éstos son almacenados en la cache hasta llenarla. Una vez que se llena la cache, se determina qué bloques deben ser eliminados cuando sea necesario liberar espacio para almacenar los nuevos bloques requeridos. La forma de liberar espacio se define a través de políticas de reemplazo:

- ✓ RANDOM (*Aleatoria*), reemplaza al azar los bloques de la cache.
- ✓ FIFO (*First In – First Out*), este tipo de políticas define que el primer bloque ingresado en la cache será el primer bloque en ser reemplazado.
- ✓ LRU(*Least Recently Used*), el menos recientemente usado, una manera de implementarlo es agregando un contador a cada bloque, el cual aumenta en 1 cada vez que se accede a cache y se reinicia solamente en el bloque que se hace referencia, se reemplaza el bloque que contenga el número mayor. Otra alternativa es por medio de listas enlazadas, en donde los bloques recientemente usados ocupan la posición final de la lista.
- ✓ MLRU (*Multiple LRU*) [carretero95], esta política es una variante de LRU, pero se ha modificado para incrementar su rendimiento. Este tipo de políticas se basa en la existencia de varias colas de bloques libres sobre cada una de las cuales se aplica una política de reemplazamiento LRU.

Parfisys [carretero95] implementa una política de reemplazo MLRU. En GPFS existen otras implementaciones de cache, un ejemplo de ello es una cache colaborativa del lado de los clientes, pero incluida dentro de la librería MPI-IO [Liao07]. Cuando se accede a un bloque que no está cacheado en ninguna parte, el proceso que ejecuta la petición tratará de cachear los datos localmente, por lo que leerá los datos desde el sistema de ficheros como si de una operación de lectura se trata. En caso de que no exista memoria suficiente se activa las políticas de reemplazo que se basan en reemplazar los datos menos recientemente usado (*LeastRecentlyUsed*).

2.2.2.5 Politicas de actualización

Para que una implementación de cache tenga un alto rendimiento y fiabilidad es necesario saber cómo y cuándo se deben actualizar los datos modificados a los dispositivos de almacenamiento, para ello existen las políticas de actualización que especifican cómo y cuándo los datos serán volcados a disco. Algunas políticas de actualización son:

- ✓ **Escritura inmediata** (*write – through*), este tipo de política consiste en que los datos se envían a disco en cuanto se modifican. Una de las ventajas de este tipo de escrituras es su fiabilidad ya que es poca la información que puede perderse en caso de que el sistema falle. No obstante su rendimiento es menor, debido a que hace uso constante de la red de interconexión y de los servidores de E/S en caso de que la cache se encuentre almacenada en los clientes.

- ✓ **Escritura retardada** (*write – back*), en este tipo de política los datos no se escriben de forma inmediata a disco, más bien los datos se mantienen en cache hasta que se presenta una política de reemplazo, es decir cuando se requieren bloques libres para nuevas peticiones. Retrasar la escritura a disco representa un mayor rendimiento, esto se logra porque las modificaciones se hacen solamente en cache, y se actualizan sólo en caso de ser necesario, de esta forma se evita el tráfico por la red. Sin embargo puede introducir un problema y es que si los datos cacheados se almacenan en memoria principal, estos pueden perderse si se presenta un fallo en el sistema, afectando así su fiabilidad.

Comparando estas dos políticas de actualización, en términos generales las escrituras retardadas brindan un mayor rendimiento, la probabilidad de que el sistema falle es pequeña, de esta manera se evita envíos incensarios de datos a los dispositivos de almacenamiento constantemente.

En NFS, como se comentó anteriormente, se trabaja con bloques de gran de tamaño, lo que indirectamente proporciona lecturas adelantadas, su política de actualización es escritura

inmediata. El cliente tiene que validar un bloque localizado en su cache, esto se hace mediante una comprobación en el servidor para saber si los atributos han cambiado. El tiempo de validación suele ser de 30 segundos.

Aunque NFS muestra una solución óptima, el intervalo de validaciones provoca demasiadas comprobaciones por lo que puede ser una carga de trabajo muy elevada para los servidores [García98]. Además se presenta el problema de accesos concurrente, situación que no se gestiona bien por este tipo de sistemas de ficheros.

Sprite, utiliza una política de escritura retardada con el fin de reducir el tráfico en la red [Nelson88]. Un cliente que ha modificado un bloque almacenado dentro de su cache puede volver a modificarlo antes de 30 segundos, lo que contribuye a reducir el tráfico de la red. Una vez pasado los 30 segundos después de la última modificación, los datos deben escribirse primero en la cache del servidor y posteriormente al disco. El intervalo de tiempo para realizar estas operaciones es de aproximadamente 30 a 60 segundos para cada una. Una ocasión más para devolver los datos a la cache del servidor antes de que transcurran los 30 segundos se presenta cuando el cliente cierra el fichero. Con este párrafo concluimos con las facetas que se deben tomar en cuenta en la elaboración de una cache de datos.

En el siguiente apartado se describe de manera breve la caches colaborativas, además de mencionar algunas implementaciones hechas dentro de sistemas de ficheros distribuidos y paralelos.

2.2.3 Cache Colaborativa en Sistemas de Ficheros

Hay que mencionar que otro tipo de cache de datos son las caches colaborativas o cooperativas. En caches colaborativas se comparten datos entre caches. Un sistema que implementa cache colaborativa puede incrementar el rendimiento de las operaciones de entrada/salida accediendo directamente a la cache de otro cliente en lugar de realizar una petición a los servidores de E/S. Por lo que un sistema de cache colaborativa [Dahlin94] se propone con la finalidad de reducir aún más la carga hacia los servidores E/S y lograr un alto rendimiento con la compartición de cache entre clientes. Añadir cache colaborativa supone

añadir un cuarto nivel a la jerarquía de memoria del sistema. Con cache colaborativa los datos pueden encontrarse, además de en la memoria local del cliente, en la del servidor o en disco, en la memoria local de otro cliente.

La figura 2.2 muestra la jerarquía entre un sistema tradicional con cache y un sistema con cache colaborativa. Al igual que en un sistema de cache tradicional, en la cache colaborativa se reduce la carga a los servidores de E/S. La escalabilidad del sistema puede incrementarse de acuerdo al número de clientes con los que se cuenten dentro del sistema, ya que, cuantos más clientes existan más memoria estará disponible para el almacenamiento de información, permitiendo tener en cache más datos por lo que se puede incrementar la probabilidad de aciertos en la cache.

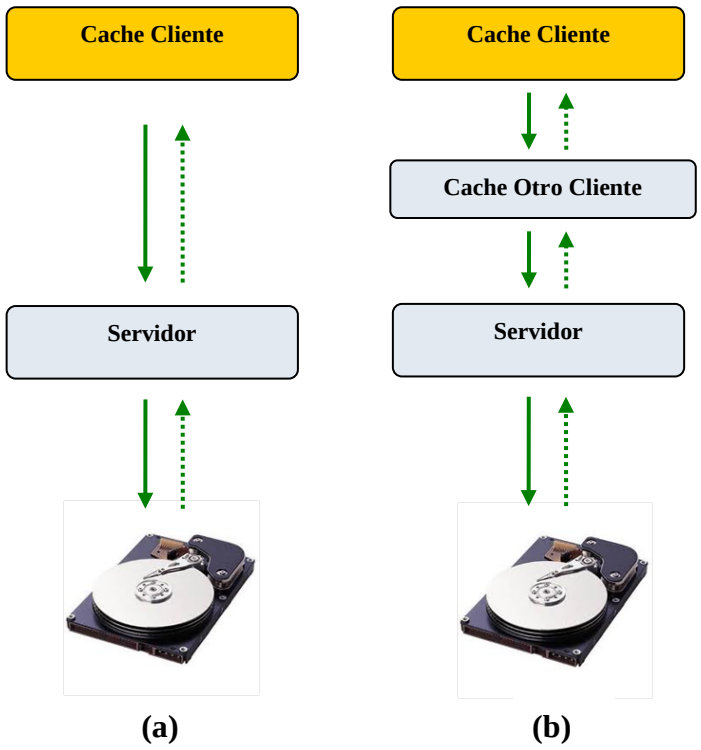


Figura 2. 2 Sistema tradicional con cache (a), Sistema con cache colaborativa (b) en sistemas de ficheros

Algunos ejemplos de cache colaborativa se han propuesto para los sistemas de ficheros distribuidos y paralelos, entre ellos los siguientes:

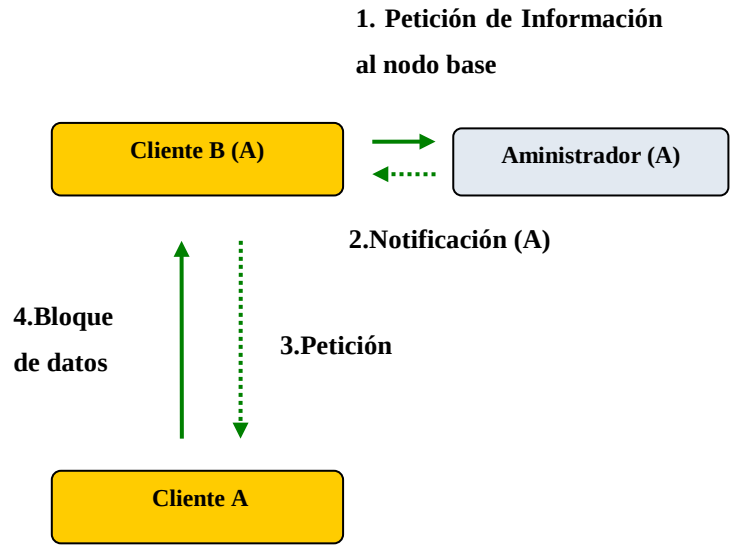
NFS-CD (*Network File System with Cluster Delegation*) [Batsakis08], es una extensión de NFSv4 [Pawlowski00]. NFS-CD implementa una cache colaborativa en la cual se delega un fichero a un grupo de clientes (*Cluster Delegation*) para que puedan leer y escribir en él sin tener que acceder al servidor. En NFS-CD se extiende el concepto de delegación de NFSv4 permitiendo que un fichero pueda ser delegado a un conjunto de clientes. Para conseguir esto lo que hace es que la delegación se pasa de un cliente a otro del conjunto. Posteriormente, un nodo adquiere la delegación del fichero (*delegation holder*) y serializara los accesos al fichero delegado, se podría decir por este motivo que actúa como pseudo-servidor.

En el caso de una petición de lectura, cuando un cliente solicita acceso a un fichero, el delegado verifica los bloques que pueden ser atendidos por la cache. Si alguno de estos bloques se encuentra en más de una cache de cliente, el delegado devuelve la ubicación de los clientes que mantienen una copia en su cache de dicho bloque, y uno de ellos es seleccionado aleatoriamente por el lector.

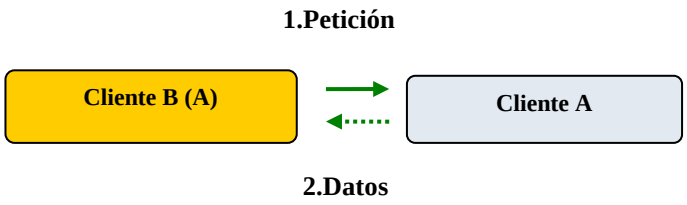
Para el caso de las escrituras, NFS-CD brinda una operación de confirmación (*fast-commit*) rápida lo que evita la necesidad de implementar escrituras síncronas de clientes en un servidor de E/S. En lugar de esto, los datos se escriben en memoria de varios clientes y aunque las copias que se hacen de cliente a cliente sean síncronas, son rápidas ya que no conlleva la escritura a disco. La escritura implementada mejora el rendimiento de escritura y la fiabilidad de los datos en la mayoría de los casos.

C2Cfs [Ermolinskiy09], como NFS-CD, también emplea un modelo de caches de datos en los clientes para compartir los ficheros y propaga las actualizaciones de cliente a cliente. Igualmente, como NFS-CD, la cache se ha implementado en NFSv4 [Pawlowski00] sin necesidad de realizar ningún cambio en los servidores y manteniendo el mismo modelo de consistencia (el modelo *close-to-open*). Este modelo de consistencia garantiza que cuando un cliente abre un fichero las actualizaciones previas del mismo son visibles correctamente en dicho cliente. Estos datos a diferencia de NFS-CD que almacena los datos en la memoria local, se almacenan en los discos de cada cliente.

Otra implementación de cache colaborativa es la que se propone en [Hwang05]. En la implementación ***Home-Based Cooperative Cache*** [Hwang05], cada bloque tiene un nodo base (*Home Node*). Este nodo base es estático, es decir, nunca cambia. Un cliente debe obtener del servidor el identificador del nodo home de un bloque sólo una vez, (cuando accede por primera vez al bloque) y entonces puede continuamente acceder al bloque a través del nodo home del bloque. Véase las figuras 2.3.



(a) Accediendo a los datos desde otro cliente distinto al nodo base cuando este cliente no tienen información sobre el nodo base



(b) Accediendo a los datos desde otro cliente distinto al nodo base cuando este clientetienen información sobre el nodo base

Figura 2. 3 Accediendo a un bloque en Home-Based Cooperative Cache

En esta implementación de cache colaborativa las peticiones de un cliente de lectura/escritura de un bloque que no tiene cacheado se dirigen al nodo base (*Home-Node*) del bloque, que es quien tiene información sobre el estado del bloque en el sistema (quién tiene copias, si se ha modificado, etc.). Si una petición de escritura de un bloque llega al nodo base, éste inválida las copias de ese bloque que se encuentran en la cache de otros clientes antes de modificarlo. Haciendo esto se consigue mantener la consistencia de los datos en las caches, por lo que se diferencia de las implementaciones de cache colaborativa mencionadas en párrafos anteriores que usan un modelo *close-to-open.*

Se ha concluido con la breve descripción sobre caches colaborativas, en el siguiente apartado se hace mención otro tipo de cache dentro de los sistemas de ficheros distribuidos y paralelos, y se le conoce como cache de metadatos.

2.2.4 Cache de metadatos

Los metadatos se definen como datos que describen de forma global la información, el contenido y otras características de los datos. Es decir, es información sobre información o datos sobre datos.

Los metadatos en un sistema de ficheros pueden ser persistentes o transitorios. Un ejemplo de metadatos persistentes son los que describen los bloques de disco empleados para almacenar los datos de un fichero. Los segundos se utilizan solo cuando el fichero es accedido, por ejemplo una marca de tiempo de cuando fue accedido el fichero por última vez. Una cache de metadatos en sistemas de ficheros se encarga de contener toda la información necesaria que describe la estructura de un ficheros de manera global, por ejemplo marcas de tiempo, tamaño, permisos, nombre, atributos, direcciones, etc.

Como la cache de datos, la cache de metadatos puede almacenarse en memoria principal para tener accesos más rápidos e incrementar así el rendimiento en los sistemas de ficheros. Puede haber cache de metadatos tanto en los clientes como en los servidores de E/S. Como se menciono anteriormente una cache de datos en los clientes reduce el tráfico por la red, el mismo beneficio se obtiene con la cache de metadatos en los clientes.

Existen diferentes tipos de cache de metadatos:

- ✓ **Cache de atributos:** Se encarga de almacenar los atributos del fichero, por ejemplo, el tamaño, permisos, accesos, etc.

- ✓ **Cache de nombres:** La utilización de una cache de nombres de ficheros facilita la localización de un fichero por su nombre. Permite reducir el envío de mensajes a los servidores de E/S, debido a que la mayoría de las peticiones pueden ser atendidas desde la cache, que proporcionará la última situación conocida del fichero.

- ✓ **Cache de particiones:** Este tipo de cache se emplea para obtener accesos más rápidos a la información de particiones.

- ✓ **Cache de información de direccionamiento:** Almacena la información necesaria para acceder a los bloques que conforman un fichero.

Los dos primeros tipos de cache de metadatos se utiliza con mayor frecuencia. Algunos ejemplos de sistemas de ficheros que implementan cache de metadatos se mencionan a continuación:

GFS [Google03] almacena 3 tipos de metadatos: el espacio de nombres de los bloques y ficheros; el mapeo de los ficheros a los bloques; y la localización de cada réplica de un bloque. Todos los metadatos se almacenan en memoria para que las operaciones sean rápidas.

En NFS [Brados99], al igual que se implementa una cache de datos, también se utiliza una cache de atributos (i-nodos). En esta cache se almacenan los atributos de los ficheros usados recientemente.

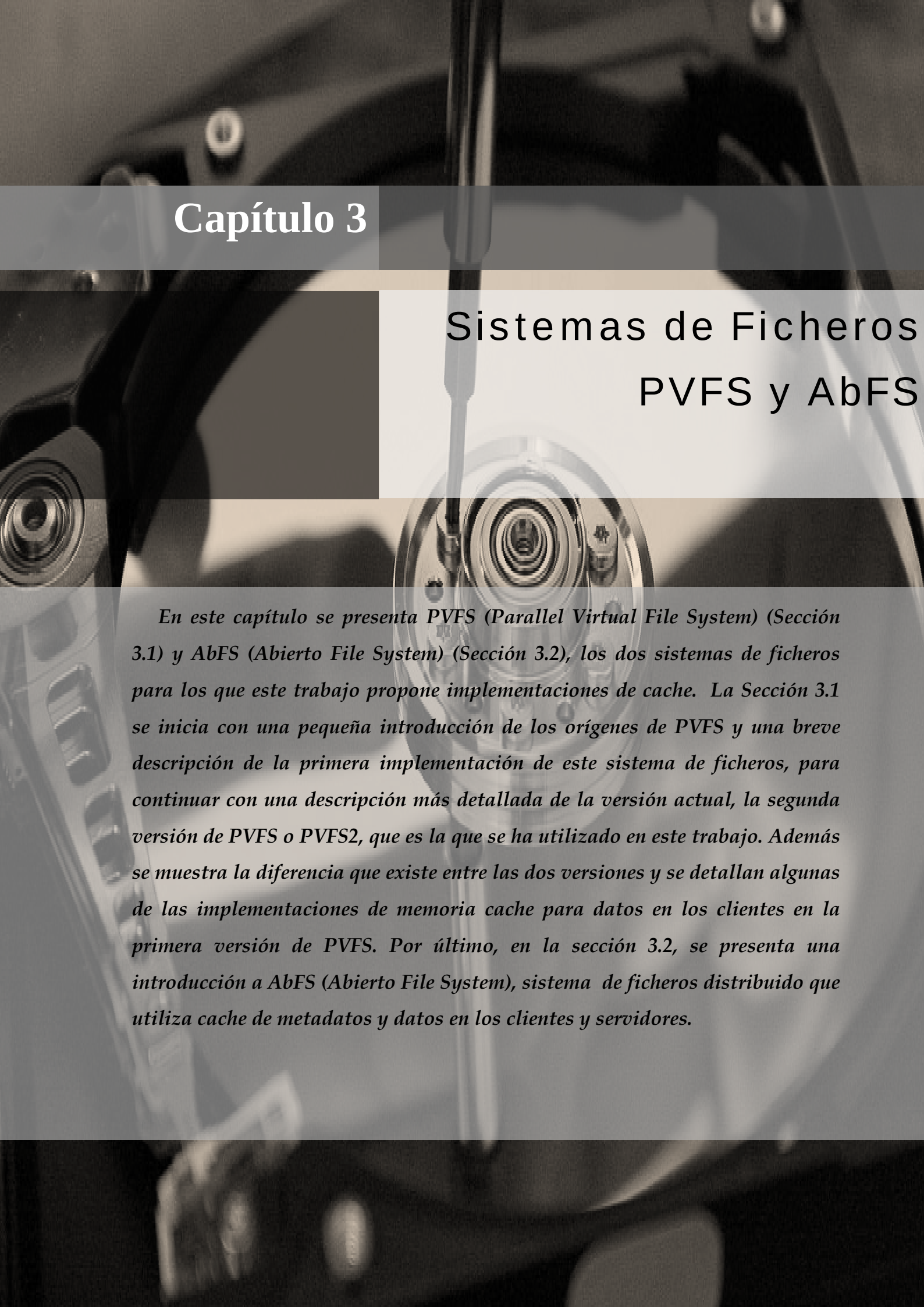
De esta forma se ha concluido el estudio de los sistemas de ficheros para entornos distribuidos y paralelos que hacen uso de la implementación de cache con el fin de mejorar las prestaciones de los sistemas.

En el siguiente capítulo se centra en el sistema de ficheros paralelo PVFS en sus diferentes versiones, características, y distintas modificaciones realizadas de este sistema de ficheros, lo que nos permitirá tener una visión más amplia en referencia a PVFS y la propuesta que se desea implementar. Así como una introducción al sistema de ficheros de AbFS.

Capítulo 3

Sistemas de Ficheros PVFS y AbFS

SUMARIO



Capítulo 3

Sistemas de Ficheros PVFS y AbFS

En este capítulo se presenta PVFS (Parallel Virtual File System) (Sección 3.1) y AbFS (Abierto File System) (Sección 3.2), los dos sistemas de ficheros para los que este trabajo propone implementaciones de cache. La Sección 3.1 se inicia con una pequeña introducción de los orígenes de PVFS y una breve descripción de la primera implementación de este sistema de ficheros, para continuar con una descripción más detallada de la versión actual, la segunda versión de PVFS o PVFS2, que es la que se ha utilizado en este trabajo. Además se muestra la diferencia que existe entre las dos versiones y se detallan algunas de las implementaciones de memoria cache para datos en los clientes en la primera versión de PVFS. Por último, en la sección 3.2, se presenta una introducción a AbFS (Abierto File System), sistema de ficheros distribuido que utiliza cache de metadatos y datos en los clientes y servidores.

3.1 PVFS

PVFS (*Parallel Virtual File System*) [Ross02] es un sistema de ficheros distribuido desarrollado para trabajar en clusters de computadores. Se ha desarrollado en la Universidad de Clemson por Walt Ligon y Blumer Eric en 1993 [Carns00]. La versión 0 de PVFS se basó en Vesta, un sistema de ficheros paralelo desarrollado en el centro de investigación IBM T.J Watson [Corbett96].

Robert Ross re-escribió el código de PVFS en el año 1994 para que pudiese ser usado en redes TCP/IP. Esta versión se distanció del diseño original basado en Vesta. Ese mismo año en una reunión entre Ligon, Tomas Sterling y John Dorband en el centro de vuelos espaciales Goddard (GSFC, por sus siglas en inglés) se acordó la construcción del primer computador Beowulf [Ligon99], así como también que PVFS pudiese ser soportado en las distribuciones de Linux para que pudiese ser incorporado en la nueva máquina (computador Beowulf). Fue hasta 1997 cuando en una reunión del grupo se pidió que PVFS se lanzara como un sistema de código abierto y funcionara sobre las distribuciones de Linux. Desde ese año hasta la fecha se han unido al grupo de desarrollo otros científicos como Phill Carns, Witchurch Dale, Harish Ramachandran, Neil Miller, Lathrum Rob y Pete Wyckoff.

En 1999 se creó PVFS2000, al cual más tarde se le conocería como PVFS2. PVFS2 es un diseño e implementación totalmente independiente de la primera versión que trata varias limitaciones del proyecto original. Este nuevo sistema de ficheros fue liberado en el año 2003. Su nuevo diseño (distribución de metadatos, integración con MPI, soporte de múltiples tipos de redes, entre otras características) facilita la experimentación y obtención de resultados. PVFS2 se ha convertido en una herramienta estándar para la comunidad científica que aborda la implementación de aplicaciones de altas prestaciones, y ha fomentado una gran variedad de investigaciones relacionadas. La primera versión fue retirada en 2005. PVFS2 se mantiene operativo y recibe soporte por parte del grupo de desarrollo.

La decisión de usar PVFS2 como una de las herramientas de partida en este trabajo se debe a que es un software libre ampliamente utilizado por parte de la comunidad científica

que, como su antecesor, no incluye cache para datos en clientes. Una de las aportaciones de este trabajo es el diseño e implementación de una memoria cache de datos en los clientes de PVFS2.

3.1.1 Primera versión de PVFS

El sistema de ficheros PVFS [Ligon99, Carns00, Ligon03] ofrece almacenamiento y recuperación de datos para aplicaciones paralelas y secuenciales en entornos cluster. Se denomina sistema de ficheros paralelo porque permite que múltiples clientes puedan acceder a la vez al mismo fichero y a distintos ficheros. Las siguientes características de PVFS son importantes debido a que son parte fundamental en el diseño del sistema y su buen funcionamiento:

- ✓ Es un sistema de código abierto [carns00]. Liberado bajo la licencia de GNU (*General Public License*).

- ✓ Permite la distribución física de los datos sobre los discos del cluster.

- ✓ Ofrece un alto rendimiento en operaciones concurrentes de lectura/escritura. Permite acceso paralelo a los datos para las aplicaciones que se están ejecutando y utilizan este sistema de ficheros.

- ✓ Implementa múltiples interfaces (POSIX, MPI-IO).

- ✓ Escalabilidad. Permite múltiples servidores de Entrada/Salida, por tanto el espacio de almacenamiento se incrementa.

- ✓ Fácil instalación.

Además, PVFS se puede montar en todos los nodos del cluster bajo el mismo directorio o bien en directorios distintos. Una vez montado, todos los nodos podrán acceder a los ficheros almacenados en esos directorios.

Una manera de poder incrementar el rendimiento de los sistemas es poder distribuir los datos (*striping*) dentro de los distintos discos que conforman el cluster y que se encuentran repartidos por los distintos nodos. Con esta distribución de datos, múltiples clientes podrán acceder de forma paralela a estos datos evitando así cuellos de botellas en los nodos de E/S (servidores E/S). PVFS hace esta distribución y por este motivo es un sistema de ficheros paralelo. PVFS permite distribuir los datos entre los nodos de E/S utilizando un modelo *round-robin.* Un nodo en PVFS puede realizar varias funciones [Carns00], estas funciones son:

- ✓ Servidor de Metadatos. El servidor de metadatos se encarga de mantener los atributos de los ficheros y directorios, tales como permisos, propietarios y localización de los datos. Algunas de las operaciones que se pueden realizar en un servidor de metadatos son: crear, eliminar, abrir o cerrar algún fichero. Esto se logra, mediante la comunicación del cliente con el servidor de metadatos a través de la librería `libpvfs`

- ✓ Servidores de E/S. Los servidores de E/S se encargan de almacenar y gestionar los accesos a los datos localizados en los directorios de PVFS

- ✓ Clientes. A través de los clientes acceden los usuarios a los datos almacenados en los directorios de PVFS.

Otras operaciones que PVFS permite realizar son: leer y escribir datos. Estas operaciones son llevadas a cabo directamente en los servidores E/S a través de la librería `libpvfs.` En este caso no hay comunicación con el servidor de metadatos. La Figura 3.1 se muestra los componentes que conforman PVFS. Hay que mencionar que, en la primera versión

de PVFS, existe un solo servidor de metadatos, uno o varios servidores de entrada/salida, y uno o múltiples clientes.

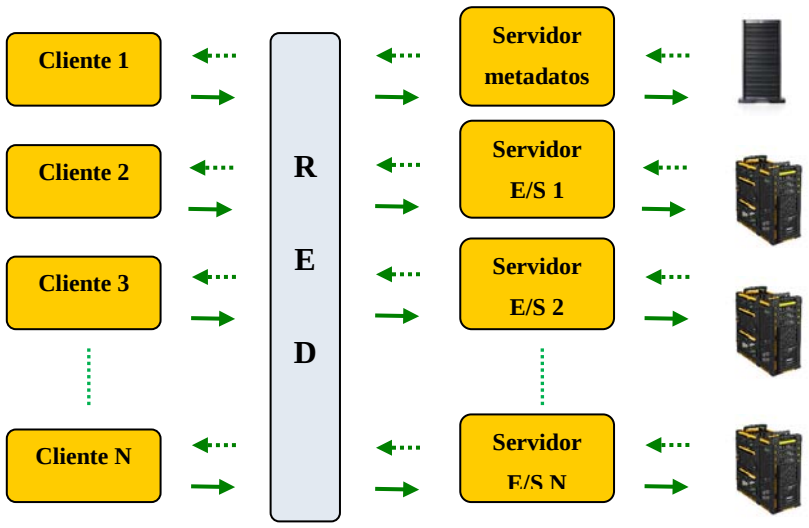


Figura 3. 1 Componentes de PVFS

3.1.2 Segunda versión de PVFS: PVFS2

PVFS2 [PVFS2, Ross02, Ramachandran02], al igual que su antecesor, es un sistema de código abierto (*Open Source*), característica que ha permitido sea uno de los sistemas favoritos para los desarrolladores de aplicaciones de entrada/salida paralela. Está diseñado sobre un modelo cliente/servidor que soporta uno o múltiples clientes así como un gran número de servidores. Estos servidores pueden ser servidores de metadatos, servidores de E/S, o incluso ambos a la vez.

Esta nueva versión implementa características (además de las mencionadas en la primera versión), que mejoran aun más la eficiencia del sistema. Estas características son mencionadas a continuación:

- ✓ Soporte modular para redes (BMI) y almacenamiento (Trove).

- ✓ Formato de peticiones muy potente para el acceso a datos no contiguos
- ✓ Flexibilidad para implementar la distribución de datos.
- ✓ Permite combinar un servidor de metadatos y un servidor de E/S en un mismo nodo.
- ✓ Múltiples servidores de metadatos (recientemente incorpora esta funcionalidad en la nueva versión del sistema que lleva como nombre OrangeFS [OrangeFS12]. Hasta hace poco tiempo solo podía tener un servidor de metadatos).
- ✓ Mejora el acceso a los datos para los clientes a través de MPI-IO [Mpich2].
- ✓ Soporte para redundancia de datos y de metadatos. (fase de investigación)
- ✓ Permite trabajar con cluster heterogéneos.

Otra característica es que PVFS2 es un sistema de ficheros sin estados, por lo que no implementa bloqueos en las interacciones cliente/servidor. También evita implementar cache para datos en el lado de los clientes, con el fin de reducir el coste que origina el mantenimiento de coherencia que conllevaría transferencias adicionales entre nodos para bloquear accesos a datos. No obstante, PVFS2 implementa en clientes caches de metadatos: acache (*atribute cache*) y ncache (*name cache*). La primera de ellas hace referencia a los atributos del fichero y la segunda es una cache de nombres, este tipo de cache se han empleado con el fin de minimizar las iteraciones al o los servidores de metadatos.

3.1.2.1 Componentes del sistema de ficheros PVFS2

La figura 3.2 muestra los componentes de PVFS2. A diferencia de PVFS, en PVFS2 se puede combinar un servidor de metadatos y un servidor de E/S en un mismo nodo, así como mantener varios servidores de metadatos.

.

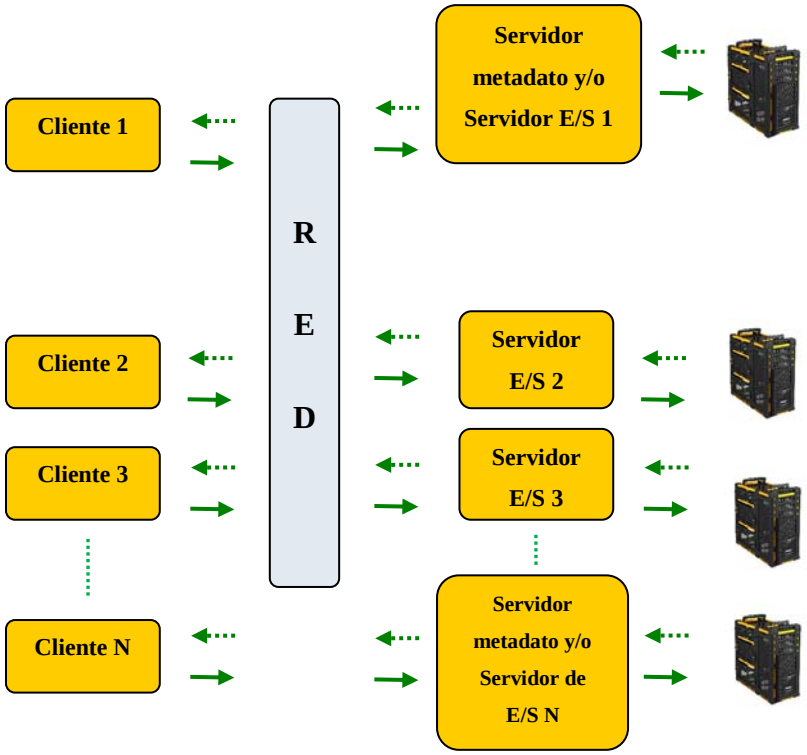


Figura 3. 2 Componentes de PVFS2

A continuación algunos conceptos o términos que debemos conocer para tener un mejor entendimiento de este sistema son descritos en los siguientes apartados, en los cuales se hace mención a los componentes y objetos del sistema de ficheros PVFS2.

3.1.2.1.1 Servidores PVFS2

Los servidores dentro de PVFS2 mantienen el espacio de almacenamiento para la información manejada en el sistema. Como se menciono más arriba los servidores pueden ser categorizados en servidores de metadatos y servidores de E/S. Los servidores ejecutan un demonio denominado *pvfs2-server*.

Un servidor de metadatos en PVFS2 además de mantener permisos, propietarios y localización de los datos, almacena marcas de tiempo y tamaños de fichero

Los servidores de E/S almacenan los datos. Los ficheros se dividen en bloques que se distribuyen por defecto de manera *Round-Robin* entre los distintos servidores de E/S. El tamaño de los bloques por defecto es de 64 KB. La figura 3.3 muestra un ejemplo de cómo se modifica este valor, para este caso el tamaño del bloque a distribuir se ha cambiado a 128KB (131072 Bytes).

```
[root@wlan1 ~]#setfattr -n user.pvfs2.dist_params -v
strip_size:131072 /mnt/pvfs2/dir

    setfattr, permite configurar los atributos de los
objetos del sistema de ficheros. En donde:

    -n, hace referencia al nombre del atributo a
configurar.

    -v, especifica el nuevo valor a asignar.

    Por último se indica la ruta o el directorio donde
se desea hacer esta modificación.
```

Figura 3. 3 Modificación del tamaño del bloque en la distribución de PVFS2

3.1.2.1.2 Cliente PVFS2

Los clientes, ejecutan un demonio conocido como *pvfs2-client*, que se encarga de la comunicación con los servidores de PVFS2. Inicialmente convierte las operaciones del sistema virtual de ficheros (VFS) en operaciones dentro de la interfaz del sistema de PVFS2. Para poder acceder al sistema de ficheros PVFS2, cada cliente debe lanzar el demonio *pvfs2-client*.

3.1.2.1.3 Colección (Collection)

Una colección en PVFS2 es un sistema de ficheros que se distribuye entre los servidores de PVFS2 disponibles. Los servidores pueden mantener múltiples colecciones, cada colección tiene asociado un identificador único (*File System IDs*). Para acceder a una colección, un

cliente requiere de un fichero de configuración similar en sintaxis al fichero `/etc/fstab`. El fichero `fstab` (*file systems table*) se encuentra en sistemas Unix (dentro del directorio `/etc/`) como parte de la configuración del sistema. Lo más destacado de este fichero es la lista de discos y particiones disponibles. En ella se indica cómo montar cada dispositivo y qué configuración utilizar. PVFS2 utiliza su propio fichero de configuración llamado `pvfs2tab`, el cual hace referencia a la configuración y el directorio en la que debe montarse el sistema de ficheros.

3.1.2.2 Objetos del sistema de ficheros PVFS2

PVFS2 tiene cuatro tipos de objetos que son visibles para los usuarios: ficheros de metadatos (*metafile*), ficheros de datos (*datafile*), directorios (*directory*) y enlaces simbólicos (*symbolic link*). En la figura 3.4 se puede visualizar los objetos de PVFS2. A continuación se definen cada uno de los objetos del sistema de ficheros PVFS2.

1. **Metafile.** Los metafile son metadatos que se describen como un conjunto de información que hace referencia a los datos de un fichero. Esta información contiene identificadores de usuario (`uid` o `user id`), identificadores de grupo (`gid` o `group id`), permisos, marcas de tiempo y parámetros de la distribución. La interfaz de sistema permite acceder o modificar esta información a través de las funciones `getattr` y `setattr`.

2. **Datafile.** Un datafile se refiere a los datos que componen un fichero. Es decir un datafile en PVFS2 contiene un fichero o parte del mismo y puede ser leído, escrito, modificado e incluso, de ser necesario, eliminado de forma individual. El número de datafile para un fichero depende del tamaño del fichero y la unidad de *striping* que se emplee al momento de distribuir los datos por los distintos servidores de E/S. La forma en que los datafile son colocados en los servidores de E/S es decidido por la distribución usada.

3. **Directorios.** Un directorio es un fichero que contiene las entradas de directorio [Ramachandran02]. Las operaciones de directorio soportadas por la interfaz de sistema son crear, leer y eliminar un directorio. Además, las operaciones de búsqueda

para obtener el `handle` del directorio y las funciones `getattr` y `setattr` para leer/modificar los atributos pueden también actuar sobre directorios.

4. **Enlaces Simbólicos (*symbolic links*).**Un enlace simbólico a un objeto es un fichero especial que apunta a otro fichero o directorio que puede tener una ubicación distinta al apuntador. Una ventaja de los enlaces simbólicos es que si este es eliminado, el fichero al que apunta no sufre ningún cambio. En PVFS2 el comando usado para crear un enlace simbólico es `pvfs-ln`. Este objeto no se muestra en la Figura 3.4.

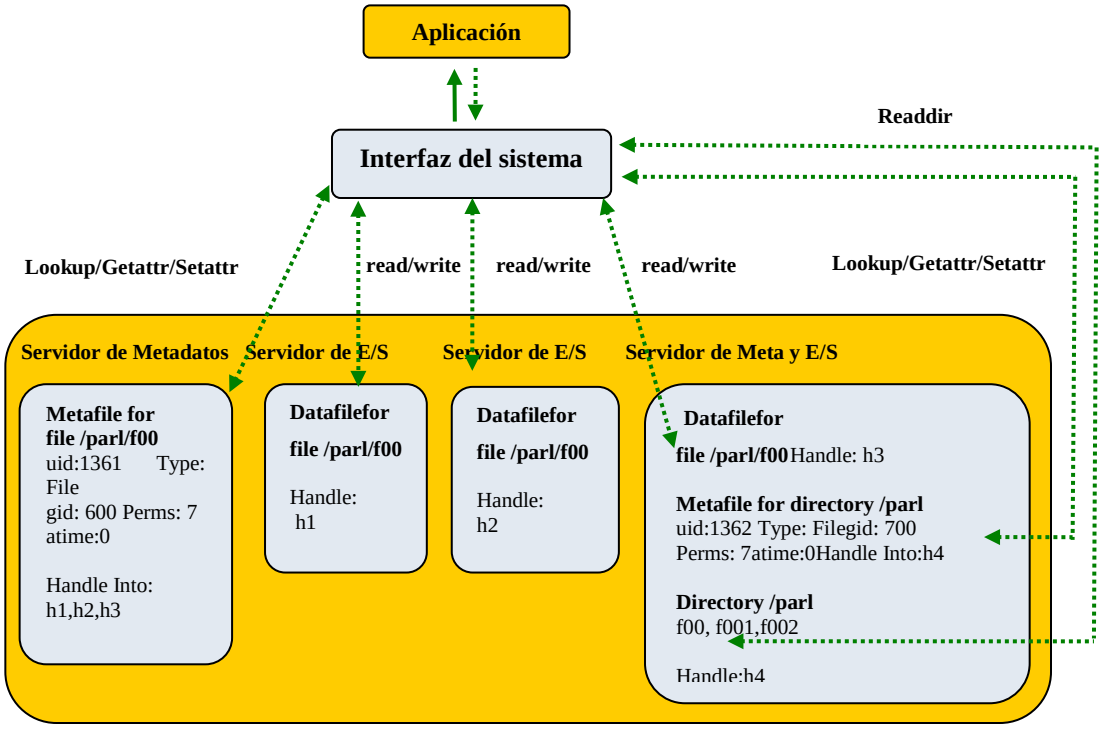


Figura 3. 4 Objetos del sistema de ficheros PVFS2

3.1.2.3 Handle

Un handle es un identificador único para cada objeto almacenado dentro de PVFS2, es como el inodo en ext3. Cada fichero, directorio y enlace simbólico tienen un handle. Por ejemplo, en la figura 3.5 se puede observar el rango de handles asignados para los metadatos y datos dentro del sistema de ficheros. Estos valores son establecidos una vez que se crea la

configuración de PVFS2, normalmente esta configuración se localiza en un fichero en `/etc/pvfs2-fs.conf`.

```
[root@wlan1 ~]#more /etc/pvfs2-fs.conf


   …..
   …..

<Filesystem>
        Name pvfs2-fs
        ID 1878670356
RootHandle 1048576
<MetaHandleRanges>
           Range wlan1 3-3074457345618258603
</MetaHandleRanges>
<DataHandleRanges>
           Range wlan1 3074457345618258604-6148914691236517204
           Range wlan4 6148914691236517205-9223372036854775805
</DataHandleRanges>
```

Figura 3. 5 Rango de handles para los metadatos y datos en PVFS2

3.1.2.4 Arquitectura de PVFS2

En la figura 3.6 se muestra la arquitectura de PVFS2 [Kunkel07]. La figura ilustra que PVFS2 está compuesto de un modelo cliente/servidor. Los clientes requieren pasar por distintas capas o niveles antes de poder leer, escribir, modificar o eliminar datos en los servidores de E/S. A continuación se describen cada una de estas capas.

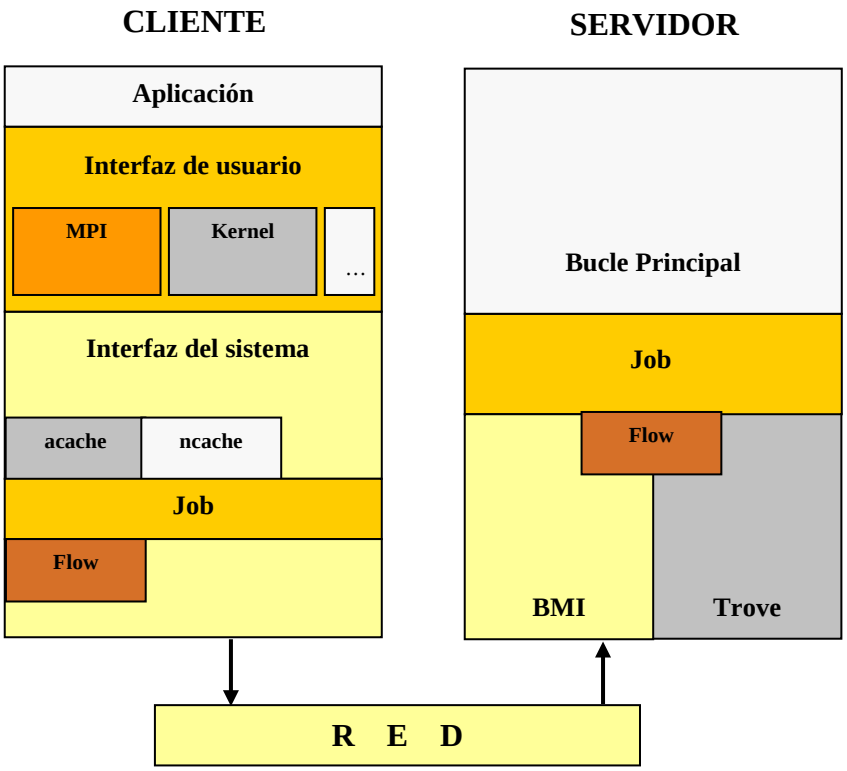


Figura 3. 6 Arquitectura de PVFS2

3.1.2.4.1 Interfaz de usuario (UserLevel Interface)

PVFS2 soporta varias interfaces a nivel de usuario. Algunas de estas interfaces disponibles son: la interfaz de kernel [kernel] o núcleo y la interfaz de MPI-IO. La interfaz del kernel se ha implementado con un módulo de PVFS2 integrado dentro del sistema virtual de ficheros (*Virtual File System*) y un proceso o demonio (`pvfs2-client`) en el espacio de usuario que se comunica con los servidores de E/S. PVFS2 fue especialmente diseñado para tener una gran integración con cualquier implementación desarrollada en la interfaz de paso de mensajes MPI-IO, que es una interfaz estándar que ofrece altos rendimientos.

3.1.2.4.2 Interfaz de sistema (System Interface)

La interfaz de sistema proporciona una API para la manipulación de los objetos del sistema. Las aplicaciones hacen uso de las funciones `libpvfs` para acceder al sistema de ficheros PVFS2, cualquier aplicación que haga uso de esta interfaz se considera un cliente. Al invocar estas funciones se ejecuta una máquina de estados que procesa las operaciones en pasos más pequeños.

La interfaz del sistema se caracteriza por permitir a las aplicaciones acceder al sistema de ficheros y por negociar con la capa de abstracción de red (Job). Los objetivos que principalmente se han perseguido a la hora de diseñar esta capa son:

- ✓ Abstracción
- ✓ Flexibilidad
- ✓ Soporte para interfaz a nivel usuario
- ✓ Robustez
- ✓ Rendimiento

En el apartado 3.3 se comentó que PVFS2 implementa dos tipos de caches en el lado del cliente. Estas caches se localizan en la capa de interfaz del sistema (*System Interface*) (véase la figura 3.6). La cache de atributos (*acache*) almacena los metadatos de ficheros y directorios durante un tiempo limitado. La cache de nombres (*ncache*), almacena nombres de ficheros y sus correspondientes handles. Para prevenir que las caches almacenen información inválida, los datos se invalidan cuando ha expirado un tiempo establecido o cuando el servidor le comunica al cliente que el objeto no existe.

3.1.2.4.3 Job

El interfaz Job es responsable de coordinar la actividad concurrente entre el BMI, Trove y Flow. Uno de los propósitos del interfaz job es proporcionar un único punto para comprobar la finalización de operaciones de cualquier componente de E/S. Por ejemplo, un llamador puede desear saber a la vez si han finalizado las operaciones de BMI y Trove. El interfaz Job

logra esta comprobación, integrando todas las operaciones en una sola cola, de esta manera se puede preguntar por un solo conjunto de funciones de acceso. El interfaz Job utiliza un interfaz no bloqueante.

3.1.2.4.4 Flow

La interfaz Flow es un flujo de datos entre dos puntos, que pueden ser la memoria, BMI o Trove. El usuario puede escoger entre diferentes protocolos de flujo definiendo el comportamiento de la transmisión de los datos. Por ejemplo, la estrategia de almacenamiento y el número de mensajes transferidos paralelamente pueden ser diferentes para dos protocolos de flujos. Para iniciar una transferencia de datos, la interfaz Flow necesita saber la definición de los datos (tamaño, posición en memoria, etc) y el destino. Con esta información la capa Flow se encarga de la transmisión de los datos.

El interfaz de Flow se ha diseñado de una manera modular. Se identifica el origen y el destino de los datos (del flujo) son llamados *endpoints*, que pueden ser cualquier dirección de red, referencias al dispositivo de almacenamiento y direcciones de memoria. Los flow son usados tanto en el lado del servidor como en el cliente para todas las operaciones de E/S. Cada parte (origen y destino) debe enviarse operaciones de sincronización. La red es el elemento común entre los flow del cliente y servidores. Es por ello que la interfaz Flow emplea etiquetas en los mensajes, así el emisor y/o receptor sabe cual es el primero y evita equivocaciones en accesos concurrentes.

3.1.2.4.5 BMI (Buffered Message Interface)

BMI provee una interfaz de red que permite el intercambio de mensajes entre dos nodos. Los clientes se comunican con los servidores mediante el protocolo de peticiones que define el diseño de los mensajes para cada operación en el sistema de ficheros. BMI implementa actualmente diferentes métodos de comunicación por ejemplo: TCP, Myricom e Infiniband.

BMI se ha diseñado específicamente para proporcionar la semántica y escalabilidad necesarias para satisfacer un sistema de ficheros paralelo de gran escala.

3.1.2.4.6 Trove

La interfaz de Trove proporciona y administra el espacio de almacenamiento para los objetos del sistema. Estos objetos se almacenan de dos maneras: como un par valor/palabra-llaveo *keyword/value* (keyval) o como una secuencia de bytes o *bytestream*. Los pares keyword/valuese usan para almacenar información de metadatos, mientras que los *bytestream* almacenan los datos de un fichero. Los `keyval` se acceden resolviendo la llave (*key*). A los datos *bytestream* se accede en bloques contiguos usando un tamaño (*size*) y una posición inicial (*offset).* También para cada uno de los objetos de almacenamiento hay un conjunto de atributos comunes almacenados.

3.1.2.4.7 Bucle principal del servidor (Server Main Loop)

El bucle principal del servidor verifica la realización de las máquinas de estados procesadas por las hebras. Una máquina de estados consiste en un conjunto de pasos o estados. Cada uno de los estados ejecuta una acción, dicha acción debe ser completada para realizar la transición hacia otro estado y este a su vez realice las operaciones correspondientes. Un ejemplo de maquina de estados puede verse en la sección 4.2.1del capítulo 4, donde se explica con más detalle el funcionamiento de las máquina de estados del cliente.

Cada transición incluye el nombre del siguiente estado. Cabe mencionar que un posible estado puede llamar a otra máquina de estados anidada en lugar de ejecutar una acción de estado. En este caso la nueva máquina de estados es ejecutada como una subrutina. Al terminar, la máquina de estados regresa a la máquina de estados llamadora o inicial.

Para cada una de las operaciones que se realiza dentro de PVFS2 se inicia una máquina de estados, en concreto, la correspondiente al tipo de operación que se desea iniciar.

3.1.2.5 Interacción Cliente/Servidor en PVFS2

La interacción cliente/servidor se realiza cuando los clientes contactan con alguno de los servidores, inicialmente se obtiene información de la configuración del sistema de ficheros. Una vez que el cliente obtiene esta información, estará listo para operar. A continuación se describe la manera de acceder a un fichero en PVFS2

A partir del nombre del fichero y mediante una operación de búsqueda (*lookup*) se obtiene su handle. Mediante este handle cualquier cliente puede intentar acceder a alguna región del fichero (esta operación puede fallar por los permisos del fichero). Si el handle es inválido, el servidor responderá al momento de intentar acceder al archivo que el hanlde no es válido.

3.1.2.6 Consistencia del sistema de Ficheros PVFS2

Uno de los elemento más complicados en la construcción de sistemas de ficheros distribuidos de cualquier tipo es la de mantener la consistencia del sistema de ficheros en presencia de operaciones concurrentes, especialmente aquellas que modifican la jerarquía de directorios.

Tradicionalmente en los sistemas de ficheros se proporciona una infraestructura de bloqueos que se usa para garantizar que los clientes puedan ejecutar ciertas operaciones atómicamente, operaciones como crear o eliminar ficheros. Desafortunadamente esos sistemas de bloqueos agregan latencia adicional al sistema y a menudo grandes complicaciones debido a las optimizaciones que son necesarios para la gestión de fallos.

PVFS2 no utiliza este tipo de sistemas de bloqueo para garantizar operaciones atómicas. En cambio, obliga a todas las operaciones que modifican la jerarquía del sistema de ficheros a ser ejecutadas en una manera que resulten en cambios atómicos en el sistema de ficheros. Para esto los clientes ejecutan secuencias de pasos (peticiones al servidor) que resultan en

operaciones atómicas en el sistema de ficheros. Veamos a continuación cómo es que PVFS2 logra lo anterior mediante un ejemplo de la creación de un nuevo fichero.

Ejecutando los pasos siguientes resultará que, en el sistema de ficheros, existirá una entrada de directorio para un fichero que realmente aún no está listo para ser accedido entre los pasos 1 y 5:

1. Crear una entrada de directorio para un nuevo fichero
2. Crear un objeto de metadatos para un nuevo fichero
3. Hacer que la entrada de directorio apunte al objeto de metadatos
4. Crear un conjunto de objetos de datos para mantener los datos para el nuevo fichero
5. Hacer que los metadatos apunten hacia los objetos de datos.

Si estas operaciones se ordenan cuidadosamente, se puede crear una secuencia de estados que siempre dejan a la jerarquía de directorio del sistema de ficheros en un estado consistente. Todas las operaciones de PVFS2 se ejecutan de esta otra forma:

1. Crear los objetos de datos para mantener los datos para un nuevo fichero
2. Crear un objeto de metadatos para un nuevo fichero
3. Hacer que los metadatos apunten hacia los objetos de datos
4. Crear una entrada de directorio para un nuevo fichero apuntando hacia el objeto de metadatos.

Con estos pasos lo último que se hace es dejar visible el objeto para que se pueda usar. Entre los pasos 1 y 4 no se puede acceder al objeto.

3.1.2.7 Representación lógica y física de datos en PVFS2

En PVFS2 los metadatos (*metafile*) son almacenados en servidores configurados para realizar esta operación. En cambio, los datos (*datafile*) son almacenados en distintos servidores, divididos en objetos o bloques, que por defecto, serán de 64 KB. (en el apartado 3.3.1.1 se hace referencia a la modificación del tamaño de bloque en la distribución)

Los ficheros son representados lógicamente en los servidores de metadatos y físicamente en los servidores de datos. Cada uno de estos servidores de datos (llamados servidores de *E/S o I/O servers*) son responsables de un conjunto de bloques. Cada bloque se identifica mediante un handle de bloque. Cada servidor de datos administra un rango distinto de handles y estos rangos son establecidos al momento de configurar el sistema de ficheros. Con esta información de handles, los clientes pueden acceder directamente al servidor que almacena un determinado bloque.

Los bloques de datos son distribuidos sobre múltiples servidores de E/S e incluso pueden ser almacenados en todos los servidores disponibles. Es importante mencionar que los servidores de datos únicamente almacenarán bloques de datos, los metadatos y otros objetos se almacenan en el servidor de metadatos.

3.1.2.7.1 Distribución de datos en pvfs2

En este apartado se menciona la forma que PVFS2 usa por defecto para almacenar los bloques de datos. Esta distribución usa tamaños de bloques de 64 KB por defecto. PVFS2 divide los ficheros en trozos con el tamaño de un bloque. Posteriormente, estos bloques son distribuidos por los servidores de E/S en una distribución establecida por defecto *Round - Robin*, lo que significa que el trozo 1 de 64 KB estará almacenado en un fichero en el primer servidor, el trozo 2 en el segundo y así sucesivamente hasta que cada trozo esté asignado a un fichero en el servidor. El trozo n+1 (donde n es el número de servidores de E/S) es agregado al fichero del primer servidor.

Para entender un poco mejor esta técnica veamos un ejemplo. Imaginemos que queremos almacenar en el sistema de ficheros PVFS2 un fichero de 416 KB. Inicialmente PVFS2 divide el fichero en 6 trozos, siendo el último de estos de 32 KB. Estos bloques son repartidos a través de todos los servidores de E/S. En la figura 3.7 se puede observar cómo se almacenan los bloques por los distintos servidores. Si el tamaño del fichero crece, por defecto, PVFS2 completará el bloque de 32 KB hasta que sea de tamaño de 64 KB. Sólo se permite que los bloques finales sean menores del tamaño específico. En el caso de la figura 3.4, el

crecimiento del fichero conllevaría a que el fichero del servidor 2 crezca hasta 128 KB, antes de que se origine otro bloque en el siguiente servidor.

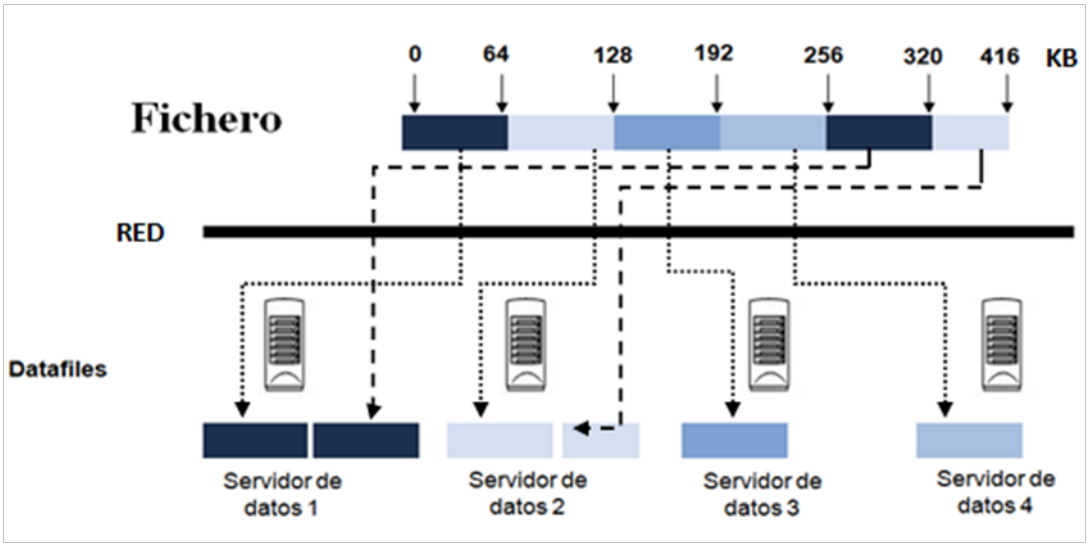


Figura 3. 7 Distribución Round-Robin en PVFS2

3.1.3 Diferencias entre PVFS y PVFS2

Una de las características más importantes que hay que destacar en PVFS2, es que a diferencia de su antecesor, se ha implementado un diseño modular en cuanto a la arquitectura del sistema se refiere. La división en módulos facilita las modificaciones de la implementación y que las implementaciones alternativas se puedan introducir fácilmente. Por ejemplo, en BMI se puede disponer de varios módulos para la comunicación (TCP/IP, InfiniBand, Myrinet) implementados por distintos grupos de investigación o en Trove pueden existir también varios módulos alternativos para la iteración con el disco desarrollados igualmente por distintos grupos. Además de esta forma también es más fácil su mantenimiento.

La tabla 3.1 [Ramachandran02], muestra algunas de las diferencias más destacables PVFS y PVFS2.

Tabla 3.1 Comparación de PVFS y PVFS2

	PVFS	**PVFS2**
Arquitectura de software	Monolítico	Modular Por capas
Peticiones E/S	*Strided* Basado en listas	Completamete General Basado en MPI
Distribución de datos	*Striping*	Código de distribución modular
Red	TCP/IP	Capa de red BMI (TCP/IP,Myricom,Infiniband)
Almacenamiento	Ficheros Unix	Capa de almacenamiento Trove
Servidores	Almacenan estado (stateful) Servidor de E/S y Servidor de metadatos separados	No almacenan estado (stateless) Servidor de E/S y servidor de metadatos combinados
Interfaces	Basado en Posix con extensiones. Otras, con capas.	Interfaz del sistema de bajo nivel Interfaces de usuario remplazables

Las aplicaciones paralelas a menudo requieren acceder a regiones de datos no continuos dentro de un fichero, por lo que se buscó que PVFS2 pudiese administrar este tipo de operaciones de E/S. Se implementó una fuerte integración con MPI-IO, de esta manera varias peticiones en paralelo son enviadas a la vez, por lo que el rendimiento en las operaciones de E/S se incrementa.

PVFS distribuye los datos en múltiples servidores de E/S. Distribuyendo los datos en múltiples servidores, los clientes poseen diferentes rutas hacia los datos, eliminado de esta forma los cuellos de botella y mejorando el ancho de banda para múltiples clientes. No obstante, la versión de PVFS2 tiene un mecanismo de distribución modular que soporta múltiples distribuciones. Por lo que permite elegir la distribución que más se adecue al modelo de acceso de una aplicación.

PVFS fue diseñado para trabajar con redes TCP. A diferencia de esto, PVFS2 provee una interfaz que permite al sistema poder interactuar con distintos protocolos y dispositivos de

red, por ejemplo InfiniBand, Myrinet y TCP. Esta interfaz, llamada BMI (*Buffered Message Interface*), permite administrar tantos mensajes pequeños como grandes con gran eficiencia.

A diferencia de PVFS, PVFS2 cuenta con una capa de almacenamiento llamada Trove, la cual se encarga del almacenamiento de los datos.

Otra diferencia importante es que en PVFS los servidores deben mantener los estados de los ficheros que se encuentran abiertos, lo que da lugar a una complejidad cuando los servidores caen por un fallo. En PVFS2, los servidores están sin estado (*stateless*), por lo que se reduce la complejidad y la facilidad en el diseño de múltiples procesos en los servidores.

Se ha concluido el estudio del sistema de ficheros PVFS, en sus diferentes versiones. En el siguiente apartado se trata el tema de las implementaciones de cache para datos realizadas en PVFS

3.1.4 Implementaciones en PVFS de cache de datos en clientes

En este apartado se hace una descripción de las distintas implementaciones de memoria cache que se han realizado en PVFS, inicialmente se propuso con una cache de datos en los clientes a nivel de kernel, y posteriormente una la implementación de una cache colaborativa dentro de los clientes de PVFS2.

3.1.4.1 Cache para datos en PVFS

Vilayannur [Vilayannur02], en su artículo “Kernel-LevelCachingforOptimizing I/O by Exploting Inter-Aplication Data Sharing”, explica como realiza la implementación de cache de E/S en los clientes fuera del código de PVFS.

La intención en el diseño de esta alternativa fue proveer a PVFS de características que incrementaran las prestaciones del sistema. La cache de los clientes trabaja sólo con los servidores de E/S por tanto no cachea ninguna información de los servidores de metadatos.

La cache de cada nodo se implementa como un módulo en el kernel de Linux (véase la figura 3.8). Las llamadas al sistema vía socket realizadas por `libpvfs` se redireccionan a este módulo del kernel. El módulo verifica si la petición realizada por la llamada al sistema se corresponde con datos que se encuentran en la cache que gestiona el módulo. Proporciona a libpvfs los datos que ha pedido que se encuentran en la cache. Para obtener los datos que no están en la cache, el módulo del kernel envía un mensaje vía socket al servidor de E/S apropiado (que es lo que hace siempre PVFS sin este módulo).

El protocolo de lectura de `libpvfs` une todas las lecturas a cada servidor de E/S y envía un mensaje vía socket a cada uno de ellos. Después espera a que llegue un reconocimiento y los paquetes de datos procedentes de cada servidor de E/S. El módulo de cache implementado verifica si los datos solicitados se encuentran en la cache. Los datos que se encuentran en la cache se descartan de las peticiones a los servidores de E/S. Como consecuencia de estos descartes puede ocurrir que se genere más de una petición para un servidor de E/S (ocurre si en el conjunto de bloques contiguos solicitado a un servidor hay un bloque que está en la cache). El módulo devuelve el control a `libpvfs` después de apuntar como pendientes los bloques que se han pedido.

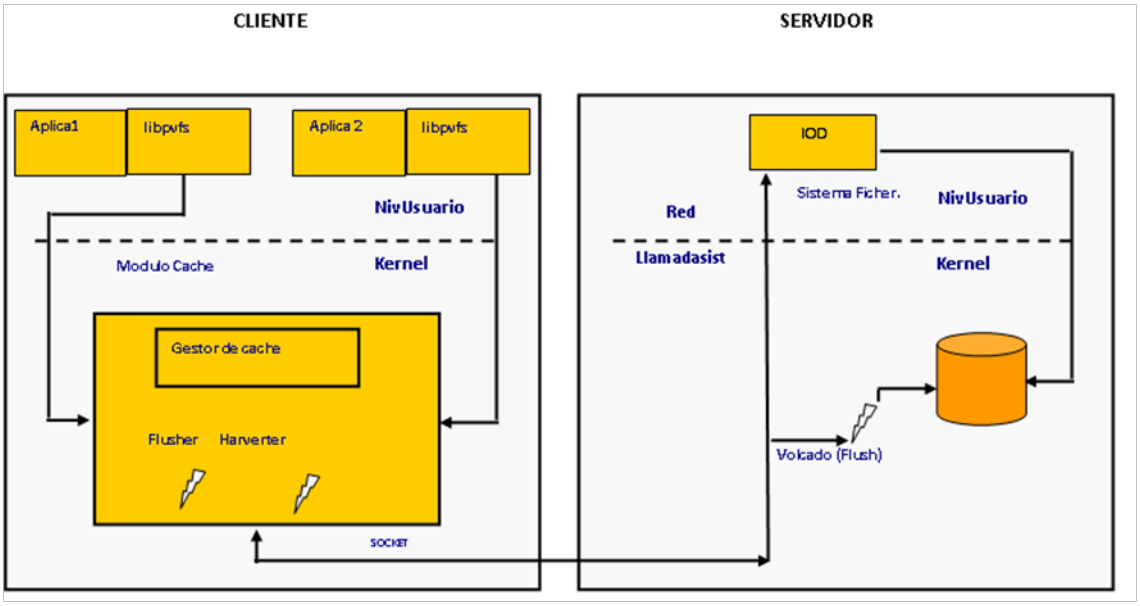


Figura 3. 8 Cache para PVFS propuesta en [Vilayannur02]

La cache descrita en [Vilayannur02] implementa un gestor de buffer completo que usa en cada cliente una tabla hash, una lista de bloques de cache libres (*free list*) y una lista de bloques de cache modificados (*dirty list*). La tabla hash se usa para albergar los bloques de cache que están ocupados. Se usa una tabla hash para mejorar el tiempo de búsqueda y de acceso a bloques. Los bloques tienen tamaño de 4 KB

El módulo del kernel, además del código que gestiona directamente las peticiones de escritura y de lectura, incluye códigos que se ejecutan en varias hebras; especialmente relevantes son las hebras denominadas *flusher* y *harvester* (recolector).

En el cliente, *flusher* se encarga de enviar los bloques modificados al servidor y, en éstos, de almacenar en disco los bloques enviados por el flusher del cliente. Se usa las hebras *flusher* porque las modificaciones que se realizan en bloques de la cache no se envían inmediatamente a los servidores implicados, sino que se colocan los bloques modificados en una lista de modificados (dirty list). Flusher “vierte” periódicamente el contenido de esta lista en los nodos de E/S; es decir, envía periódicamente los bloques de la lista de modificados a los nodos de E/S(concretamente, a los que tienen esos bloques en disco) dejándola vacía.

La hebra *harvester* se activa cuando el número de bloques libres baja a un determinado valor umbral (es decir, cuando la lista de bloques libres está casi vacía). *Harvester* desaloja un número bloques de la caché que están ocupados. El número que desaloja está fijado de antemano y se decide utilizando un algoritmo de reemplazo LRU aproximado (no usa un algoritmo LRU exacto porque supone una penalización demasiado alta en cada petición de lectura/escritura). El algoritmo LRU da prioridad a la hora de desalojar a los bloques que no están modificados. Dado que las estructuras de datos las utilizan varias actividades a la vez su acceso está protegido usando cerrojos (*locks*).

El proceso de escritura comentado, que es el que se utiliza por defecto, no garantiza coherencia de cache porque no propaga lo que se escribe en una cache al resto de caches del sistema. Esta escritura supone un problema para aplicaciones en las que se comparten datos que se modifican. Para las aplicaciones en las que se necesita compartir datos por parte de

varios clientes se proporciona una versión especial de escritura denominada `sync_write`. La escritura `sync_write` propaga lo que se escribe al servidor al que pertenece el bloque modificado e invalida las copias de lo modificado en otras caches. Como la coherencia se mantiene con la granularidad de un bloque, se usa un directorio con una entrada por bloque que mantiene qué caches tienen copia de este bloque. Realmente, el directorio se compone de un subdirectorio en cada servidor de E/S que tiene información sobre dónde hay copias de los bloques que tiene almacenados en los discos dicho servidor.

3.1.4.2 Cache Colaborativa en PVFS

Existen trabajos en los que se proponen la implementación de cache colaborativa [Dahlin94, Hwang04] en PVFS. La cache colaborativa permite que las peticiones de datos que no se satisfacen en la cache local de un cliente se puedan satisfacer por la cache de otro cliente. En caso de no encontrarse copia en otro cliente se accede al disco del servidor de E/S donde se encuentra el bloque.

La cache colaborativa mejora las prestaciones del sistema de ficheros porque provoca una reducción en las peticiones al servidor y mejora el tiempo de acceso. Se consigue mejorar el tiempo de acceso porque traer datos desde la memoria de otro cliente es más rápido que traerlos del disco de un servidor de E/S. En definitiva con la cache colaborativa se crea una gran cache global que disminuye el número de fallos de cache. Cuanto mayor sea el número de nodos cliente mayor será el tamaño de esta cache global y el número de aciertos. Este comportamiento permite intuir que una implementación de cache colaborativa puede conseguir un mejor escalado del sistema que otras alternativas de implementación. Una de las implementaciones realizadas sobre cache colaborativa en PVFS es: *Coopc-PVFS* (véase la figura 3.9), se encuentra agregado en un módulo del kernel de PVFS. Como se dijo en el capítulo anterior, la cache colaborativa permite compartir los datos entre caches. Por ejemplo; cuando se presenta una situación de lectura, un administrador de cache lo primero que hace es buscar los datos dentro de su propia cache, en caso de que no los localice, busca en la cache de otros clientes. Si alguno mantiene los datos, el administrador de cache los obtiene de ese cliente; pero si los datos no se localizan en ninguna cache, entonces los datos se solicitan al servidor de E/S.

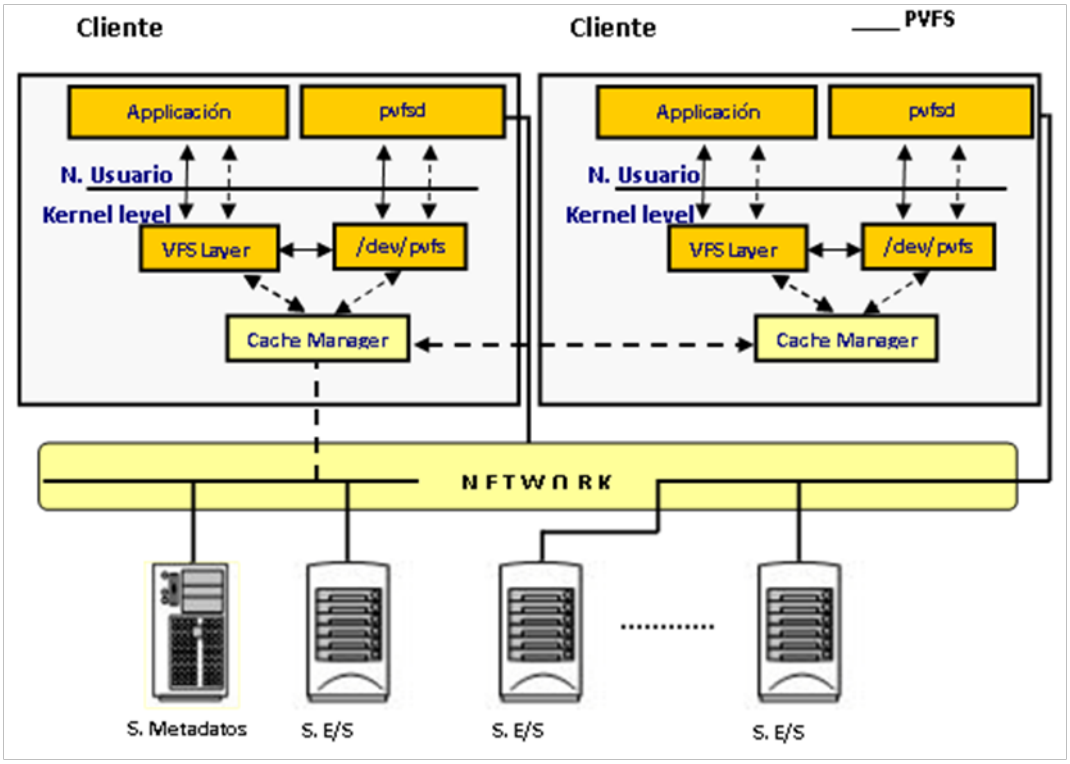


Figura 3. 9 Arquitectura de Coopc-PVFS

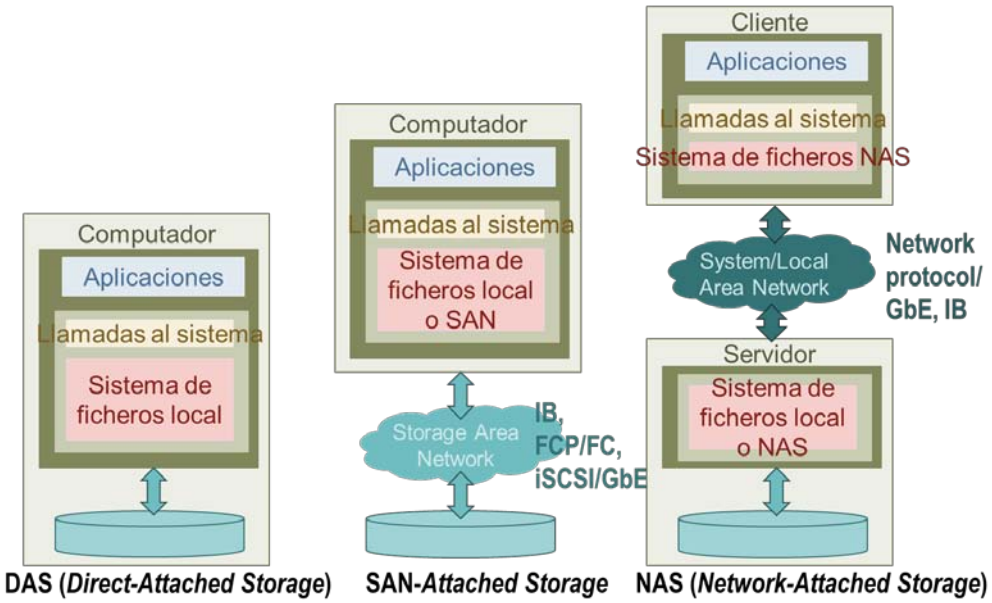
Cuando la aplicación requiere realizar una escritura, el administrador de cache debe enviar un mensaje de invalidación del bloque al administrador de metadatos, quien se encarga de notificar a todos los clientes que permanecen con una copia del bloque abierto. Una vez que esta petición ha sido propagada en todos los clientes, estos invalidan su copia y por lo tanto se puede escribir en el servidor de E/S. De esta forma se mantiene la coherencia de cache en Coopc-PVFS. Así concluimos el análisis de las distintas implementaciones de cache de datos en PVFS.

3.2 AbFS

AbFS es un sistema de ficheros que se está desarrollado desde el año 2010 en la Universidad de Granada por el grupo de investigación de sistemas distribuidos en el que colaboran investigadores tales como: Antonio Francisco Díaz García, Mancia Anguita López, Julio Ortega Lopera, entre otros.

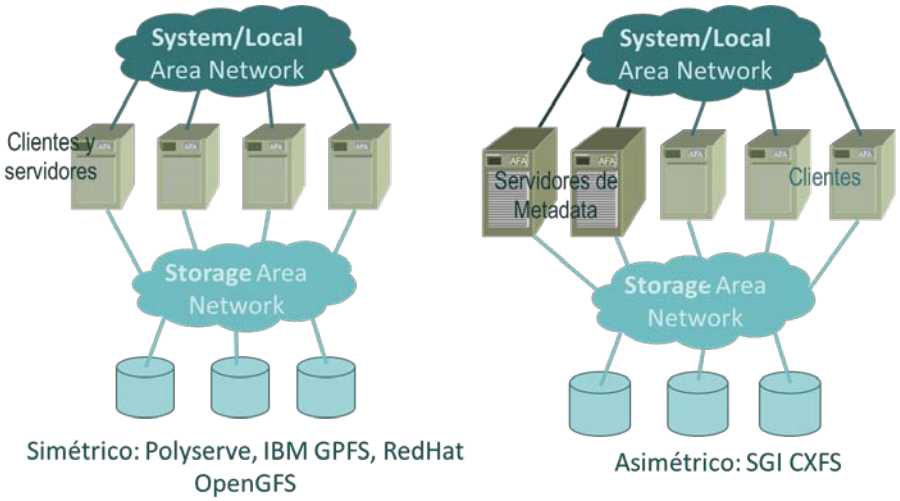
En [Anguita11] se sitúa y clasifica al sistema de ficheros AbFS (*Abierto File System*) dentro del conjunto de los sistemas de los sistemas de ficheros. En esta sección se van utilizar esta referencia para introducir al sistema de ficheros AbFS.

Los computadores actuales (con multinúcleos y procesadores multihebra) se pueden conectar fácilmente usando redes de comunicaciones con el objetivo de obtener plataformas de cómputo de altas prestaciones. A estas plataformas se les puede añadir almacenamiento SAN (figura 3.10, figura 3.11) para proveer al sistema de las prestaciones y capacidad de almacenamiento que requieren las aplicaciones científicas y de propósito general que deben ejecutar. Sin embargo, estas soluciones comerciales de almacenamiento basada en SAN (redes de área de almacenamiento) suponen usualmente una solución cara, especialmente si no se amortiza el desembolso inicial que suponen. Por otra parte, hay redes en plataformas cluster con prestaciones similares a las de las redes dedicadas de sistemas de almacenamiento SAN y, además, los dispositivos de almacenamiento de los computadores del cluster se pueden usar, a través de la red del cluster (ahorro al no necesitar una red extra tipo SAN), como almacenamiento para las aplicaciones (ahorro en almacenamiento extra) [Anguita11].



(a) (b) (c)

Figura 3. 10 Comparativa de almacenamiento (a) DAS, (b) conectado con SAN, (c) NAS



(a) (b)

Figura 3. 11 Sistema de ficheros SAN Cluster simétrico (a) y asimétrico (b)

Los cluster pueden aprovechar los dispositivos de almacenamiento no utilizados que tiene ya disponibles en sus computadores a través de alguno de los sistemas de ficheros libres de reciente aparición, como PVFS[Carns00], Oracle-Sun Lustre [Braam02] and Ceph [Weil06].

Siguiendo esta línea de investigación en sistemas de ficheros distribuidos para cluster, estamos desarrollando AbFS (*Abierto File System*) [Díaz12, Díaz11a, Díaz11b]. AbFS permite que los dispositivos DAS (*Direct Attached Storage*) baratos de los computadores de un cluster se puedan compartir por todos estos computadores. AbFS ofrece una única imagen de estos recursos DAS (figura 3.12).

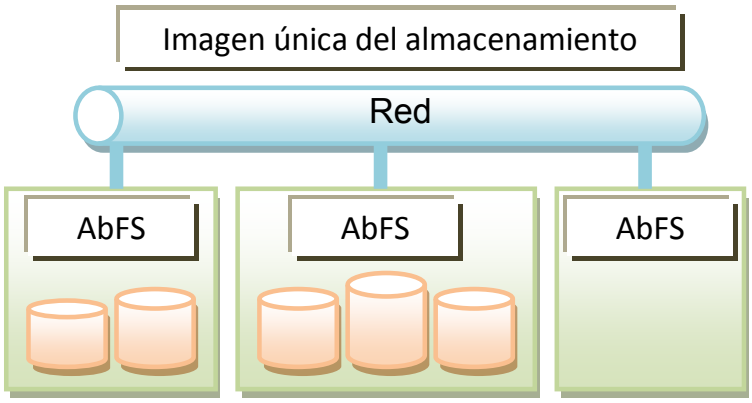


Figura 3. 12 Sistema de ficheros AbFS

Un sistema de ficheros, como por ejemplo AbFS, que agrega al dispositivo local de un computador los dispositivos DAS de otros computadores (que son dispositivos disponibles comercialmente y de precio asequible) a través de una red de área de sistema o local, pueden potencialmente proveer un servicio de almacenamiento de altas prestaciones con un coste bajo (es decir, un sistema de almacenamiento eficiente) en comparación con los sistemas de ficheros SAN (figura 3.11), tales como GPFS [Schmuck02], RedHat GFS [Soltis96], SGI CXFS [SGICorp.11, SGI11], HP PolyServeMatrix Server[Polyserve10] o Oracle OCFS[Oracle11]. En realidad, si el sistema de almacenamiento es gratuito, no habría coste adicional para el cliente ya que el hardware que tiene ya en su plataforma (red, dispositivos DAS) es el que se va a usar en el sistema de almacenamiento [Anguita11].

La figura 3.13 [Anguita11] permite caracterizar a AbFS dentro de los sistemas de ficheros. AbFS es un sistema de ficheros global, es decir, los clientes comparten el espacio de nombres de los ficheros a los que acceden, al menos a partir del punto donde se montan en el espacio de nombres o árbol de directorios. Los sistemas de ficheros globales se dividen en dos grupos en función de si los clientes acceden a los datos usando un protocolo de red o usando un protocolo a nivel de bloque. Actualmente AbFS permite a los clientes acceder al almacenamiento usando un protocolo de red, por tanto es un sistema de ficheros de red o NAS, pero se pretende que en un futuro incluya además el acceso a nivel de bloque usando iSCSI. Como se ha comentado más arriba, AbFS permite que los computadores de un cluster compartan su almacenamiento, luego permite múltiples servidores de datos y, por tanto, que múltiples clientes accedan en paralelo al almacenamiento compartido que está repartido en un cluster de servidores. Es un sistema de ficheros simétrico, es decir, no desacopla el acceso a metadatos del acceso a datos; de hecho los servidores de datos son servidores de metadatos y los clientes pueden ser también servidores. Los clientes tienen información para obtener en qué servidor está los metadatos del objeto al que quiere acceder; por tanto, no hay que pasar por un servidor intermedio para localizar los metadatos.

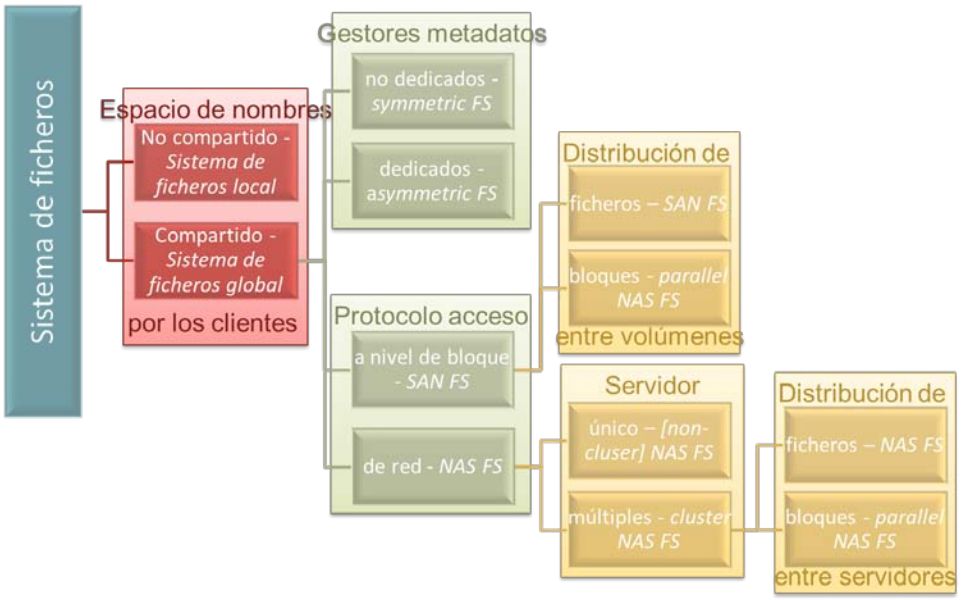


Figura 3. 13 Relación entre distintas clases o tipos de sistemas de ficheros y distintos criterios de clasificación de sistemas de ficheros (espacio de nombres, gestores de metadatos, protocolo de acceso, distribución de datos y número de servidores de datos) (FS: *File System* o Sistema de Ficheros)

A nivel ilustrativo, la figura 3.14 muestras los distintos bloques internos de AbFS.

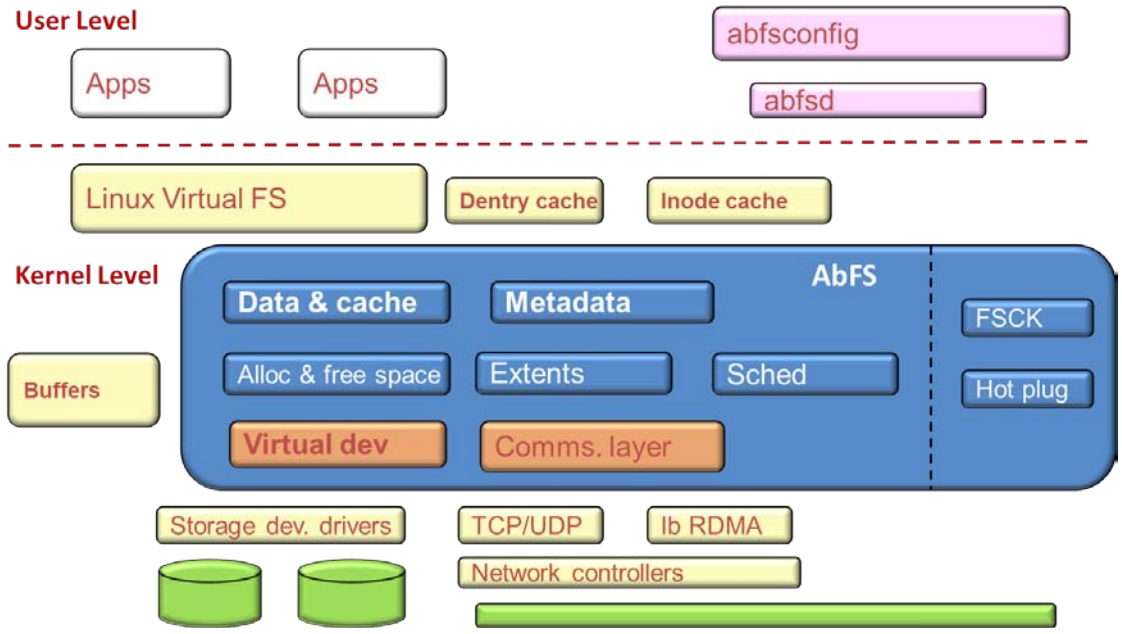


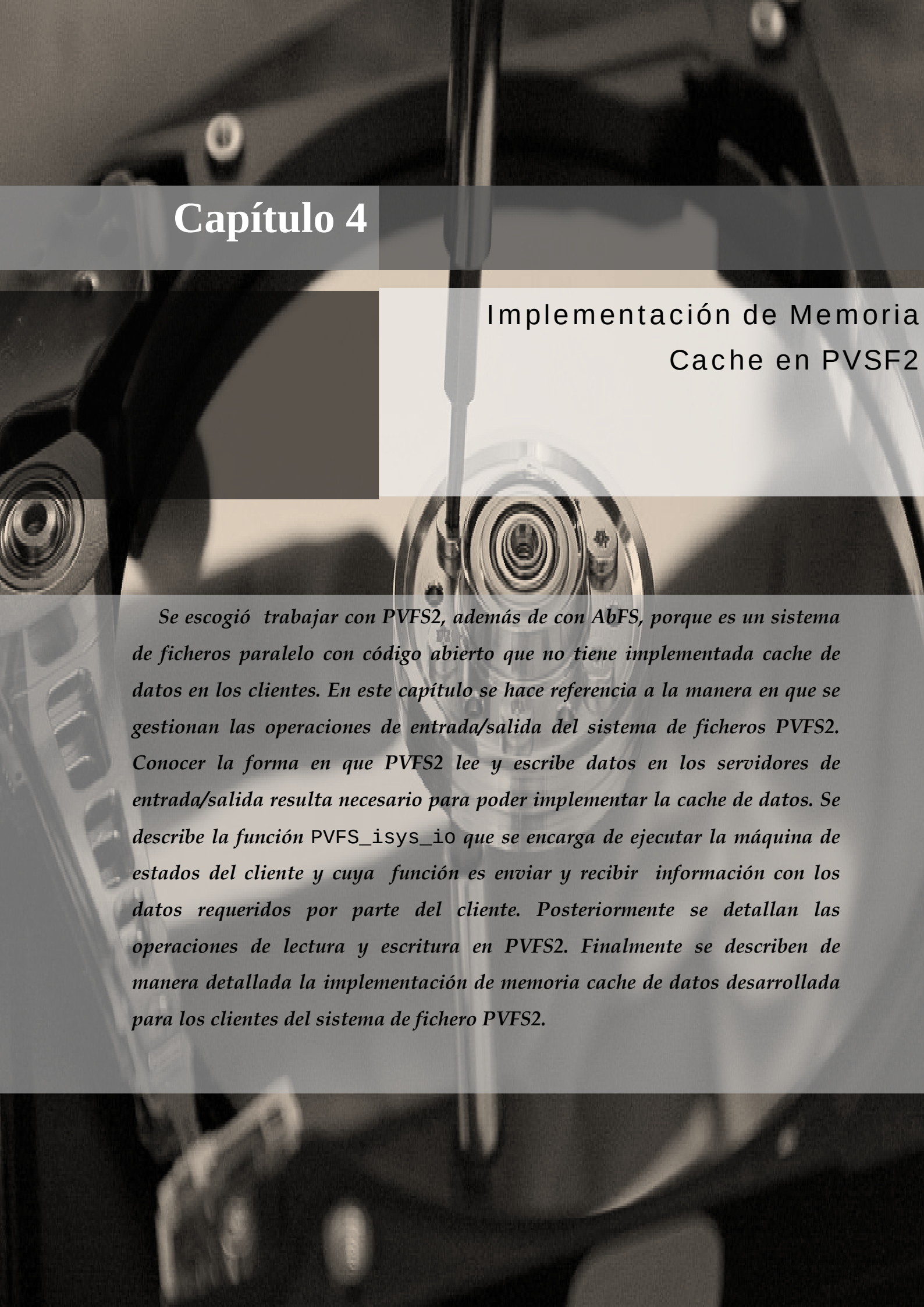
Figura 3. 14 Arquitectura de AbFS

De esta manera se concluye con el análisis de los distintos sistemas de ficheros utilizados para las diferentes propuestas de implementación de cache de datos en los clientes. En el siguiente capítulo se describen las implementaciones de memoria cache de datos realizadas sobre los clientes de PVFS2 [Camacho10] y AbFS [Diaz12, Diaz11a, Diaz11c].

Capítulo 4

Implementación de Memoria Cache en PVSF2

SUMARIO



Capítulo 4

Implementación de Memoria Cache en PVSF2

Se escogió trabajar con PVFS2, además de con AbFS, porque es un sistema de ficheros paralelo con código abierto que no tiene implementada cache de datos en los clientes. En este capítulo se hace referencia a la manera en que se gestionan las operaciones de entrada/salida del sistema de ficheros PVFS2. Conocer la forma en que PVFS2 lee y escribe datos en los servidores de entrada/salida resulta necesario para poder implementar la cache de datos. Se describe la función `PVFS_isys_io` *que se encarga de ejecutar la máquina de estados del cliente y cuya función es enviar y recibir información con los datos requeridos por parte del cliente. Posteriormente se detallan las operaciones de lectura y escritura en PVFS2. Finalmente se describen de manera detallada la implementación de memoria cache de datos desarrollada para los clientes del sistema de fichero PVFS2.*

4.1 Introducción

En los capítulos anteriores de este trabajo se han analizados los objetivos, características, ventajas y desventajas, así como las propuestas de implementación de memoria cache en los sistemas de ficheros para entornos distribuidos y paralelos.

Un objetivo de este trabajo es implementar una memoria cache de datos en los clientes de PVFS2. PVFS2, al igual que su antecesor PVFS1, no incluye cache para datos en clientes.

Inicialmente se hace una descripción de la gestión de las operaciones de entrada/salida en PVFS2. El apartado 4.2.1 presenta qué es una máquina de estados. En particular presenta la máquina de estados del cliente (apartado 4.2.1.1), puesto que a través de los distintos estados de la máquina de estados del cliente se envían y reciben los datos requeridos por los usuarios del sistema de ficheros PVFS2. Para que la máquina de estados del cliente pueda ser lanzada hacia los servidores de E/S se requiere de la función `PVFS_isys_io` (apartado 4.2.2). Esta función gestiona las operaciones de lectura/escritura que el cliente solicita y ejecuta la máquina de estados. Los apartados 4.2.3 y 4.2.4 describen el proceso de una lectura y escritura en PVFS2.

El apartado 4.3 hace mención a la comparativa entre PVFS2 y ext3, cuya finalidad era conocer si PVFS2 podría mejorar el rendimiento del sistema, reducir las latencias e incrementar el ancho de banda en el envío/recepción de los datos debido al uso de la cache de datos en los clientes.

El resto de este capítulo describe la implementación de memoria cache en los clientes de PVFS2 (apartado 4.4). Se muestran parte de las modificaciones de código que han sido necesarias realizar para el correcto funcionamiento en lecturas/escrituras (apartados 4.4.1 y 4.4.2) y para poder volcar datos en los servidores de E/S (Apartado 4.4.3).

4.2 Operaciones de entrada/salida en PVFS2

Es necesario describir el funcionamiento de las operaciones de entrada/salida de PVFS2 para que se puedan comprender las modificaciones que se han hecho sobre el código original. La documentación que aborda las particularidades de la implementación de PVFS2 es escasa en su web oficial, aunque se pueden encontrar algunos detalles en trabajos realizados por terceros [Kunkel06, Kunkel07, Kuhn07]. Inicialmente se describe la máquina de estados del cliente, después se detalla la función `PVFS_isys_io` (función que inicia una operación de lectura/escritura) y por último se presenta, la forma en que PVFS2 gestiona las operaciones de lectura y escritura.

4.2.1 Máquinas de Estados

Las operaciones en PVFS2 se implementan como máquinas de estados, tanto en los clientes como los servidores de PVFS2. Las máquinas de estado están escritas en archivos con extensión `.sm`. El código de las máquinas de estado tiene secciones de código adicionales que se distinguen porque empiezan y terminan con `"%%"`. Un analizador sintáctico (*parser*) transforma estas secciones de las máquinas de estado en código C. De esta forma se obtiene el código C completo que se compila y enlaza.

Una máquina de estados consiste en un conjunto de estados. Un estado inicial indica el comienzo de la máquina. Una máquina de estados sólo puede estar en un estado cada momento. Cada estado incluye una acción (puede ser una función o una máquina de estados anidada) seguida por un conjunto de posibles transiciones a otros estados. El estado al que se va a pasar depende del resultado de la acción: la función o máquina de estados anidada, una maquina de estados anidada se ejecuta dentro de la máquina de estados principal y realiza cierta tarea o acción que ha sido definida previamente. Una vez completada la máquina de estados anidada retorna el control a la máquina de estados principal para que continúe con los posibles estados siguientes. Cada transición incluye el nombre del siguiente estado. En el listado 4.1 se muestra un ejemplo genérico del código de máquina de estados.

Listado 4.1 Ejemplo genérico del código de máquina de estados.

```
     /*Inicio de un archivo .sm  */
     /*El código en la parte superior del archivo es texto en C plano.*/
     /*Las acciones de los estados deben ser declaradas aquí antes de la máquina de
estados.*/

     static  PINT_sm_action  state_action_1(struct  PINT_smcb  *smcb,  job_status_s
*js_p);
     static  PINT_sm_action  state_action_3(struct  PINT_smcb  *smcb,  job_status_s
*js_p);
     static  PINT_sm_action  state_action_4(struct  PINT_smcb  *smcb,  job_status_s
*js_p);

     /* Funciones auxiliares y otras declaraciones */

     #define RETVAL 1
     %%

     /*Después del doble porcentaje va la declaración de la máquina*/

     machine mi_maquina_sm (state_1,state_2,state_3)
     {
         State state_1
                  {
                     run state_action_1;/*run: ejecuta una función relacionada con
el estado donde se encuentra*/
     success => state_2; /*success: significa que la función devuelve 0*/
      default  =>  state_4;/*default:  Este  es  el  último  paso  en  la  lista  de
acciones. Se ejecuta en caso de que la función no devuelva 0, y el código de error
no coincida con ningún valor previamente establecido */
                  }

          State sate_2
                  {
                        Jump   otra_maquina_de_estados_sm;/*jump:   Indica   que   el
siguiente estado a ejecutar es una máquina de estados anidada*/

                      RETVAL => state_3;/*RETVAL: Significa que la transición al
siguiente estado es definido por la variable del código de error js.p->error_code
*/
     }
          State sate_3
                  {
     pjmp state_action_3/*pjmp ejecuta saltos en paralelo*/
```

```
     {

     /*los valores aquí son inicializados en state_action_3*/

     4 => parellel_state_machine_1;
     3 => parellel_state_machine_2;
     RETVAL => parellel_state_machine_3;
               }
     default => state_4;
     }
          State sate_4
                  {

     /*La acción de este estado, limpia los datos después del salto paralelo pjmp*/

     run state_action_4;
     default => terminate;
     }
     }
     %%

     /*Después del segundo doble porcentaje todo el código es en C plano*/
     /*Aquí implementamos todas las acciones de los estados*/

     static  PINT_sm_action  state_action_1(struct  PINT_smcb  *smcb,  job_status_s
*js_p);
     {
     }
     static  PINT_sm_action  state_action_3(struct  PINT_smcb  *smcb,  job_status_s
*js_p);
     {
     }
     static  PINT_sm_action  state_action_4(struct  PINT_smcb  *smcb,  job_status_s
*js_p);
     {
     }
```

4.2.1.1 Máquina de estados del cliente de PVFS2

En los siguientes apartados se describe la máquina de estados del cliente y sus posibles estados, la función (`PVFS_isys_io`) encargada de ejecutar esta máquina de estados y la manera en que se gestiona una operación de lectura/escritura en los clientes del sistema de

ficheros PVFS2. Esta descripción permite comprender de manera clara qué parte de código interviene en las operaciones de E/S en PVFS2 y qué parte ha sido necesaria modificar para el correcto funcionamiento de la implementación de memoria cache para datos en los clientes de PVFS2.

Como se muestra en la figura 4.1, la máquina de estados del cliente puede mantener diferentes estados posibles. Estos estados son ejecutados en base al valor devuelto por cada uno de los estados que previamente se han lanzado. Estos posibles estados se encargan de llamar a las funciones necesarias para obtener atributos (tamaño, ubicación, etc) y datos del fichero que se ha solicitado previamente en una operación de lectura y/o escritura en el cliente.

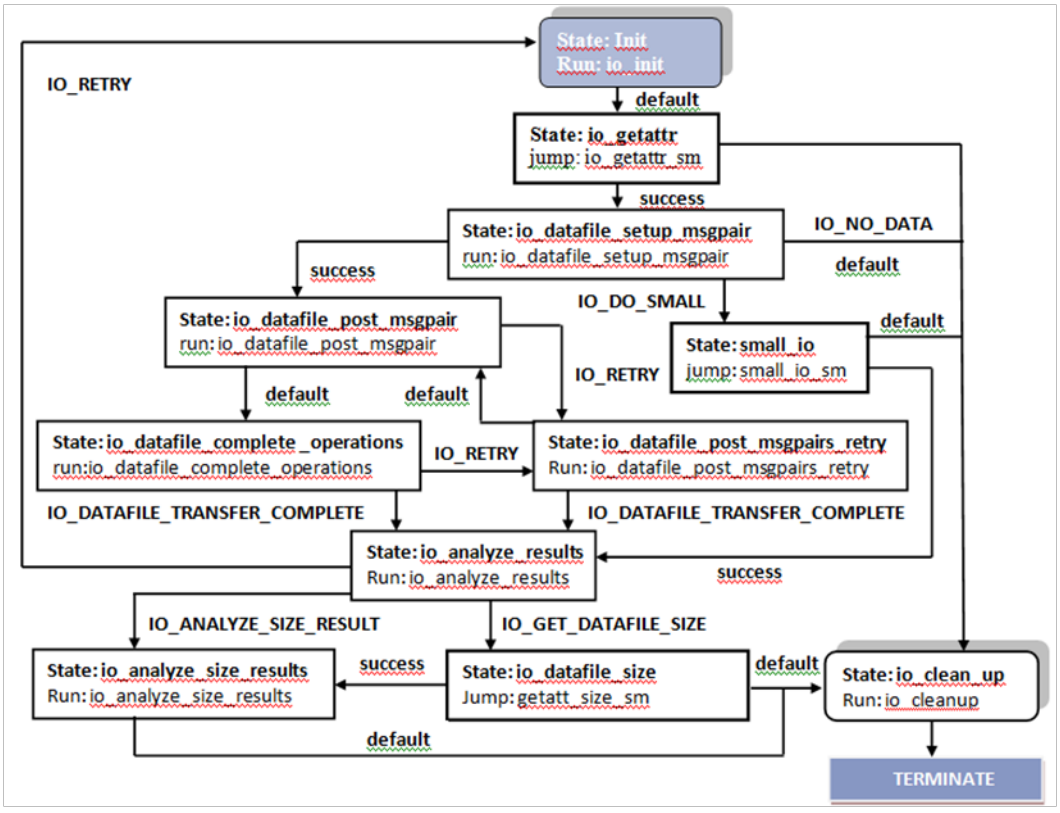


Figura 4. 1 Máquina de estados del cliente de PVFS2

En el listado 4.2 se muestra el código de la máquina de estados del cliente y sus posibles estados, además de la descripción de cada una de las funciones que intervienen dentro de la máquina.

Listado 4.2 Código de la máquina de estados del cliente

```
     /*Inicio de un archivo .sm  */
     /*El código en la parte superior del archivo es texto en C plano.*/
     /*Las acciones de los estados deben ser declaradas aquí antes de la máquina de
estados.*/
     /* Funciones auxiliares y otras declaraciones */
     /*Después del doble porcentaje va la declaración de la máquina*/



%%

machine pvfs2_client_io_sm
{
    state init
    {
run io_init;
    default => io_getattr;
}

    state io_getattr
    {
        jump pvfs2_client_getattr_sm;
        success => io_datafile_setup_msgpairs;
        default => io_cleanup;
 }

    state io_datafile_setup_msgpairs
    {
        run io_datafile_setup_msgpairs;
        IO_NO_DATA => io_cleanup;
        IO_DO_SMALL_IO => small_io;
        success => io_datafile_post_msgpairs;
        default => io_cleanup;
    }

    state small_io
    {
    jump pvfs2_client_small_io_sm;


        success => io_analyze_results;
        default => io_cleanup;
    }

    state io_datafile_post_msgpairs
    {
        run io_datafile_post_msgpairs;
        IO_RETRY => io_datafile_post_msgpairs_retry;
        default => io_datafile_complete_operations;
    }

    state io_datafile_post_msgpairs_retry
```

```
    {
        run io_datafile_post_msgpairs_retry;
        IO_DATAFILE_TRANSFERS_COMPLETE => io_analyze_results;
        default => io_datafile_post_msgpairs;
    }

    state io_datafile_complete_operations
    {

     run io_datafile_complete_operations;
        IO_DATAFILE_TRANSFERS_COMPLETE => io_analyze_results;
        IO_RETRY => io_datafile_post_msgpairs_retry;
        default => io_datafile_complete_operations;
    }

    state io_analyze_results
    {
        run io_analyze_results;
        IO_RETRY => init;
        IO_ANALYZE_SIZE_RESULTS => io_analyze_size_results;
        IO_GET_DATAFILE_SIZE => io_datafile_size;
        default => io_cleanup;
    }

    state io_datafile_size
    {
        jump pvfs2_client_datafile_getattr_sizes_sm;
        success => io_analyze_size_results;
        default => io_cleanup;
    }

    state io_analyze_size_results
    {
  run io_analyze_size_results;
        default => io_cleanup;
    }

    state io_cleanup
    {

    run io_cleanup;
        default => terminate;
}
}

%%
     /*Después del segundo doble porcentaje todo el código es en C plano*/
     /*Aquí implementamos todas las acciones de los estados*/

static PINT_sm_action io_init(
        struct PINT_smcb *smcb, job_status_s *js_p)
{
     /*En esta acción, principalmente se inicializa la estructura getattr, (se
utilizará para obtener los atributos del fichero al que se desea realizar E/S).
Esta estructura es usada en la máquina de estados para almacenar todos los
atributos del fichero al que se le efectuará E/S. Posteriormente se ejecuta el
estado io_getattr, inicialmente este estado, lanza inmediatamente la máquina de
estados  pvfs2_client_getattr_sm (localizada en usr/src/pvfs/src/client/sysint/sys-
getattr.sm), que es la que se encargará de obtener los atributos solicitados*/

     /*En caso de que la función devuelva 0, se ejecuta el siguiente estado
io_datafile_setup_msgpairs, en caso contrario el último estado de la máquina de
estados del cliente es llamado y se finaliza la operación de E/S que este
ejecutando en ese momento. Una vez terminada la acción, se retorna un valor donde
```

```
se indica que la acción se ha completado*/
return SM_ACTION_COMPLETE;
}


static PINT_sm_action io_datafile_setup_msgpairs(
        struct PINT_smcb *smcb, job_status_s *js_p)
{
     /*Una  vez que la máquina de estados pvfs2_client_getattr_sm ha finalizado, y
si es que no ha ocurrido un error en el proceso, el siguiente paso es lanzar la
función io_datafiles_setup_msgpairs, (msgpairs, par de mensajes que se envían a los
servidores) y tiene como principales acciones las siguientes:

     1.- Inicialmente verificar algunos errores que puedan haber ocurrido en la
máquina de estados anterior, como que el fichero al que se desea hacer
entrada/salida no sea un directorio o un tipo de fichero no conocido. Además de que
el fichero al que se desea hacer E/S no sea un fichero inmutable.

     2.- Posteriormente se inicializa el arreglo de índices de datafiles, que será
utilizado para almacenar los índices de los datafiles que se les realizará la E/S.
Esto es importante, por ejemplo cuando se quiere realizar una escritura o una
lectura en cierta parte del fichero, lo que representa que no se escriba en todos
los datafiles que representan lógicamente al fichero, si no que solo se lleve a
cabo en cierto número de ellos. Para eso se utiliza este arreglo.

     3.- Una vez inicializado el arreglo de índices, se inicializa el arreglo
dfile_size_array, el cual es de tipo PVFS_size (Entero de 64 bits) y almacenará los
tamaños de los datafiles en cada uno de los servidores.

     4.- Ahora se inicializa un arreglo de enteros llamado sio_array, el cual se
utiliza dentro de la función io_find_target_datafiles, para almacenar los índices
de los datafiles a los cuales se les puede realizar E/S pequeña.

     5.- Como siguiente paso se llama la función io_find_target_datafiles(), la
cual determina el subconjunto de datafiles que actualmente contienen datos a los
cuales se está interesado en la petición. Realmente calcula si es que un
determinado datafile contiene datos a los que se está interesado ingresar
(lecturas) o si es que se deben introducir datos en algún determinado datafile
(escrituras). Los índices de los datahandles se almacenan en la variable sm_p-
>u.io.datafile_index_array, y el numero de los datahandles a los que se debe
realizar E/S se almacena en target_datafile_count. Además en la variable sio_array
se almacenan los índices de los datafiles a los que se les puede realizar E/S
pequeña,   y sio_count determina a cuántos de ellos se les puede hacer esta
operación.

     6.- Después de saber a cuales datahandles se les debe realizar E/S, se
comparan las variables sio_count, que indica a cuantos datafiles se les puede
realizar E/S pequeña y la variable target_datafile_count, la cual almacena el total
de datahandle a los que se debe leer o escribir datos, según sea el caso de la
operación. Esta operación (sio_count  == target_datafile_count) indica si es que la
máquina de estados debe pasar a un nuevo estado el cual realiza E/S pequeña a los
datafiles.

     Este nuevo paso evita que se realicen flujos hacia los servidores, enviando
los datos que se desean escribir o leer en un mensaje inesperado, y de igual
manera, el servidor envía o recibe los datos en este mismo tipo de mensajes.Por
ahora, la semántica seguida de la E/S pequeña solo se lleva a cabo si es que los
tamaños de los datafiles son pequeños ( a todos los datafiles se les puede realizar
E/S pequeña). Según el código, esto puede cambiar en un futuro, por ejemplo si es
que sio_count es solo un porcentaje de los datafiles, comparados mediante un
umbral.

     7.- Si es que no se puede hacer entrada salida pequeña se continua con el
lanzamiento de la función io_context_init(), la cual inicializa los contextos que
```

```
serán utilizados en la comunicación con los servidores. Principalmente esta función
crea espacio para datafiles_count estructuras para posteriormente inicializarlas.
El principal campo que se utiliza para inicializar los contextos es msg. La cual es
una estructura de tipo PINT_msgpair_state. Básicamente, un contexto es una entidad
que se encarga de controlar cada una de las operaciones de E/S de los clientes
hacia los servidores. Cada contexto se almacena mediante una estructura llamada
PINT_client_io_ctx.

     En el caso de que la petición a realizar sea una entrada/salida pequeña se
ejecuta la máquina de estados pvfs2_client_small_io_sm, localizada en
usr/src/pvfs/src/client/sysint/sys-small-io.sm.*/

/*Maquina de estado para peticiones pequeñas, y se encuentra anidada en la máquina
de estados del cliente*/

nested machine pvfs2_client_small_io_sm
{
    state setup_msgpairs
    {
        run small_io_setup_msgpairs;
        success => xfer_msgpairs;
        default => return;
    }

    state xfer_msgpairs
    {
        jump pvfs2_msgpairarray_sm;
        default => cleanup;
    }

    state cleanup
    {
        run small_io_cleanup;
default => return;
    }
}

%%

     /*La máquina de estados para peticiones pequeñas, cuenta de 3 posibles estado
donde se realiza la lectura/escritura en los servidores de E/S. Una operación de
lectura/escritura pequeña se ejecuta cuando el tamaño de la petición es más pequeño
que el tamaño de un mensaje inesperado aceptado por la interfaz de BMI , (En esta
versión de PVFS, una E/S pequeña es para peticiones menores de 16KB). Por tanto no
es necesario ejecutar una configuración inicial para el mensaje antes de enviar los
datos actuales(FLOW), en lugar de eso, solo se empaquetan los datos en un mensaje
inesperado.

     Si no hay E/S pequeña, la acción de este estado se ejecuta y la acción del
siguiente estado es llamada(io_datafile_post_msgpairs).Una vez terminada la acción,
se retorna un valor donde se indica que la acción se ha completado*/
return SM_ACTION_COMPLETE;
}



static PINT_sm_action io_datafile_post_msgpairs(
        struct PINT_smcb *smcb, job_status_s *js_p)
{
     /*Esta acción ejecuta las siguientes operaciones para cada msgpair:

     1) Codifica la petición
     2) Calcula el tamaño máximo de respuesta del servidor.
     3)  Reserva  espacio  de  memoria  tipo  BMI  para  los  datos  de  respuesta
```

```
(codificada)
     4) Obtiene una etiqueta de sesión (session tag) para los pares de mensajes.
     5) Lanza la recepción de la respuesta
     6) Lanza elenvió de la petición
     7) Almacena los id de los "job" para verificarlos posteriormente.

     Una vez terminada la acción, se retorna un valor donde se indica que la acción
se ha completado*/

return SM_ACTION_COMPLETE;

}

static intio_datafile_post_msgpairs_retry(
        struct PINT_smcb *smcb, job_status_s *js_p)
{


     /*En caso de haber un error en algún msgpair en la acción anterior, esta
acción io_datafile_post_msgpairs_retry, hace una reintento para E/S, esperando un
poco y envía nuevamente los msgpair fallidos.Una vez terminada la acción, se
retorna un valor donde se indica que la acción se ha completado*/


return SM_ACTION_COMPLETE;
    }


static PINT_sm_action io_datafile_complete_operations(
        struct PINT_smcb *smcb, job_status_s *js_p)
{


     /*Esta acción permite asegurarse que todas las operaciones ejecutadas están
terminadas y contabilizadas. Ya que, este paso maneja todas las terminaciones de
las operaciones, existe un caso especial en la terminación de una recepción de
msgpair (msgpir recv), en este caso se lanza la operación de flujo (flow operation)
tan pronto como se determine que se ha recibido la confirmación de recepción*/

     /*una vez terminada la acción, se retorna un valor donde se indica que la
acción se ha completado*/

return SM_ACTION_COMPLETE;
}

static PINT_sm_action io_analyze_results(
        struct PINT_smcb *smcb, job_status_s *js_p)
{
      /*En este estado se encarga de verificar que los datafile mantengan el
      número de bytes solicitados con la finalidad de que se devuelva al cliente
      el valor correcto, por lo que se cerciora de la existencia de huecos en el
      fichero lógico. El algoritmo para ello es el siguiente.

      1. Si el tamaño de la solicitud es equivalente a la cantidad de  bytes
      leídos, sabemos que no hay agujeros o huecos y el tamaño total es el valor
      correcto para devolver al llamador.

      2. Si el punto anterior es falso, indica que hay  un hueco en el fichero
      dentro de la región solicitada o que la petición ha sobre pasado el final
      del fichero. Si la petición no ha sobrepasado el final del fichero, entonces
      el valor retornado de bytes leídos es equivalente al tamaño de la petición.
      Para comprobar que la petición no ha llegado al final del fichero, se
      recorren los datafiles requeridos, calculando los offsets de cada uno y
      comparándolos con los bstream dentro de los servidores. Si estos tamaños
```

```
       coinciden, se sabe que la petición no ha llegado al final del fichero y por
       tanto se retorna que el tamaño de bytes leídos es equivalente al tamaño de
       la petición.

       3. Si ninguno de los offset de los datafile de destino son >= al upper bound
       (límite superior) del fichero solicitado, se verifican todos los datafile
       que estén más allá del último datafile de destino. Para esto, es necesario
       obtener el tamaño de cada uno y ejecutar una comparación como la descrita en
       el punto 2, si bien, si uno de los offset es >= al upper bound de el fichero
       solicitado, entonces se conocerá que los bytes retornados son del tamaño del
       fichero solicitado.

       4. Si, no encontramos ningún datafile con un offset >= al upper bound del
       fichero solicitado, el valor retornado de los bytes leídos es el tamaño de
       la petición del fichero menos el ultimo offset de los datafile(donde ocurre
       el final del fichero(EOF). Una vez terminada la acción, se retorna un valor
       donde se indica que la acción se ha completado*/

return SM_ACTION_COMPLETE;
}

static PINT_sm_action io_analyze_size_results(
        struct PINT_smcb *smcb, job_status_s *js_p)
{

     /*Una vez llegado a este estado, se conocen todos los tamaños de los datafile,
este estado, permite finalizar la verificación de que los datos solicitados no
sobrepasen el final del fichero, y regresa los valores apropiados para los bytes
leídos al cliente. La verificación itera a través de todos los datafile y compara
su tamaño con el upper bound(ub) del fichero solicitado. Si uno de los datafiles
es >= que el ub, se sabrá que la petición no ha sobrepasado el final del fichero.
De lo contrario, el valor retornado para los bytes leídos es calculado desde el
tamaño de la petición y el offset más grande de los datafile(el actual final del
fichero). Una vez terminada la acción, se retorna un valor donde se indica que la
acción se ha completado*/

return SM_ACTION_COMPLETE;
}

static PINT_sm_action io_cleanup(
        struct PINT_smcb *smcb, job_status_s *js_p)
{

     /*Este estado, es el encargado de inicializar los valores utilizados dentro
de la máquina de estado, por ejemplo la destrucción de los contextos mediante la
función io_conexts_destroy (io_contexts_destroy(sm_p);), el arreglo de índices de
datafile  (io_datafile_index _array_ destroy (sm_p) ),  los atributos del fichero (
PINT_SM_GETATTR_STATE_CLEAR(sm_p->getattr);), etc. y una acción de terminar indica
el fin de la máquina de estados del cliente (SM_ACTION_TERMINATE). Una vez
terminada la acción, se retorna un valor donde se indica que la acción se ha
completado*/

return SM_ACTION_TERMINATE;
}
```

4.2.2 PVFS_ isys_io.sm

Para que la máquina de estados del cliente pueda ser lanzada hacia los servidores de E/S se requiere de la función `PVFS_isys_io,` debido a que esta función gestiona las operaciones de lectura/escritura que el cliente solicita y ejecuta la máquina de estados. Esto es importante comentarlo debido a que gran parte de las modificaciones realizadas en el código de PVFS2 para la implementación de memoria cache en los clientes se ha realizada sobre esta función.

Una petición de lectura/escritura por parte del cliente a los servidores de E/S, se realiza a través de la función `PVFS_sys_io,` la cual a su vez llama a la función `PVFS_isys_io`. Esta función contiene una serie de parámetros que se inicializan con los datos necesarios para poder ejecutar una operación de lectura/escritura en los servidores. Algunos de estos parámetros son:

- offset_inicial(*`file_req_offset`*)
- Puntero al buffer local del cliente (**buffer*, El buffer local del cliente, es un espacio de memoria reservado para leer/escribir datos desde el cliente o bien escribir datos desde los servidores de E/S)
- Tipo de operación (*`io_type`*, lectura o escritura)
- Tamaño de la petición requerida (*`aggregate_size`*)

Estos valores se envían a la máquina de estados del cliente, quien se encarga de ejecutar las operaciones necesarias para el envío/recepción de los datos. En listado 4.3 se puede ver parte del código de la función `PVFS_isys_io` y una breve descripción de los parámetros que utiliza.

Listado 4.3 Código de la función PVFS_isys_io

```
    /* El siguiente código muestra como esta declara la función PVFS_isys_io,
quien se encarga de ejecutar las operaciones de E/S sobre pvfs2*/

     PVFS_error PVFS_isys_io(
     PVFS_object_ref ref,
     PVFS_Request file_req,
     PVFS_offset file_req_offset, /*int64_t*/
```

```
      void *buffer,
      PVFS_Request mem_req,
      const PVFS_credentials *credentials,
      PVFS_sysresp_io *resp_p,
      enum PVFS_io_type io_type,
      PVFS_sys_op_id *op_id,
      void *user_ptr)


       /* La estructura PVFS_object_ref, almacena el handle y el identificador del
sistema de fichero y otros parámetros para referenciar a un fichero, directorio o
enlace simbólico*/

     typedef struct{
             PVFS_handle handle; /*id_handle*/
             PVFS_fs_id fs_id; /*id_file_system*/
             int32_t    __pad1;
             } PVFS_object_ref;


     /* La estructura PINT_Request, es la encargada de almacenar  la información
necesaria para poder traer los datos correctamente que se han solicitado, un
ejemplo de estos valores son, el offsets, número de elementos requeridos, numero de
bloques, desplazamiento, tamaño de la operación, numero de trozos de datos
continuos,etc */

     typedef struct PINT_Request {
     PVFS_offset  offset;  /*offset de inicio del último elemento configurado */
     int32_t      num_ereqs; /*número of elementosrequeridosen un bloque */
     int32_t      num_blocks;   /*número de bloques*/
     PVFS_size    stride;       /*desplazamiento entre bloques*/
     PVFS_offset  ub;           /*límite superior*/
     PVFS_offset  lb;           /*límite inferior*/
     PVFS_size    aggregate_size; /*cantidad de datos solicitados*/
     int32_t      num_contig_chunks; /*número de trozos continuos*/
     int32_t      depth;    /*número de niveles anidados*/
     int32_t      num_nested_req;/*número de peticiones anidadas*/
     int32_t      committed;   /*indica si la petición ha sido comprometida*/
     int32_t      refcount;    /*número de referencias a esta estructura*/
     struct PINT_Request *ereq;  /*tipo de elemento*/
     struct PINT_Request *sreq;  /*tipo de secuencia*/
     } PINT_Request;


       /*la estructura PVFS_credentials, se encarga de almacenar el identificador
de usuario y de grupo dentro del sistema de ficheros*/

      typedef struct
      {
      PVFS_uid uid; /*user_ id*/
```

```
       PVFS_gid gid; /*group_ id*/
       } PVFS_credentials;

        /* la estructura PVFS_sysresp_io, almacena el resultado de una operación de
E/S (total de bytes leidos/escritos) */

       struct PVFS_sysresp_io_s
       {
       PVFS_size total_completed;
       }; typedef struct PVFS_sysresp_io_s PVFS_sysresp_io;

     /*  La  estructura  PVFS_io_type,  almacena  el  tipo  de  operación  de
entrada/salida, una lectura se representa con el valor de 1 y una escritura con el
valor de 2*/

     enum PVFS_io_type
      {
      PVFS_IO_READ  = 1,
      PVFS_IO_WRITE = 2
      };

     /* PVFS_sys_op_id y *user_ptr, son identificadores que se emplean cuando la
máquina de estados del cliente es ejecutada.*/

      PVFS_sys_op_id  *op_id  /*Holds  a  non-blocking  system  interface  operation
handle. */
      void *user_ptr

     /*Una vez que la función PVFS_isys_io, esta completada con la información
requerida una serie de instrucciones son realizadas con la finalidad de lanzar la
máquina de estados, quien será la encargada de leer/escribir los datos desde/hacia
los servidores*/

        /*Máquina de estados del cliente*/

     return PINT_client_state_machine_post(
            smcb,  op_id, user_ptr);


   }
```

4.2.3 Lecturas en PVFS2

Hasta este punto se ha hablado de la máquina de estados del cliente y de la función `PVFS_isys_io`, las cuales intervienen en las operaciones de E/S en PVFS2. A continuación se va a describir el procedimiento que se sigue en las lecturas y en las escrituras de PVFS2.

Esta descripción es un paso previo necesario para que se pueda comprender la modificación del código de PVFS2 realizada para la incluir de la implementación de memoria cache que aquí se presenta.

Cuando se solicita una operación de lectura se ejecuta la función `PVFS_isys_io` con los parámetros requeridos, indicándole en el campo `io_type` que el tipo de operación es de lectura; es decir, se pone el valor de este parámetro a 1. Una vez que la función es ejecutada se inicializa una serie de comprobaciones necesarias para realizar la lectura sobre un fichero. Algunas de estas comprobaciones son: validar el identificador de fichero (*handle*) y el identificador del sistema de ficheros (*fs_id*), verificar que el tipo de operación (*lectura/escritura*) se ha especificado correctamente. Así bien, concluidas estas comprobaciones, una estructura del tipo PINT_client_sm (estructura para la máquina de estados del cliente *sm_p*) es rellenada con los parámetros solicitados, y posteriormente la máquina de estados del cliente es lanzada hacia los servidores de E/S, los cuales son los encargados de devolver los datos del fichero al buffer local del cliente (*buffer*).La figura 4.2, ilustra una operación de lectura dentro de los clientes de PVFS2.

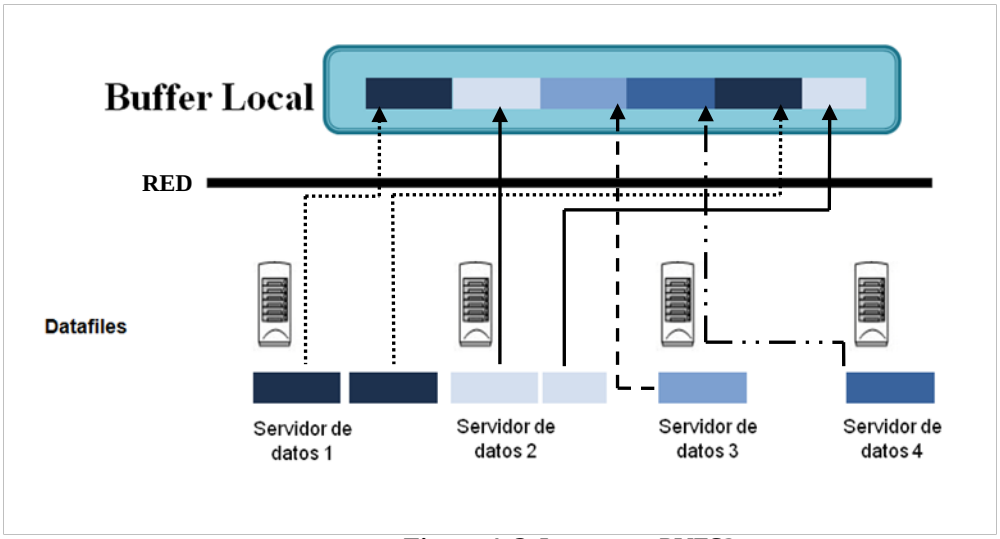


Figura 4. 2 Lectura en PVFS2

4.2.4 Escrituras en PVFS2

Las escrituras de datos contenidas en el buffer local del cliente, al igual que las lecturas, se ejecutan a través de la función `PVFS_isys_io`, solo que esta vez el valor correspondiente de `io_type` es 2. Después de ejecutar la función y validar el identificador de fichero (*handle*), el identificador del sistema de ficheros (*fs_id*), el tipo de operación (*escritura*) y de realizar alguna otra operación, una estructura del tipo `PINT_client_sm`(estructura para la máquina de estados del cliente *sm_p*) es rellenada con los parámetros necesarios para lanzar la máquina de estados del cliente que contiene un puntero al buffer local del cliente con los datos escritos que se van a enviar hacia los servidores de E/S. Los datos que se van a escribir se distribuyen en bloques (por defecto se utilizan bloques en el *striping* de 64KB) entre los dispositivos de almacenamiento en los servidores de E/S, empleando un método de distribución (por defecto, se usa Round-Robin). Cada fichero tendrá varios datafiles en cada uno de los servidores de E/S. La figura 4.3 ilustra cómo un fichero es enviado y distribuidos entre los servidores de E/S.

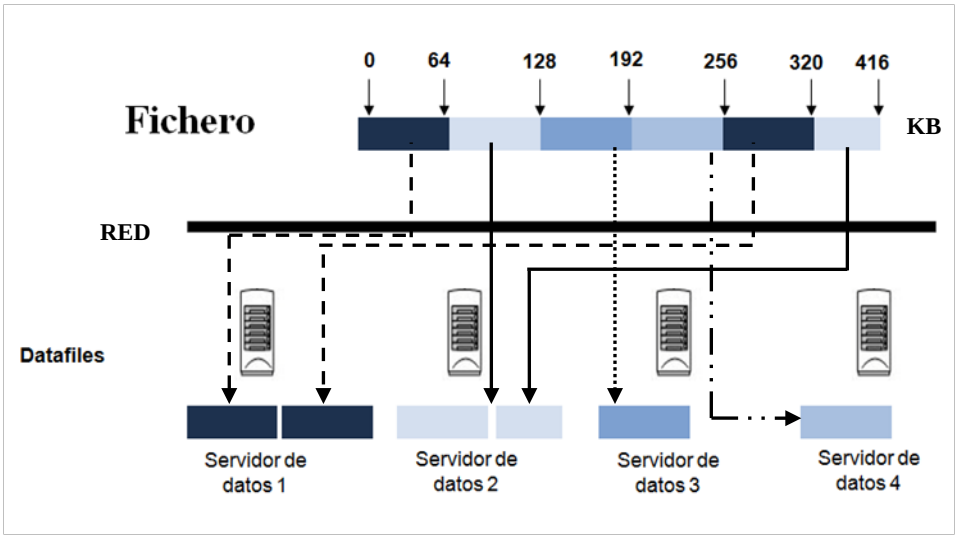


Figura 4. 3 Escritura en PVFS2

Con este apartado se concluye con la descripción de las operaciones de E/S en PVFS2. A continuación se muestra de forma detallada la implementación de cache para datos propuesta en los clientes de PVFS2.

4.3 Comparativa PVFS2 y ext3

Antes de empezar con la descripción de la implementación de memoria cache en los clientes de PVFS2 hay que mencionar que, como paso previo a esta implementación se hizo una comparación entre el sistema de ficheros PVFS2 y ext3. Inicialmente, se realizó un análisis y evaluación del sistema de ficheros PVFS2, cuya finalidad era la de tener la certeza de que el desarrollo de la implementación de memoria cache para datos en los clientes podía beneficiar significativamente al rendimiento del sistema, reduciendo las latencias e incrementando el ancho de banda en el envío/recepción de los datos.

Se llevó a cabo la evaluación de ambos sistemas donde los resultados obtenidos (Véase capítulo 6, Evaluación) nos permitieron deducir que sería eficiente añadir cache en los clientes de PVFS2. Con esta cache se podría esperar conseguir unas prestaciones más parecidas a las que se obtienen con ext3 cuando se accede a datos que se encuentran en cache del cliente.

4.4 Implementación de cache para datos en los clientes de PVFS2

En esta implementación de cache para datos en los clientes[Camacho10], se aprovecharon algunas de las características de PVFS2 (por ejemplo, maquinas de estados, funciones de accesos no contiguos, etc). La tabla 4.1, muestran las características principales de la implementación de cache en PVFS2.

Tabla 4.1 Características de Implementación de Cache en PVFS2

Principales características de la implementación
✓ Se implementa a Nivel de Usuario. No requiere modificación de la implementación en kernel de PVFS2. La caché se implementa en la memoria del cliente mediante la utilización de una memoria compartida por procesos.
✓ La búsqueda de los bloques se ha acelerado utilizando una tabla hash. ✓ En la implementación se usan listas enlazadas.
✓ Se utiliza LRU (*Least Recently Used*) como política de remplazo
✓ Los bloques de memoria caché tiene un tamaño por defecto de 4KB.

Esta implementación se realiza a nivel de usuario. De esta forma hay que atravesar menos capas de software para acceder a la cache desde las aplicaciones (Figura 4.4). Se utiliza bloques de cache de 4KB ya que la mayoría de las rutinas que asignan memoria emplean el tamaño de página. Las peticiones realizadas al sistema por parte de las aplicaciones se envían a la cache de datos. Los datos en una petición de las aplicaciones pueden englobar uno o más bloques, el primero y el último de los bloques no necesariamente completos.

La cache se encarga de verificar si todas las peticiones puedan ser atendidas completamente por la cache de datos del cliente, de ser así, en el caso de una lectura los datos son copiados desde la cache al buffer local del cliente. En una operación de escrituras los datos son escritos en cache. Así se evita enviar peticiones innecesarias a los servidores de E/S. Si dichas peticiones no pueden ser atendidas en su totalidad por la cache, se lanza la máquina de estados del cliente con la información modificada que se requiere para lazar una petición a los servidores de E/S correspondientes.

La cache de datos del cliente verifica los bloques (de 4KB) que pueden ser atendidos localmente y omite dichos bloques en la petición que se realiza a los servidores de E/S. Una vez que han llegado los datos solicitados en la petición, se almacenan dentro de la cache en bloques. La petición enviada al servidor de E/S engloba un número entero de bloques, pero no tienen que ser contiguos como se aclara más adelante. Los datos solicitados por la aplicación se pasan desde la cache al buffer local del cliente (apartado 4.2.2). En los apartados 4.4.1 y 4.4.2 se pueden encontrar más detalles.

En la implementación de cache de datos para PVFS2, los datos se almacenan en bloques que son reservados previamente en una memoria compartida. A la memoria compartida que conforma la memoria cache tienen que acceder distintos procesos. Los procesos, al contrario que las hebras, no pueden compartir memoria sin intervención del sistema operativo. Para permitir que distintos procesos en el cliente puedan acceder a la cache se ha utilizado los mecanismos que ofrece Linux para permitir que varios procesos puedan mapear un segmento de su espacio de direcciones virtual en un mismo segmento de memoria física. El acceso a este espacio de memoria compartido requiere utilizar funciones de sincronización. Los cambios que un proceso realice a los valores almacenados en esta memoria compartida serán visibles a todos los procesos que la comparten. En el apéndice A, en la sección memoria compartida, se explica detalladamente la creación y manipulación de una memoria compartida por procesos.

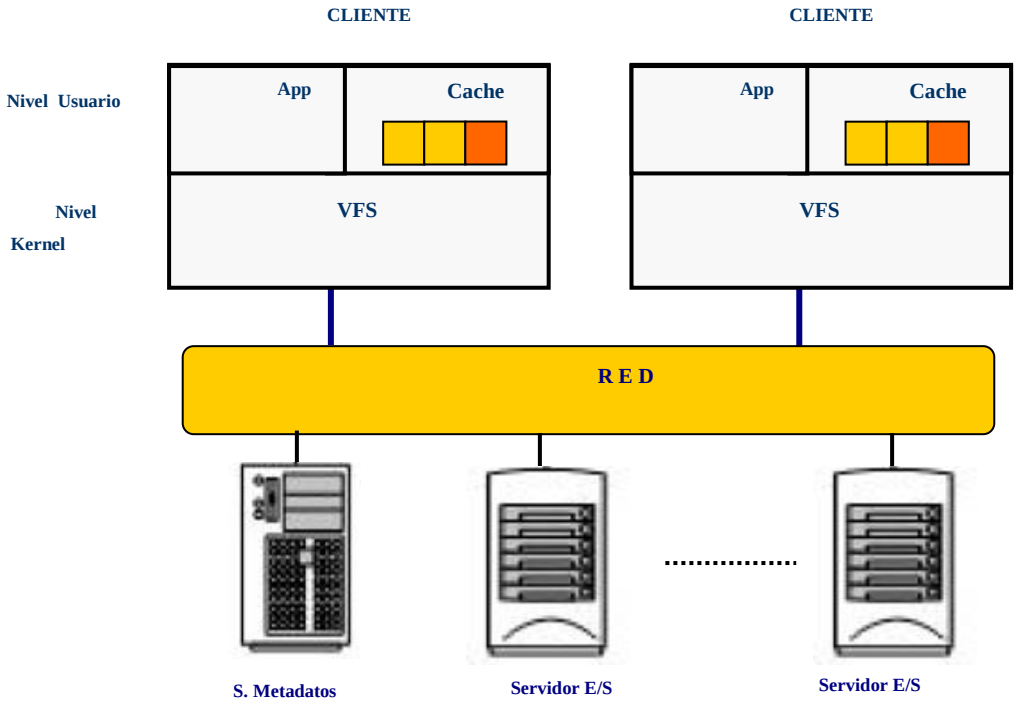


Figura 4. 4 Diseño de Implementación de cache en PVFS2

El siguiente fragmento de código (listado 4.4) se ha utilizado para la creación de la zona de memoria compartida utilizada en la implementación de memoria cache en los clientes de PVFS2. La creación de la memoria compartida se especifica dentro del fichero `\pvfs-2.7.1\src\apps\kernel\linux\pvfs2-client-core.c`. Dentro de este fichero encontramos que se crea el proceso (cliente) que accederá al sistema de ficheros PVFS2. De esta manera se concluyó que sería conveniente añadir nuestro código en este nivel, para que la creación de la memoria compartida se hiciera a la par con el cliente.

Listado 4.4 Creación de la memoria compartida en PVFS2

```
/*Declaración de estructuras para reservar la memoria cache*/
#define SEMKEYL ((key_t) (9998))/*key to semaphore*/
#define KEYL ((key_t) (9999))  /*key to share memory*/

#define SEGSIZEL (sizeof (hash))
#define HASH_LIST 2048      /* Tamaño de la table hash*/
#define FREE_LIST  4096     /* Tamaño de la lista de bloques libres */
#define CACHE_SIZE 4096/* Tamaño de la memoria cache*/
#define BLOCK_SIZE_CACHE 4096 /* Tamaño de bloque( por defecto 4KB)*/
```

```
     /*Definición de estructuras para reservar el espacio en la memoria compartida
*/typedef struct data
     {
     char data[BLOCK_SIZE_CACHE];
     }data;

     /*Definición de estructura para almacenar los metadatos */
     typedef struct metadata
     {
     int64_t id_block;
     int modify;
     int32_t  dirty_bit; /*Etiqueta que verifica si el bloque esta marcado como
sucio*/
     uint32_t  block_state;  /*Estado de bloque (por ejemplo si se encuentra
almacenado en cache*/
     int ban;
     struct data *cache;
     PVFS_object_ref ref_cache;
     struct list_head list_meta;
     struct list_head list_lru;
     struct list_head list_lru_dirty_bit;
     }metadata;

     /*Definición de estructura para alamacenar la cabecera a los metadatos*/
     typedef struct hash_list
     {
     struct list_head head_meta;
     }hash_list;

     /*Definición de estructura para asignar la memoria compartida*/
     typedef struct hash
     {
     int32_t flag;
     uint32_t pos_in_cache;
     struct hash_list hash[HASH_LIST];
     struct metadata meta_cache[FREE_LIST];
     struct data cache[CACHE_SIZE] __attribute__((aligned(64)));
     struct list_head head_lru;
     struct list_head head_lru_dirty_bit;
     int64_t stored_ptr;
     }hash;

     /*Declaración de la memoria compartida*/
```

```
hash* hash_memoria(hash *hash_ptr)
{
/*Declaración de identificadores de memoria compartida */
 int a,k,id;
void *new_ptr=0,*stored_ptr=0;


/*creación del segmento de memoria compartida*/
if((id = shmget(KEYL, SEGSIZEL + 16, IPC_CREAT | IPC_EXCL| 0777))==-1)
{
 if((id=shmget(KEYL,SEGSIZEL + 16,0777))==-1)
 {
 FILE *shm_get;
shm_get=fopen("/root/Escritorio/file/Fallo_shmget","w");
 fprintf(shm_get,"\nFallo al crear SHMGET = %d",id);
 fclose(shm_get);
 }


/* Vinculación de la memoria compartida*/
 if((new_ptr=shmat(id,0,0))==(void*)-1)
 {
 FILE *shm_mat;
shm_mat=fopen("/root/Escritorio/file/Fallo_shmmat","w");
 fprintf(shm_mat,"\nFallo al crear SHMMAT = %p",new_ptr);
 fclose(shm_mat);
 }


 stored_ptr=*(void **)new_ptr;
/*Borrado de la memoria compartida*/
 shmdt(new_ptr);
 new_ptr = shmat(id,stored_ptr,0);
 hash_ptr =(hash *) (new_ptr + 16);
 }else /*si la memoria compartida no existe*/
        {
/*Vinculación de la memoria compartida*/
new_ptr = shmat(id,0,0);
   *(void **)new_ptr = new_ptr;
  hash_ptr =(hash *) (new_ptr + 16);


/*Control de la memoria compartida*/


shmctl(id, SHM_LOCK, (struct shmid_ds *) NULL);


/*Inicialización de otros parámetros, por ejemplo,  la tabla hash y las
listas LRU */
```

```
     if(hash_ptr->flag==0)
         {
          INIT_LIST_HEAD(&hash_ptr->head_lru);
          hash_ptr->head_lru=hash_ptr->head_lru;
          INIT_LIST_HEAD(&hash_ptr->head_lru_dirty_bit);
                        hash_ptr->head_lru_dirty_bit=hash_ptr->head_lru_dirty_bit;

           for(k=0;k<HASH_LIST;k++)
        {
          INIT_LIST_HEAD(&hash_ptr->hash[k].head_meta);
          hash_ptr->hash[k].head_meta=hash_ptr->hash[k].head_meta;

        }

           for(a=0;a<FREE_LIST;a++)
        {
                            list_add_tail(&hash_ptr->meta_cache[a].list_lru,
&hash_ptr->head_lru);
                            list_add_tail(&hash_ptr-
>meta_cache[a].list_lru_dirty_bit, &hash_ptr->head_lru_dirty_bit);
          hash_ptr->meta_cache[a].dirty_bit=0;
          hash_ptr->meta_cache[a].id_block=0;
       hash_ptr->meta_cache[a].ban=0;
          hash_ptr->meta_cache[a].cache=&hash_ptr->cache[a];

        }
     hash_ptr->flag+=1;
       }
             }
     /*retorna la dirección que apunta a la memoria compartida*/
     return hash_ptr;
     }
```

Se ha implementado una política de remplazo LRU. Con esta política, si la cache está llena y hay que almacenar un nuevo bloque, se remplaza por el nuevo bloque el bloque menos utilizado. Si el bloque a remplazar tiene su bit sucio (modificado) activo, se envía su contenido al servidor de datos correspondiente, antes de añadir el nuevo bloque. Si la cache no estuviera llena, simplemente se introduce el nuevo bloque en un espacio libre. En el siguiente listado se puede ver el código relacionado con la LRU, este código se ha introducido en la función

`PVFS_isys_io` (Sección 4.2.2) y es utilizado en al momento de añadir un nuevo bloque en la memoria cache.

Listado 4.5 Lista LRU en la Implementación de cache en PVFS2

```
     /*La estructura meta_cache (tipo de estructura metadata, descrita en el
listado 4.4), almacena los metadatos referentes al bloque. Primero es necesario
obtener la posición de la lista LRU donde  escribiremos.*/


       meta_cache = meta_insert_lru(lhash_cache);
       /* Se obtiene la posición de la LRU*/


      /*Verifica el estado del bloque*/
      if(meta_cache->dirty_bit==1)


     Si el bloque está marcado con el bit de modificación o sucio, indica que el
bloque se encuentra ocupando esa posición modificada y es necesario hacer un
volcado de los datos antes de añadir un nuevo elemento.

     En el caso de que el dirty_bit (bit sucio o modificado) sea distinto de 1,  se
procede a añadir el nuevo elemento en la lista LRU. La siguiente función es
empleada para añadir un nuevo elemento en la LRU.

     /*Agregar un nuevo elemento en la lista LRU*/
      metadata* meta_insert_lru(hash *lhash_cache_datos)
      {
      metadataPtr newPtr=NULL;
      struct list_head *curr_pos;


     /* 1.- Obtener y Regresar un puntero de tipo metadata que este al inicio de
LRU */
      /* 2.- Una vez que obtenemos la dirección, hay que eliminarlo de la cabeza de
LRU */
     /* 3.- Esta nueva dirección obtenida desde la LRU, se inserta en cola de la
LRU */

      /* Obtenemos la dirección del ""primer"" elemento de la LRU*/
      list_for_each(curr_pos, &(lhash_cache_datos->head_lru))
           {
             newPtr = list_entry(curr_pos,struct metadata,list_lru);
      return newPtr;
      }
return newPtr;
}
```

Cuando un cliente realiza una operación de lectura/Escritura a un fichero, esta es recibida por un gestor de cache el cual se encarga de descomponer la operación en bloques (`Size_Operación/4096`), la búsqueda de los bloques se realiza mediante una tabla hash. De esta forma se reducen los tiempos de búsqueda y acceso a los datos. El listado 4.6 muestra el código para obtener una posición dentro de la tabla hash.

Listado 4.6 Tabla hash en la Implementación de cache en PVFS2

```
      …….
      …….
     /*Función dirección, devuelve la posición dentro de la tabla hash*/

      int64_t direccion(int64_t potition)
      {
       int64_t p;
        p=(int64_t)potition%HASH_LIST;
      return p;
      }


      …….
      …….

     /*Operación modulo para adquirir una nueva posición*/
     posicion=direccion(x);

     /*Se ejecuta a continuación la función de búsqueda que usa la hash. A esta
función se le pasan  los siguientes parámetros: posición del buffer, un puntero a
la memoria compartida, un identificador de fichero, credenciales, tamaño de la
operación y el tipo de operación en caso de ser  una lectura o una escritura. Esta
función se encarga de buscar los bloques que han sido previamente solicitados, y su
función es devolver los bloques que han sido encontrados en cache.*/

      int64_t search_cache(void *bufferptr,hash *lhash_cache_datos,
       PVFS_object_ref *ref_handle,
       const PVFS_credentials *credentialsprt,
       int64_t file_req_offset,
       int64_t agg_size,
       int *io_type_cache)


     //Inicialmente se revisa si la tabla hash contiene elementos
     list_empty(&(lhash_cache_datos->hash[posicion].head_meta))==1)


     //En caso de contener elementos, se procede a la búsqueda del bloque.
     /*Devuelve el bloque a buscar*/
```

```
     meta_cache = list_entry(curr_pos3,  struct metadata, list_meta);

     /*Una vez obtenida la posición de los bloques solicitados se mueven al final
de la lista LRU, para que quede constancia  que son ahora los bloques más
recientemente usados*/

      list_move_tail(&(meta_cache->list_lru), &(lhash_cache_datos->head_lru));/
```

Si los datos se encuentran en cache estos se copian directamente al buffer local del cliente, en caso contrario se crea una lista de bloques no encontrados. Esta lista indica al cliente qué bloques debe solicitar a los servidores de E/S.

El listado 4.7 muestra el código de la función empleada para crear la lista de bloques no encontrados, esta función se encuentra dentro de la función `search_cache`, la cual es llamada dentro de la función `PVFS_isys_io`. Esta función tiene parámetros que indican el número de bloques no encontrados, offsets, así como el tipo de operación que se desea ejecutar, entre otros. Una particularidad de esta función es que almacena los datos en el buffer local del cliente de manera temporal (espera en el caso de las lecturas que los datos sean recibidos desde los servidores de E/S o si es una escritura, hasta que los datos sean escritos en cache), permitiendo disminuir las asignaciones de memoria. En el caso de que la petición sea menor al tamaño de la estructura usada para esta función, un espacio de memoria es reservado de manera independiente. Una vez que los datos están listos para escribirse en cache una función de eliminación de bloques no encontrados es ejecutada con la finalidad de liberar el espacio que será necesario para escribir los nuevos datos.

Listado 4.7 Función para la lista de bloques no encontrados en la implementación de cache en PVFS2

```
     /*Función para insertar los bloques no encontrados*/

     No_Blocks_insert(No_BlocksPtr *sPtr, int64_t value, int64_t offsetaux,int64_t
final_offsetx,char   *bufferptr,int64_t   num_blocks2,int64_t   *offsetptr,   struct
metadata *meta_cache_noblocks,int *io_type_cache_aux)
     {
```

```
      No_BlocksPtr newPtr, previousPtr, currentPtr;

      if(num_blocks2!=1)
      {
      if(value==1)
                {
          posi3=offsetaux-offsetaux;
         }else if(value>1)
        {
         posi3=(offsetaux-*offsetptr);

        }
       }else if(num_blocks2==1)
       {posi3=offsetaux-offsetaux;
       }

     if( ((final_offsetx-offsetaux)< sizeof(No_Blocks)) || io_type_cache_aux==1 ||
io_type_cache_aux==2)
     {

     newPtr =malloc(sizeof(No_Blocks));
     }
     else{
     newPtr =(void *)&bufferptr[posi3];

     /*se asigna la dirección para el nuevo elemento*/

     }

     if(newPtr != NULL)   /* existe espacio en el buffer local del cliente*/
     {
     /*Asignamos a la estructura los data del bloque faltante y al apuntador next
es igual a NULL*/
     newPtr->offset_block=offsetaux;
         newPtr->final_offsetblock=final_offsetx;
     newPtr->bloque=value;
         meta_cache_noblocks->modify=1;
         newPtr->nextPtr=NULL;

     /*previousPtr y currentPtr se utilizaran para ir saltando entre la lista de
elementos*/

                 previousPtr = NULL;
                 currentPtr = *sPtr;
```

```
        /*Iniciamos el recorrido a traves de la lista para buscar el espacio en
donde insertaremos la nueva estructura*/


     while(currentPtr != NULL && value > currentPtr->bloque)
             {
                      previousPtr = currentPtr;
                      currentPtr = currentPtr->nextPtr;
     }


       /*Verifica si la lista está vacía. En caso de estarlo se asigna el inicio
de la lista a newptr*/

     if(previousPtr == NULL)
         {
                 newPtr->nextPtr = *sPtr;
                 *sPtr = newPtr;
                 }else
      {
              previousPtr->nextPtr = newPtr;
     newPtr->nextPtr = currentPtr;
                     }
            }else
      {
      printf("no puede ser insertado, no existe espacio en memoria");
      }
     return 0;
     }

     /*Función para eliminar la lista de bloques no encontrados*/

int64_t No_Blocks_del(No_BlocksPtr *sPtr, int64_t value, int64_t offsetaux,int64_t
final_offsetx,int *io_type_cache_aux)
{
No_BlocksPtr tempPtr;

     /*Preguntamos que si el valor a eliminar es el primero de la lista*/
if (value == (*sPtr)->bloque)
   {
    if( ((final_offsetx-offsetaux)< sizeof(No_Blocks)) || io_type_cache_aux==1 ||
io_type_cache_aux==2)
 {
```

```
/*asignamos el primer elemento de la lista a tempPtr para posteriormente
eliminarlo*/
   tempPtr = *sPtr;


      /*Hacemos al segundo nodo de la lista el primer elemento*/
 *sPtr = (*sPtr)->nextPtr;
         free(tempPtr); /*liberamos el elemento tempPtr*/
 }else{
      /*Hacemos al segundo nodo de la lista el primer elemento*/
      *sPtr = (*sPtr)->nextPtr;                }
 }
return 0;
}
```

La figura 4.5 ilustra las estructuras de datos empleadas en la implementación de memoria cache para clientes de PVFS2.

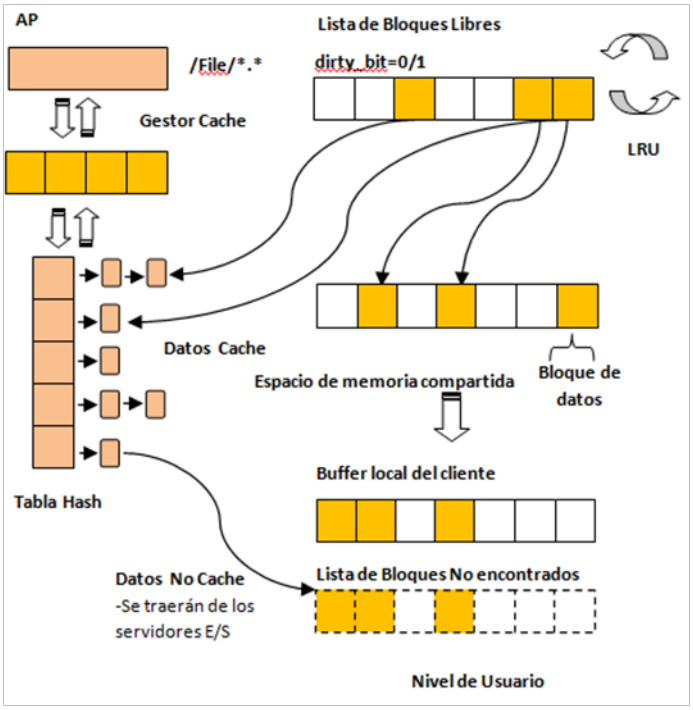


Figura 4. 5 Estructuras de datos para la implementación de memoria cache de PVFS2

4.4.1 Lecturas de la cache de datos en PVFS2

Hasta este punto se ha descrito las principales características de la implementación mediante fragmento de código en donde se explica las funciones y estructuras que han sido necesarias para realizar la implementación de memoria cache en los clientes de PVFS2.

En los siguientes apartados describiremos cómo se ejecutan las operaciones de E/S en nuestra implementación y los posibles casos que pueden presentarse. Como primera parte se describe la forma en que las lecturas se realizan, posteriormente se describe como llevan a cabo las escrituras por parte de las aplicaciones y por último cómo vuelca (envía) los datos a los servidores de E/S.

Las peticiones de lectura hacia el sistema por parte de las aplicaciones son enviadas a la cache de datos. Inicialmente el gestor de cache se encarga de verificar los datos que pueden ser atendidos por ésta.

El código del Listado 4.8 muestra cómo se lleva a cabo una operación de lectura cuando hay cache en PVFS2. La implementación de memoria cache hace uso de la función `PVFS_isys_io` (para más detalle sobre la función puede mirar el apartado 4.2.2). A esta función se le ha añadido un nuevo parámetro: un puntero al inicio de la memoria cache

Listado 4.8 Operación de lectura cuando los datos están en la implementación de cache en PVFS2

```
     /* Función que gestiona las operaciones de lectura/escritura que el cliente
solicita, y ejecuta la máquina de estados */

        PVFS_isys_io( PVFS_object_ref ref, PVFS_Request file_req, PVFS_offset
file_req_offset, void *buffer, PVFS_Request mem_req, const PVFS_credentials
*credentials,PVFS_sysresp_io *resp_p, enum PVFS_io_type io_type, PVFS_sys_op_id
*op_id,void *user_ptr, hash *lhash_cache)

      /*hash *lhash_cache, puntero al inicio de la memoria compartida*/
     /*Posteriormente se obtienen los atributos del fichero, mediante la función
PVFS_sys_getattr.*/

     PVFS_sys_getattr(ref, PVFS_ATTR_SYS_ALL_NOHINT
```

```
                (PVFS_credentials *) credentials, &getattr_response,

      lhash_cache);

     /*Posteriormente se continúa con una función de búsqueda (search_cache). Para
el caso de que la petición solicitara bloques completos, en caso de localizar todos
los datos solicitados, estos son llevados al buffer local del cliente mediante la
función memcpy, que se ejecuta en un ciclo de acuerdo al número de bloques
requeridos(Tamaño_operación/Tamaño_bloque_cache(4KB por default)), esto es porque
los datos no se almacenan de manera continua en la cache. El siguiente fragmento de
código describe cómo se realiza este procedimiento*/

     list_for_each(curr_pos3, &(lhash_cache_datos->hash[posicion].head_meta))
     {
     /*devuelve el bloque a buscar*/

     meta_cache = list_entry(curr_pos3,  struct metadata, list_meta);
     ……
     ……
     /*compara si el bloque solicitado corresponde al bloque almacenado en cache*/

     if((ref_handle->handle  ==  meta_cache->ref_cache.handle)  &&  (bloquer  ==
meta_cache- >id_block))
     ……
     ……
     /*Verifica que el tipo de operación es una lectura */

     if(io_type_cache== 1)
      ……
      ……

     /*Copiamos tantos bloques de 4096 bytes sean requeridos al buffer local del
cliente*/
     memcpy(&bufferptr[posi],meta_cache->cache->data,4096);
     ……
     ……
     /*No obstante puede presentarse el caso en que el tamaño de la petición del
primer y último bloque no corresponda al tamaño de bloque de cache (4KB por
default),  para este caso estos bloques copian solamente de la cache al buffer
local del cliente los datos solicitados.*/

     /*Comprueba si es el primer y último bloque*/

     if((offsetaux==0&&agg_size<4096)||(((final_offset2-offsetaux)!=0) &&
((final_offset2-offsetaux)<4096)))

      {

     /*Obtenemos la posición donde empezaremos a copiar de la cache*/
       pos2=((offsetaux)-((bloquer-1)*(4096)));
```

```
        residuo2 = final_offset2-offsetaux;

     /*Obtenemos la posición en el buffer local del cliente donde empezaremos a
escribir*/

       if(bloquer==1)
         {
          posi=0;
         }else if(bloquer!=1)
       {
       posi=(offsetaux-file_req_offset);
       }

     /*Copiamos el total de datos solicitados al buffer local del cliente(residuo2)
*/

      memcpy(&bufferptr[posi],&meta_cache->cache->data[pos2],residuo2);

  }

     /*se realizan los desplazamiento dentro de la LRU, indicando que los datos
previos son los recientemente leídos */

     list_move_tail(&(meta_cache->list_lru), &(lhash_cache_datos->head_lru));

      list_move_tail(&(meta_cache->list_lru_dirty_bit),&(lhash_cache_datos
      ->head_lru_dirty_bit));
     }
```

En el listado 4.8 hemos podido observar la manera en que los datos son tratados cuando se localizan en la cache de datos del cliente, sin embargo existe también el caso de que los datos no se encuentren en cache. Para acceder a estos dato en el caso de las lecturas continuas (véase la figura 4.9), se crea una lista de bloques no encontrados (`No_Blocks_insert`). Se calcula el total de datos continuos que pueden ser solicitados en una sola petición, el número de operaciones a realizar (cuando el primer y último bloque no corresponde al tamaño de bloque de cache, se cuentan como operaciones extras), y el inicio del buffer local. Una vez que los parámetros tienen la información necesaria, una máquina de estados modificada (offsets, aggregate_size, entre otros) con la información requerida es enviada a los servidores E/S para que se devuelvan los datos que han sido previamente solicitados. El siguiente listado 4.9 muestra algunos de los parámetros que se requieren modificar para que la operación se realice de forma correcta.

Listado 4.9 Operación de lectura cuando los datos no están en la implementación de cache en PVFS2

```
     /* Función que gestiona las operaciones de lectura/escritura que el cliente
solicita, y ejecuta la máquina de estados */

     PVFS_isys_io( PVFS_object_ref ref, PVFS_Request file_req, PVFS_offset
file_req_offset, void *buffer, PVFS_Request mem_req, const PVFS_credentials
*credentials,

      PVFS_sysresp_io *resp_p, enum PVFS_io_type io_type, PVFS_sys_op_id *op_id,
      void *user_ptr, hash *lhash_cache)
      {
       ……
                    sm_p->u.io.io_agg=agg; /*tamaño de la operación*/
                    sm_p->u.io.io_blo=blo; /*bloque inicial*/
                    sm_p->u.io.io_oper=oper; /*número de operaciones*/
                   sm_p->u.io.file_req_offset = file_req_offset;/*offset*/
       ……
                    sm_p->u.io.io_type = io_type; /*tipo de operación de E/S*/
                    sm_p->u.io.file_req = file_req; ; /*estructura file_req */
                    sm_p->u.io.mem_req = mem_req; /*estructura mem_req */
                    sm_p->u.io.buffer   =   &buffer[posi];/*dirección   donde   se
almacenan los datos*/
      ……
                    sm_p->object_ref = ref; /*estructura ref, almacena el id_file
system y el identificador del fichero(handle)*/

      /*Envió de la máquina de estados*/

      PINT_client_state_machine_post( smcb,  op_id, user_ptr);
      }


     /*Una vez que los datos son devueltos al buffer local del cliente se almacenan
en la cache en bloques de 4KB por default. Esto se logra mediante un ciclo que
calcula en cuantos bloques de 4KB se descompondrá la operación. Se obtiene la
posición en la que será escrito el primer bloque y se empieza a realizar las copias
mediante la función memcpy.*/

     While (x<y)
     {
     cache_datos = obtenemos la posición donde empezaremos a escribir;
     memcpy(cache_datos, buffer_local_cliente,4096);
     ……
     }
     x++;
     }
```

Cuando los datos no están en cache y la petición de lectura es menor que el tamaño de bloque de cache, se trae el bloque completo desde el servidor de E/S. Es decir, la máquina de estados del cliente es lanzada, y se ejecuta de manera anidada (apartado 4.2.1) la máquina de estados para peticiones pequeñas (listado 4.2) con información que indica que la operación será de 4KB y que el buffer destino será el de la cache del cliente. Posteriormente sólo se copian los datos solicitados por la aplicación desde la cache al buffer local del cliente (recuerde que las peticiones son menores que el tamaño de un bloque). En caso de que los datos se localizan en la cache, éstos son copiados desde la cache al buffer local del cliente.

Es importante mencionar lo que ocurre cuando el número de bloques a copiar en cache excede el tamaño de la misma. Únicamente se copia el número de bloques que caben en ella. Es decir los bloques iníciales son descartados dentro de la lista de bloques no encontrados, esto es debido a que se utiliza una política de remplazo, donde los bloques menos utilizados son los primeros en ser remplazados (LRU, apartado 4.4), se ha optado esta medida para no generar carga de trabajo y tráfico en la red innecesario, ya que disminuye en el rendimiento del sistema. A continuación se muestra mediante figuras los posibles casos que pueden existir en la implementación de cache cuando se realiza una operación de lectura

Cuando se presenta una operación menor de 4KB, y los datos se localizan en cache, solamente se leen los datos requeridos y se copian al buffer local del cliente. Por ejemplo si se requiere leer parte de un fichero, supongamos del byte 3000 al 5000, esto implicaría copiar trozos de 2 bloques correspondientes al fichero que se encuentra almacenado en cache, el primer trozo seria de 1096 bytes y el segundo de 904 bytes, que juntos sumarian los 2000 bytes solicitados en la petición, la figura 4.6, ilustra este tipo de operación.

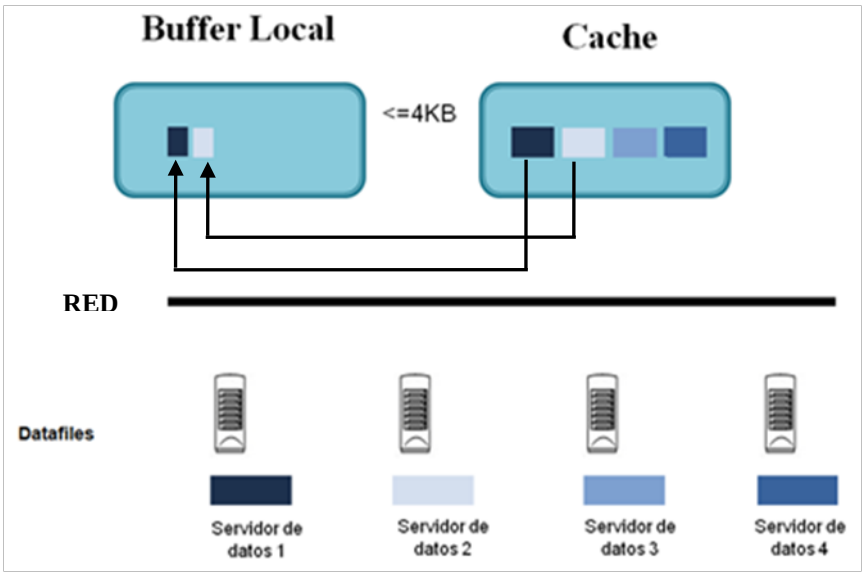


Figura 4. 6 Operación de lectura <4KB_datos_ cache

En el caso de que los datos no estén en cache (figura 4.7), volviendo al ejemplo anterior, en lugar de solicitar 2000 bytes al servidor se procede de la siguiente forma:

1) Se solicitan los bloques completos donde se encuentran los datos solicitados.

2) Se lee de la posición 0 hasta 8192 del fichero; es decir, se solicitan los dos primeros bloques completos

3) Se almacenan los dos bloques de 4KB en la cache.

4) Por último, se envían exclusivamente los datos solicitados al buffer local del cliente.

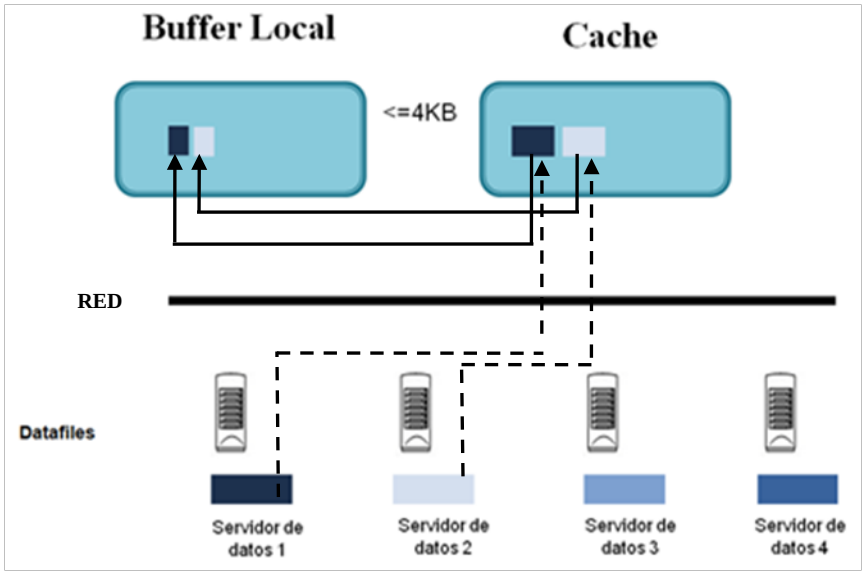


Figura 4. 7 Operación de lectura < 4KB_datos_no_ cache

A continuación los posibles casos cuando las operaciones son mayores o igual al tamaño de bloque en cache.

Para el caso de lecturas mayores o iguales a 4KB, se plantean los siguientes casos:

1) Los datos están almacenados en memoria cache (figura 4.8).
2) Los datos no se localizan en memoria cache (figura 4.9).

Para el caso de lecturas mayores o iguales a 4KB con datos que se localizan en cache, al igual que en el caso de las operaciones pequeñas, los datos se leen completamente desde la cache y son copiados al buffer local del cliente. Es decir por ejemplo, si se requiere leer una parte del fichero, supongamos del byte 0 al 24576, esto implicaría copiar 6 bloques de 4KB que se encuentra almacenado en cache. Juntos sumarian los 24576 bytes solicitados en la petición. La figura 4.8, ilustra este tipo de operación.

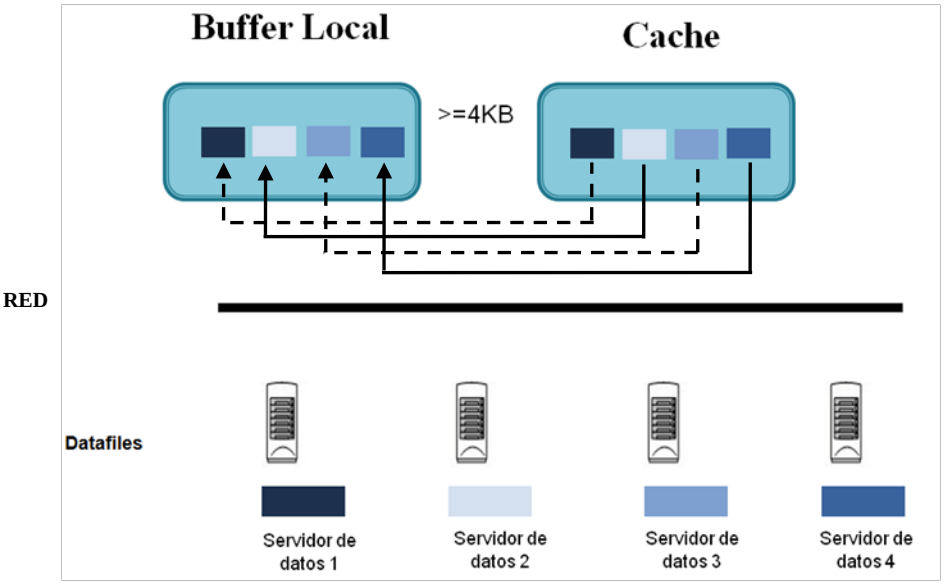


Figura 4. 8 Operación de lectura >= 4KB_datos_ cache

Si los datos no se encuentran en cache, la operación se ejecuta como lo hace PVFS2 sin esta implementación. Es decir, los datos son copiados directamente al buffer local del cliente, con la finalidad de reducir el número de operaciones que se realiza sobre la cache, ya que los datos pueden almacenarse en el buffer local del cliente de forma continua. Posteriormente, los datos son copiados del buffer local del cliente a la cache de datos. Figura 4.9.

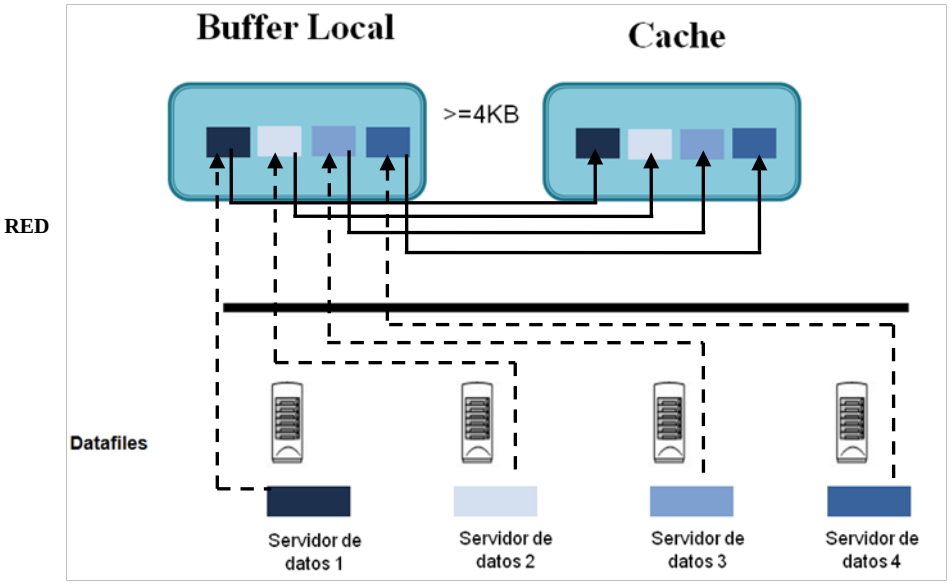


Figura 4. 9 Operación de lectura >=4KB_datos_no_ cache

También puede darse el caso de que las peticiones encuentren sólo algunos bloques en cache, como vimos anteriormente. Esto se soluciona enviando una petición como lo hace normalmente PVFS2 sin la implementación. Sin embargo surge una pregunta:

¿Qué sucede cuando los datos que se necesitan leer no están en posiciones contiguas?

Es decir, los datos que no están no ocupan posiciones contiguas en el fichero. Para este caso se ha empleado la función de PVFS2 `PINT_Request_hindexed`, la cual es rellenada de acuerdo al número de peticiones no continuas. De esta forma, los datos son traídos desde el servidor al buffer local del cliente en una sola máquina de estados. Lo anterior supone una ventaja para el procesamiento del sistema, ya que no es necesario estar realizando varias peticiones a los servidores E/S. Esto representa una disminución del tráfico en la red y una

mejora en el sistema. Posteriormente estos datos son copiados a la cache de datos del cliente. En Listado 4.10 se pueden ver los detalles del código implementado para este caso.

Listado 4.10 Operación de lectura mediante la función PINT_Request_hindexed de PVFS2

```
     /*Inicialmente se reserva el espacio que se utilizara para almacenar el offset
y   aggregate_size   de   los   bloques   que   serán   solicitados   en   la   función
PINT_Request_hindexed, esto mediante un malloc, que posteriormente es liberado bajo
la sentencia free(x)*/

     block_to_request_db.count_cache=number_block_to_request_db; /*Número de bloques
solicitados*/

     block_to_request_db.aggregate_size_cache=(int32_t*)malloc(sizeof(int32_t)*block
_to_request_db.count_cache);

     block_to_request_db.offset_cache=(int64_t
*)malloc(sizeof(int64_t)*block_to_request_db.count_cache);

     /*Oper_hindexed,  es  un  parámetro  que  almacena  el  número  de  operaciones  que
deben realizarse dentro de la máquina de estados, la siguiente condición compara que
oper_hindexed sea > 1, así la máquina de estados es lanzada con más de una operación
y por tanto se rellenan los offset y aggregate_size de cada de estas */

     if(oper_hindexed>1){

       block_to_request_db.offset_cache[operaciones_db]=offset_db;

      block_to_request_db.aggregate_size_cache[operaciones_db]=agg_size_db;

     oper_hindexed++;}

     /*Una vez que se tienen los datos de cada una de las operaciones. La función
PVFS_Request__hindexed es ejecutada y  se encargara de traer los distintos datos
solicitados de un mismo fichero en una sola máquina de estados, evitando que se
envíen cada operación en máquinas de estados diferente. Con esto se reduce el
procesamiento y el rendimiento del sistema incrementa*/

     PVFS_Request_hindexed(oper_hindexed,block_to_request_db.aggregate_size_cache,
        block_to_request_db.offset_cache,  PVFS_BYTE,&mem_req);

     PVFS_Request_hindexed(oper_hindexed, block_to_request_db.aggregate_size_cache,
       block_to_request_db.offset_cache, PVFS_BYTE, &file_req);
```

A continuación se describe las variables que se utilizan en la función PVFS_Request_hindexed:

- `Oper_hindexed`, Es un contador, y almacena el número de operaciones que deben realizarse.
- `block_to_request_db.aggregate_size_cache`, es el tamaño de la operación.
- `block_to_request_db.offset_cache`, es el desplazamiento donde inicia la operación.
- `PVFS_BYTE`, el tipo de dato que se utiliza.
- `mem_req` y `file_req`, estas variables son parte de la estructura `PINT_Request`, e indican que valores mantendrá cada variable en cada una de las operaciones a ejecutar. La estructura `PINT_Request` es la encargada de almacenar la información necesaria para poder traer los datos que se han solicitado, un ejemplo de estos valores son: el offsets, número de elementos requeridos, número de bloques, desplazamiento, tamaño de la operación, número de trozos de datos continuos, etc.

4.4.2 Escrituras en la cache de datos enPVFS2

En el apartado anterior se analizó los posibles casos para las operaciones de lectura en la cache de datos en PVFS2. Ahora toca el turno a la operación de escritura en la cache de datos en PVFS2. En el texto se usan fragmentos de código para explicar las funciones y estructuras que han sido necesarias para realizar la implementación de memoria cache en los clientes de PVFS2 cuando una operación de escritura es ejecutada por las aplicaciones que hacen uso del sistema. En los siguientes párrafos de esta sección describiremos como se resuelven los posibles casos que se presentan cuando una escritura es realizada y como los datos son volcados (enviados) a los servidores de E/S.

La escritura de datos en la implementación de cache en PVFS2, se reciben por el gestor de cache. Se pueden presentar dos casos:

- Si los bloques a modificar se encuentran almacenados en la cache, se procede a su modificación y se activa el bit de cambio o modificación en cada uno de ellos.

- Si los bloques no se encuentran en cache, se escriben en la cache y se activa su bit de modificación.

Como se mencionó anteriormente, una operación de escritura se lleva a cabo con la función `PVFS_isys_io`. Al igual que las lecturas se sigue una serie de pasos; por ejemplo, la obtención de atributos del fichero mediante el uso de la función `PVFS_sys_getattr`, así como la búsqueda de bloques asociados a las posiciones que se van a modificar utilizando la función `search_cache`. En esta última función cambia un parámetro con respecto al caso de las lecturas (`io_type_cache`), este parámetro nos permite definir que el tipo de operación a realizar es una escritura.

Para el caso de una operación de escritura mayor o igual que 4KB, inicialmente los datos son buscados en la cache del cliente mediante la función `search_cache`, esta función realiza la búsqueda de los datos en bloques de 4KB (tamaño de bloque por defecto) en la cache, mediante la tabla hash. Si los datos se encuentran almacenados en la cache, los nuevos datos son copiados desde el buffer local del cliente a la cache en bloques de 4KB modificando los datos existentes. Un bit de modificación o sucio se pone a 1dentro de los atributos de cada uno de los bloques. Véase la figura 4.10.

El listado 4.11 muestra el código correspondiente cuando se realiza una operación de escritura >=4KB y los datos están almacenados en la implementación de cache en PVFS2.

Listado 4.11 Operación de escritura cuando los datos están en la implementación de cache en PVFS2

```
     /*Inicialmente los datos son buscados en cache, mediante un ciclo que se
ejecuta de acuerdo al tamaño de la operación/tamaño de bloque en cache, si la
operación es menor al tamaño de bloque de cache por lo menos se ejecuta una vez y
una la operación es tratada en base a su tamaño (esto se explica en la sección de
escrituras menores a 4KB.)*/

     list_for_each(curr_pos3, &(lhash_cache_datos->hash[posicion].head_meta))
     {

     /*Devuelve el bloque a buscar*/
     meta_cache = list_entry(curr_pos3,  struct metadata, list_meta);
     ……
     ……
```

```
      /*compara si el bloque solicitado corresponde al bloque almacenado en cache*/
      if((ref_handle->handle   ==   meta_cache->ref_cache.handle)   &&   (bloquer   ==
meta_cache- >id_block))
      ……
      ……

      /*Verifica que el tipo de operación es una escritura */
      if(io_type_cache== 2)
        ……
        ……

      /*Copiamos los datos en bloques de 4096 bytes como sean requeridos a la cache
del cliente*/
      memcpy(meta_cache->cache->data,&bufferptr[posi],4096);
      ……
      ……
      /*indica que el bloque ha sido modificado*/
      dirty_bit=1;
        ……
        ……
      /*se realizan los desplazamiento dentro de la LRU, indicando que los datos
previos son los recientemente escritos */
      list_move_tail(&(meta_cache->list_lru), &(lhash_cache_datos->head_lru));
      list_move_tail(&(meta_cache->list_lru_dirty_bit),&(lhash_cache_datos
       ->head_lru_dirty_bit));
      }
```

En la Figura 4.10 se ilustra lo que ocurre si la operación es mayor o igual al tamaño de 4KB y los datos están en cache. En este caso se modifica el bloque o los bloques en cache que se desea y se marcan con el bit de modificación a 1.

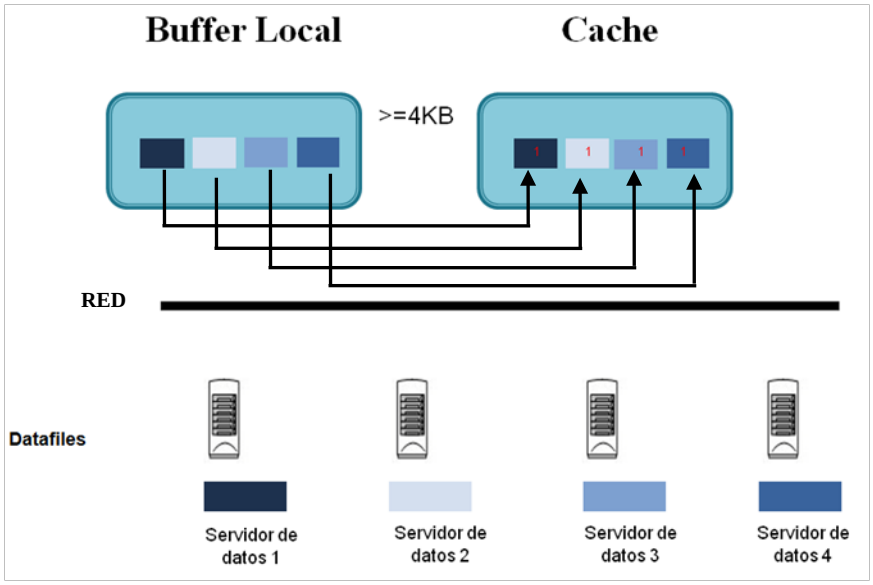


Figura 4. 10 Operación de escritura>=4KB_cache

No obstante, también se presentan peticiones en donde uno o varios bloques solicitados no se encuentran almacenados en cache. Para resolver esta situación, se genera una lista de bloques no encontrados y posteriormente se obtiene la dirección (bloque de cache) en donde los datos serán almacenados dentro de la cache, para ello utilizamos la función `meta_insert_lru`. Antes de realizar la escritura del nuevo bloque se comprueba que la posición obtenida de la cache esté marcada con el bit de modificación (`dirty_bit`) igual a 0. Si el valor de `dirty_bit` está puesto a 0, la operación se realiza de manera similar que la operación anterior, es decir como si los datos estuviesen en la cache. Se activa su `dirty_bit`. Véase la figura 4.11, 4.12, 4.13. El listado 4.13, presenta el fragmento de código utilizado para realizar una escritura mayor o igual que 4KB cuando los datos no están almacenados en la cache de datos.

Listado 4.12 Operación de escritura cuando los datos no están en la implementación de cache en PVFS2

```
     /*Inicialmente si los datos no se encuentran almacenados en cache una lista de
bloques no encontrados es generada(No_Blocks_insert)con los parámetros necesarios
para la identificación de ese bloque en particular. Posteriormente se obtiene la
posición donde se escribirá ese nuevo bloque dentro de la cache mediante el uso de
la función meta_insert_lru*/

     meta_cache = meta_insert_lru(lhash_cache);


     /*Una vez obtenida la posición donde se almacenaran los datos dentro de la
cache, se procede a verificar que la posición a ocupar este libre*/
if(meta_cache->dirty_bit = 0)

     /*Se asigna la posición que ocupara dentro de la tabla hash*/
     psaux=ref.handle*block;

     /*Operación Modulo para adquirir una nueva posicion*/

     posicion=direccion(psaux);

     /*Se comienza con la copia de los datos hacia la cache*/

     memcpy(meta_cache->cache->data,&buffer[pos_in_cache],4096);

     /*Posteriormente se va realizando el desplazamiento dentro del buffer local del
cliente para indicar desde que posición comenzara a copiar*/

     pos_in_cache=pos_in_cache+4096;


     /*Después se activa el bit de modificación del bloque. Y los metadatos
necesarios para la identificación del bloque dentro de la cache son completados, por
ejemplo ref. handle(identificador del fichero), id_block (identificador de bloque),
etc.*/
```

```
      meta_cache->dirty_bit=1;
      meta_cache->ref_cache.handle=ref.handle;
      meta_cache->ref_cache.fs_id=ref.fs_id;
      meta_cache->ref_cache.__pad1=ref.__pad1;
      meta_cache->id_block=block;

      /*Finalmente se realizan los desplazamiento dentro de la LRU, indicando que los
datos previos son los recientemente escritos */

list_move_tail(&(meta_cache->list_lru),(&lhash_cache->head_lru));

list_move_tail(&(meta_cache->list_lru_dirty_bit),(&lhash_cache-
>head_lru_dirty_bit));
```

Si los datos no están almacenados y además las posiciones de cache a utilizar devueltas por la LRU se encuentran con el `dirty_bit` activado, es necesario realizar un flush o volcado de los datos existentes a los servidores de E/S antes de que sean remplazados por los nuevos datos (Figura 4.11). Una vez concluido el flush de los datos, el valor de `dirty_bit` es puesto a 0 (Véase la figura 4.12). Este nuevo valor nos indica que el bloque puede ser sobrescrito (más detalle sobre el *flush* en el apartado 4.4.3).

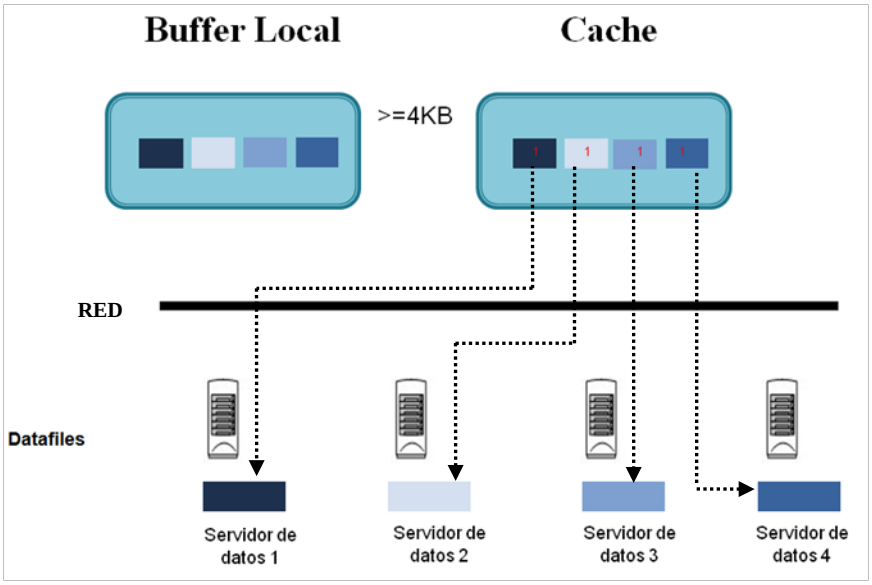


Figura 4. 11 Operación de escritura>=4KB y dirty_bit = 1_no_cache

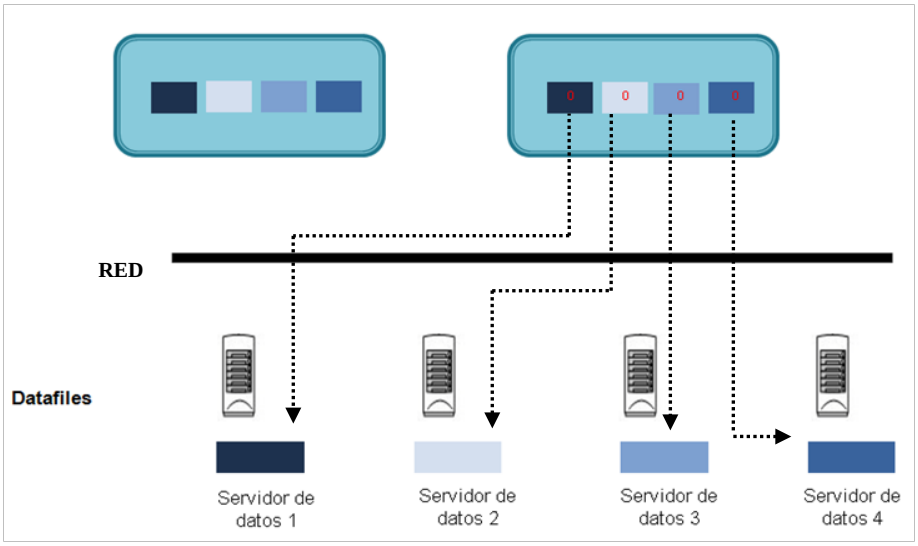


Figura 4. 12 Operación de escritura>=4KB y dirty_bit = 0_no_cache

Si la operación es mayor o igual al tamaño de 4KB y `dirty bit` es 0. Los datos se copian directamente en cache y el valor de `dirty_bit` es puesto a 1 (Figura 4.13)

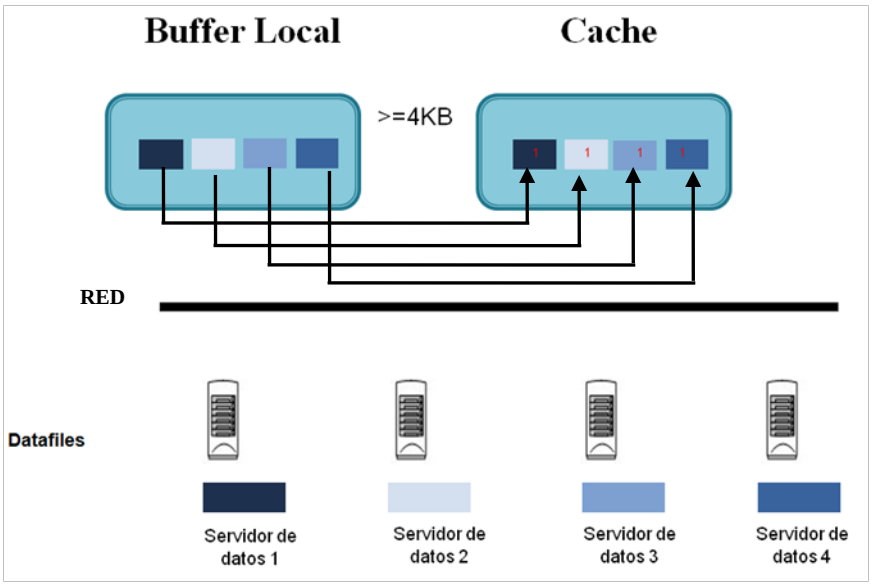


Figura 4. 13 Operación de escritura>=4KB_no_cache

Si hay una petición menor al tamaño de bloque de la cache (<4KB) y los datos que se desean modificar están almacenados dentro de la cache, se modifica el trozo correspondiente y el bit de modificación de los bloques implicados se ponen a 1 (obsérvese que pueden estar implicados uno o dos bloques). En caso de que los datos no se localicen en cache, se emitirá una operación que lea los bloques (4KB) de los servidores de E/S. Estos se encargarán de enviar los datos pertenecientes a los bloques solicitados al cliente. Estos bloques se almacenarán en cache. Una vez concluida la operación de lectura, se vuelve a la operación original, en este caso, una escritura menor a 4KB. Para completarla se modifica la parte requerida en la petición sobre la cache, y se activa el bit de modificación de los bloques implicados. Las figuras 4.14 y 4.15 ilustran una operación de escritura menor que 4KB.

En la figura 4.14 se ilustra lo que ocurre si la operación es menor del tamaño de 4KB y los datos están en cache. En este caso se modifica el bloque o los bloques en cache que se desea y se marcan con el bit de modificación a 1.

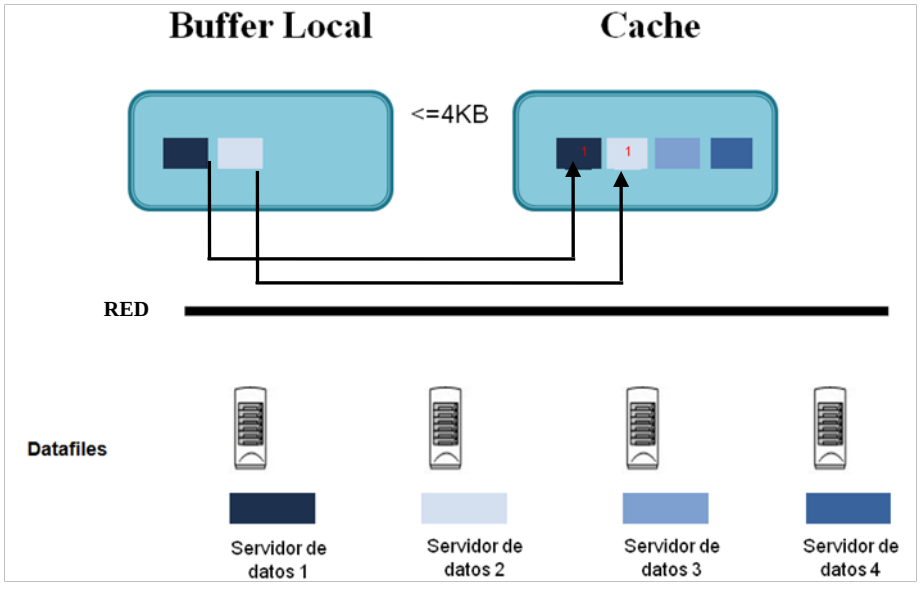


Figura 4. 14 Operación de escritura <4KB_cache

En la Figura 4.15 se ilustra el caso de que los datos no están almacenados en cache. En este caso se realiza una operación de lectura, que se encarga de traer los bloques de 4KB correspondientes a la cache. Posteriormente se escriben los datos requeridos desde el *buffer* del cliente a la cache y el bit de modificación de los bloques implicados se ponen a 1.

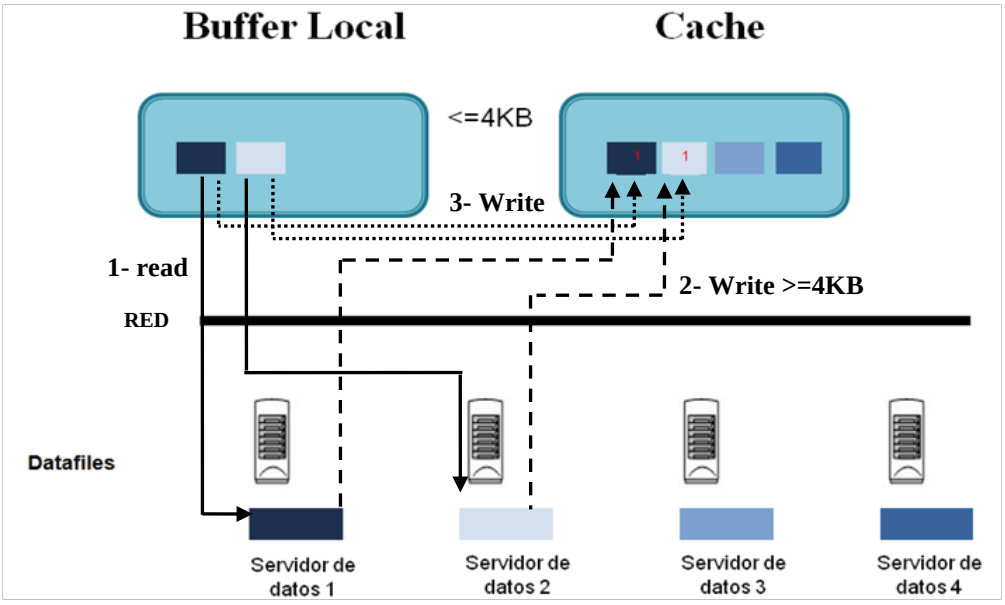


Figura 4. 15 Operación de escritura <4KB_no_cache

4.4.3 Volcado de datos al servidor de E/S

Durante el funcionamiento de la cache será muy probable que se tengan que remplazar algunos bloques almacenados en la memoria caché por otros que sean recientemente solicitados por el cliente. Si estos bloques que se remplazan se han modificado, hay que almacenarlos en los servidores de datos para que no se pierda su copia más reciente. Por este motivo, se ha implementado una función de volcado o flush de bloques de la cache a servidores E/S.

A continuación se enumeran los casos que pueden presentar cuando se recibe una petición de Lectura/Escritura (Figura 4.16).

1. Cuando se realiza una operación de E/S, los bloques asociados a la petición son buscados en cache. Si los datos se localizan en cache (es decir los bloques de datos corresponden a los solicitados en la petición y pertenecen al mismo fichero) y los bloques que ocupan se encuentran marcados con el bit modificación a 1, la operación se realiza directamente en cache y no se requiere momentáneamente realizar ningún cambio en los servidores de E/S, de esta forma los datos quedan en espera de ser

copiados a los servidores de E/S cuando se le indique o sea necesario. En caso de que el bit de modificación estuviese en 0, la operación se realiza de la misma manera y el valor del bit de modificación es puesto a 1.

2. Cuando se van a escribir datos nuevos que no se encuentran almacenados en cache, y no hay bloques libres en ésta, se genera una lista de bloques no encontrados, y posteriormente se obtiene mediante la *LRU* las posiciones a utilizar en cache para estos nuevos bloques. Supongamos como primer caso que las posiciones a utilizar en cache se encuentran marcadas con el bit de modificación igual a 1. Inicialmente se calcula cuantos bloques de cache serán requeridos para realizar la operación solicitada por la aplicación. Una vez realizado el conteo de los bloques necesarios para escribir en cache, se verifica cuales bloques están marcados como modificados o sucios. Los bloques marcados como modificados que se van a ocupar por bloques "nuevos" se deben volcar a los servidores correspondiente. Al estar los bloques de cache marcados como modificados, es necesario hacer el *volcado* o *flush* de N bloques de cache sean requeridos y su bit de modificación se debe limpiar (poner a 0).

3. Puede ocurrir que los bloques a remplazar no sean todos del mismo fichero. En este caso se deben realizar varias operaciones de volcado (*flush*), en concreto una por fichero y el valor del bit de modificación se pone a 0.

4. En caso de que los bloques a remplazar formen parte de un mismo fichero, pero los datos almacenados no pertenecen a bloques contiguos, al igual como se explicó en el apartado 4.4.1 se utiliza la función de PVFS2 `PINT_Request_hindexed`. Salvo que en esta ocasión en lugar de leer los datos desde los servidores de E/S, estos son escritos en los dispositivos de almacenamiento de los servidores de E/S. En el caso de que los datos no sean del mismo fichero, se hace lo mencionado en el punto 3.

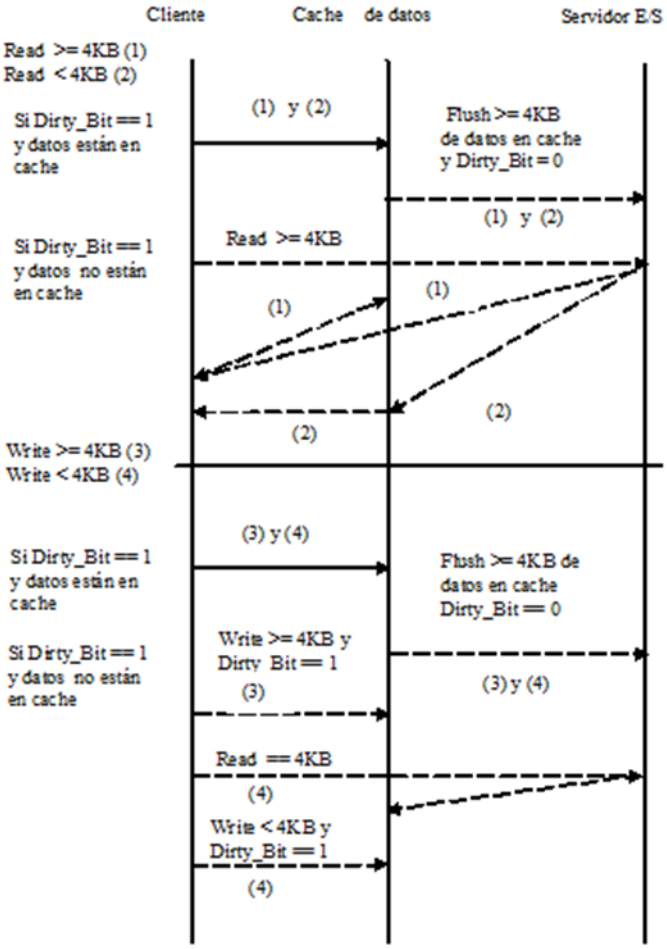


Figura 4. 16 Esquema de volcado o flush de datos de la cache en PVFS2

En la implementación realizada el mantenimiento de coherencia de la cache es responsabilidad del programador. La implementación ofrece al programador una función flush para enviar explícitamente los bloques modificados a los servidores de E/S, el programador debe usar esta función flush para enviar los datos a los servidores de E/S cuando haya bloques en la cache con el estado de modificado (`dirty_bit` a 1) para que la copia que utilice el último cliente que solicita los datos sea la más actual. En la función se especifican los datos que serán actualizados en los servidores de E/S.

Al no usar protocolo de mantenimiento de coherencia las operaciones de E/S son más rápidas que usando un protocolo. No obstante, se está trabajando en la elaboración de un protocolo de mantenimiento de coherencia para PVFS2 y en esta memoria se presenta una implementación de cache que mantiene coherencia para el sistema de ficheros AbFS. El

capítulo 5 muestra esta implementación de cache propuesta en AbFS y el capítulo 6 la evaluación de las implementaciones de cache realizadas en PVFS2 y en AbFS.

Capítulo 5

Implementación de Memoria Cache en AbFS

SUMARIO

Capítulo 5

Implementación de Memoria Cache en AbFS

En este capítulo se detallada la propuesta de cache implementada en AbFS. Inicialmente se describe la relación entre los elementos internos en clientes y servidores (sección 5.2), dispositivos virtuales (sección 5.3), cache de datos de AbFS (sección 5.4), los modos de funcionamiento de AbFS (sección 5.5). Se hace la descripción del control de coherencia de cache interna de AbFS (sección 5.6). Posteriormente se hace referencia a la forma en que AbFS gestiona los metadatos (sección 5.7). Por último se describe la gestión de extents en memoria en AbFS (sección 5.8)

5.1 Introducción

En [Diaz11a] se describe AbFS (*Abierto File System*) como un sistema de ficheros distribuido que permite que se compartan los dispositivos de almacenamiento local de bajo coste que tienen los computadores de un cluster. AbFS se ha implementado en el núcleo de sistema operativo (capítulo 3, sección 3.6). La implementación de la gestión de metadatos de AbFS mezcla en la misma estructura el espacio de nombres y los atributos, y combina hash, tablas, estructuras jerárquicas y caches. Con esta última combinación evita los problemas que presentan las implementaciones basadas en hash y las basadas en tablas.

AbFS usa caches de metadatos y de datos en los clientes. Las caches comparten el mismo protocolo de mantenimiento de coherencia y aprovechan algunas de las estructuras de cache del VFS (*Virtual File System*) de Linux reduciendo, de esta forma, la complejidad añadida. Usar caches mejora las prestaciones porque reducen el tiempo de acceso a los datos por estar más cerca y, adicionalmente, también porque disminuyen los accesos al servidor y, por tanto, su congestión. En AbFS, las caches reducen la congestión de los servidores y de las caches del servidor también cuando se abre un fichero para escritura. El primer nodo que abre un fichero para escritura es el propietario del fichero. Se consigue reducir la congestión porque los clientes que escriben en el fichero combinan las escrituras en la cache del propietario en lugar de ir al servidor y porque los clientes leen, cuando hay propietario, de la cache del propietario en lugar de leer del servidor. En esta situación, es decir, cuando hay propietario, la cache tiene unas prestaciones similares a la cache cooperativa basada en home de [Hwang05]. Como se puede deducir AbFS añade comunicaciones cliente-cliente y servidor-servidor a las comunicaciones cliente-servidor utilizadas en sistemas de ficheros basados en el modelo cliente-servidor, como NFS.

5.2 Relación entre elementos internos en clientes y servidores

Para que el sistema ofrezca tiempos de respuesta rápidos es vital contar con elementos que queden almacenados en cache de forma que permita que los nodos puedan tener localmente información sin necesidad de consultar continuamente al servidor. Estos elementos

pueden cambiar en el tiempo y por lo tanto es necesario que exista un modelo de coherencia que garantice cuándo los datos son válidos en cada momento.

Cada nodo, sea cliente o servidor debe almacenar información de:

- **Inodos:** Son las unidades principales de almacenamiento ya que identifican cualquier elemento que se almacena (ficheros, directorios, enlaces, etc). Su información principal es:
  - **Atributos VFS:** Propiedades relacionadas directamente con VFS (permisos, UID, GID, tiempos, etc.). La mayor parte es accesible por VFS pero se gestiona por AbFS.
  - **Radix Tree:** Guardan la relación entre el desplazamiento dentro de un fichero y las páginas utilizadas para almacenar datos. Gestionada por VFS con funciones de AbFS..
  - **Información de servidor:** Información necesaria por AbFS cuando dicho inodo está almacenado en un servidor. (Qué clientes tienen copia, dónde está el inodo en disco, …)
  - **Información de extents:** En esta versión los extents se gestionan sólo por el servidor, pero está previsto que los clientes también puedan gestionarlos. Mantiene en memoria el árbol rojo-negro en uso (no todas las ramas) de la información de extents de un inodo (sólo para ficheros).
  - ***Información de transacciones:*** Cuando un nodo recibe operaciones de lectura/escritura en un fichero se inicia una transacción que debe completarse para saber cuándo puede liberarse un inodo y cambiar de estado. Estas transacciones son importantes ya que garantizan que las escrituras son atómicas para mantener el estándar POSIX de acceso a ficheros.
- **Dentry cache:** (*dcache*): Almacena los nombres de los elementos. Son todos los que están, pero no están todos los que son, es decir, se almacenan entradas dentro de un directorio, pero no estarán todas las entradas de un directorio.

- **File:** Ficheros abierto. Cada vez que se realiza una función *open* sobre un fichero el VFS gestiona información propia de un fichero (puntero actual de lectura escritura, fcntl, etc)

En la figura 5.1 se muestra la relación de dichos elementos en tiempo de ejecución.

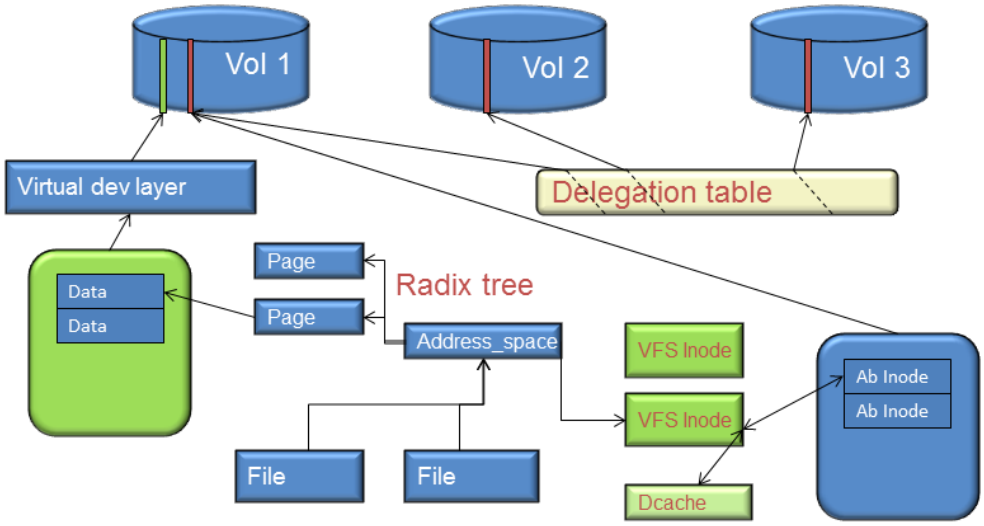


Figura 5. 1 Relación entre los elementos principales de AbFS en tiempo de ejecución

Una diferencia importante de AbFS con otros sistemas de ficheros en red es su carácter híbrido SAN/NAS. La mayor parte de sistemas de ficheros en red mantienen una filosofía NAS (Lustre, PVFS2, NFS,...), es decir, la relación entre clientes y servidores para los datos de los ficheros viene dada por algún identificador para dichos ficheros (número de inodo o handler) y la posición dentro de dicho fichero. Esto es debido a que utilizan otro sistema de ficheros local en un nivel inferior (ej:ext3, ext4) para el almacenamiento de los datos, por lo que dicha capa abstrae la localización física de los datos, quedando desconocida hasta para los propios servidores. Otros sistemas, como GPFS, mantiene un modelo SAN que obliga a los clientes acceder a nivel de bloque directamente con las unidades de almacenamiento, lo que conlleva un control de la localización exacta de los bloques de los ficheros.

AbFS plantea un modelo híbrido ya que los clientes pueden acceder directamente en modo bloque ya que conocen la localización de los bloques de un fichero y esto permite un acceso más rápido. AbFS no utiliza un sistema de ficheros local pues tiene su propia gestión de bloques utilizados por cada fichero, así como la gestión del espacio libre y ocupado de cada volumen. Esto implica un diseño más complejo ya que además del control de bloques implementa los dos modelos y la coherencia entre el acceso de ambos modelos. Para la gestión de bloques asignados a cada fichero se utiliza un modelo basado en extent. Un extent hace referencia a un conjunto de bloques contiguos, de forma que con pocos extents se puede codificar la localización completa de un fichero. El control de la reserva de espacio de bloques ocupados en AbFS tiene en sus propios mecanismos de cache en el servidor, y un sistema basado en una división de los volúmenes en superbloques con múltiples mapas de bits de zonas del disco con tamaños de ocupación de distinta granularidad, lo que permite reducir las operaciones de reserva de grandes bloques.

5.3 Dispositivo virtual

Trabajar en modo bloque ofrece múltiples ventajas ya que permite aprovechar algunos de los mecanismos del VFS de Linux de forma que los datos quedan almacenados en *buffers*. Las operaciones de lectura y escritura de bloques en dispositivos pueden optimizarse utilizando estos *buffers* ya que en el caso de las escrituras pueden quedar pendientes de completarse en el dispositivo (escrituras asíncronas) y en el caso de lecturas se puede acceder directamente a la memoria que guarda de forma temporal dichos datos. En los dispositivos de almacenamiento se utilizan sectores como unidad mínima, y un bloque se compone de uno o más sectores, por lo que pueden almacenarse varios bloques consecutivos en una página. El tamaño usual de página en Linux es de 4KB. Manteniendo este modelo en el que una única página puede contener uno o más bloques en la memoria y dado que el núcleo requiere cierta información de control asociada a dichos datos (cual es el dispositivo de bloques, el bloque específico, estado del buffer, descriptor de página), cada *buffer* se asocia con un descriptor mediante una estructura `struct buffer_head`.

AbFS necesita hacer referencia a la posición física de ciertos elementos y para ello utiliza 64 bits tratando de aprovechar al máximo dichos bits en función de los objetos que se

almacenan. Bloque es la unidad lógica que se utiliza para localizar datos de cierto tamaño en todo el sistema. Cualquier elemento, inodos, ficheros, etc., están situados en algún bloque. La unidad de trabajo en general es un bloque que, por defecto, mide 4KB. Este parámetro es común a todo el sistema de ficheros independientemente del número de bytes por sector de cada disco. Se podría trabajar con otro tamaño pero en principio no plantea limitaciones trabajar con dicho tamaño.

Un bloque se localiza por la tupla (número de volumen, número de bloque) que se codifica con 64 bits (Figura 5.2). A esta tupla la llamaremos ***dirección de bloque lógico***. Cada volumen está identificado por un número entre 1 y 4095 de 12 bits (el volumen 0 hace referencia al volumen donde está guardada dicha información). El resto de los bits hasta completar los 64 se utilizan para localizar el bloque dentro del volumen. Cada nodo, ya sea cliente o servidor, tiene una lista con todos los volúmenes, en la que se indica en qué nodo servidor se encuentra cada uno o, si el volumen es local, el dispositivo y partición asignados; por tanto, cualquier nodo puede localizar directamente un bloque, ya sea local o remoto.

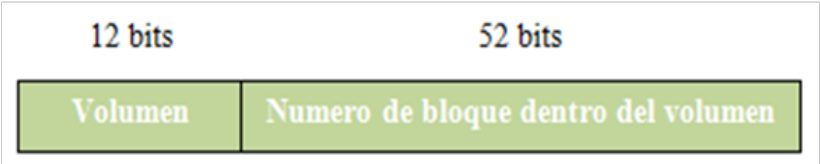


Figura 5.2 Localización de un bloque mediante la Tupla (Volumen, bloque) de 8 bytes

Ya que el tamaño de bloque está directamente relacionado con el número de volúmenes, para ampliar a más volúmenes debería aumentarse el tamaño de bloque. En esta versión los inodos y extents deben estar localizados en el propio volumen. Se utilizan directamente 64 bits de desplazamiento sobre el volumen. Se puede plantear una modificación basada en 12 bits para el volumen y 52 bits para indicar el resto, pero accesos a otros volúmenes fuera del nodo podrían ralentizar considerablemente el acceso. Toda la información global, es decir, la almacenada en disco o la que se transmite por red, está codificada en Little Endian porque está orientado a su uso en arquitecturas de Intel que utilizan este criterio de codificación. Aunque realmente sería muy fácil cambiarlo a Big Endian si resultara necesario por las macros que definen la representación de los datos en tiempo de compilación. La información local, la que

gestiona internamente cada nodo, se mantiene en el formato nativo de la arquitectura de cada nodo.

Cuando un cliente requiere un bloque se solicita a través del VFS que mantiene la cache de bloques del dispositivo (*buffers*). Si dicho bloque no está en memoria el VFS hace una llamada a AbFS solicitándolo mediante un descriptor de operación de entrada/salida llamado *bio.* AbFS analiza dicha petición y comprueba si la operación es local o remota. En la mayoría de los casos las operaciones son remotas ya que los clientes no comparten espacio de almacenamiento, pero cuando un nodo actúa como cliente y como servidor puede acceder como cliente a datos que están almacenados localmente, por lo que hay que realizar dicha operación de forma local. AbFS calcula el dispositivo físico y el bloque a partir de la tabla de volúmenes.

5.4 Cache de datos de AbFS

AbFS incluye varios niveles de caché para almacenar los datos basada en los *buffers* de Linux. Inicialmente está la cache del cliente y posteriormente está la cache del servidor. Cuando cliente y servidor son el mismo nodo (porque un nodo es servidor y está accediendo a datos que están almacenados en el almacenamiento local de dicho nodo,) la cache está unificada y no hay múltiples copias de los datos.

Cuando un cliente empieza a realizar escrituras, genera *buffers,* que al cabo de unos segundos empiezan a sincronizarse. Aunque la transferencia por red es bastante rápida, un cliente puede crear *buffers* mucho más rápidamente por lo que el cliente empieza a incrementar su memoria asignada a *buffers*. Cuando los servidores reciben los bloques, la capa de dispositivo virtual de AbFS empieza a crear buffers asociados a los bloques locales, lo que permite una respuesta rápida a los clientes. Los servidores completan sus escrituras de forma asíncronas de forma más lenta según la velocidad que permiten los dispositivos.

Estas caches intermedias permiten que muchos benchmarks muestren gran velocidad de transferencia incluso en escritura ya que absorben un volumen elevado de datos. En el caso de escrituras síncronas, las operaciones se consideran completas cuando se envían al servidor, por lo que la caché del servidor también ofrece un almacenamiento elevado. En sistemas de

ficheros locales, cuando se hacen escrituras síncronas en disco se hacen directamente al dispositivo, por lo que puede ocurrir que las escrituras síncronas en AbFS sean más rápidas que sistemas de ficheros locales debido a la caché de los servidores.

En el caso de las lecturas actúan igualmente los dos niveles de cache, en clientes y en servidores de forma que si un cliente accede a datos recientes tendrá los datos en sus propios *buffers* locales. La cache de los servidores también permite que lecturas de otros clientes puedan acceder directamente a los *buffers* del servidor acelerando las siguientes lecturas.

AbFS utiliza el mecanismo de lectura adelantada (*read a head*) de Linux lo que permite mejorar el acceso a ficheros optimizando las transferencias entre clientes y servidores

5.5 Modos de funcionamiento de AbFS

El intercambio de información entre clientes y servidores en AbFS puede resumirse en cuatro casos principales:

5.5.1 Caso I: Creación de ficheros y acceso

En gran cantidad de ocasiones un cliente crea ficheros y accede a ellos. En este caso se da prioridad a dicho cliente al acceso para que tenga el control total sobre ellos. El cliente accede a los bloques de dichos fichero de forma directa y mantiene una cache de los datos incluso en su propia memoria (*buffers*), lo que le permite tanto un acceso rápido a los datos que está creando como trabajar de forma asíncrona y posteriormente escribir de forma masiva dichos bloques optimizando así la transferencia con los servidores. La figura 5.3 muestra el caso I.

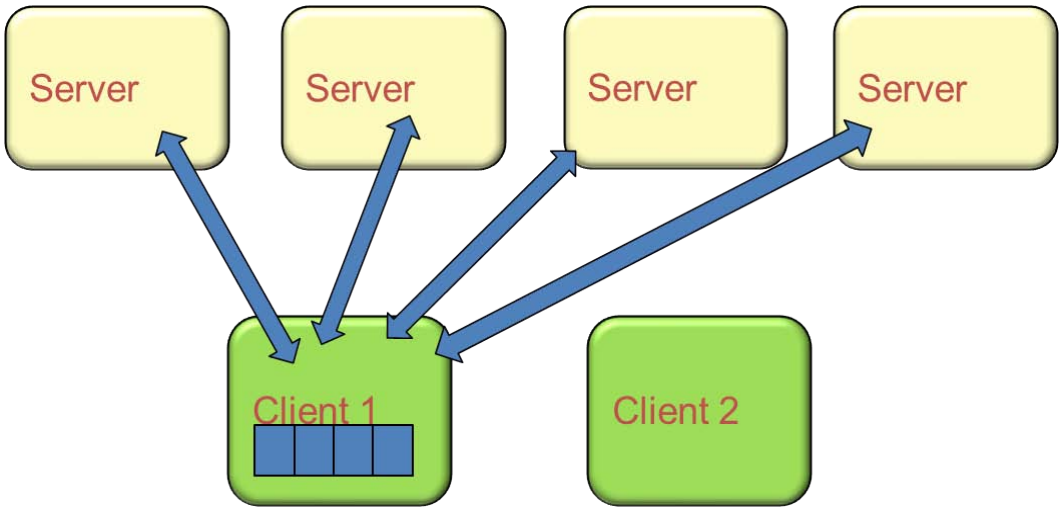


Figura 5.3 Caso I: Creación de fichero y acceso

Los servidores determinan la localización de los bloques asignados a los ficheros mediante extents. Los clientes acceden directamente a los servidores ya que AbFS distribuye los ficheros entre los servidores. Los clientes tienen la información sobre los extents de un fichero, lo que les permite acceder a los datos de forma rápida ya que conocen los bloques ocupados por dichos ficheros. Clientes y servidores también incorporan una cache de extents que se describe posteriormente.

5.5.2 Caso II: En modo lectura todos los clientes activan sus caches

Los servidores controlan en todo momento el uso de los ficheros, si están abiertos para lectura o escritura (o ambos) y por quién. En este caso los servidores saben que los datos no cambian y se pueden mantener múltiples copias de los datos en los clientes (*buffers*) de forma que acceden a nivel de bloque y el acceso de los clientes es local. Véase la figura 5.4.

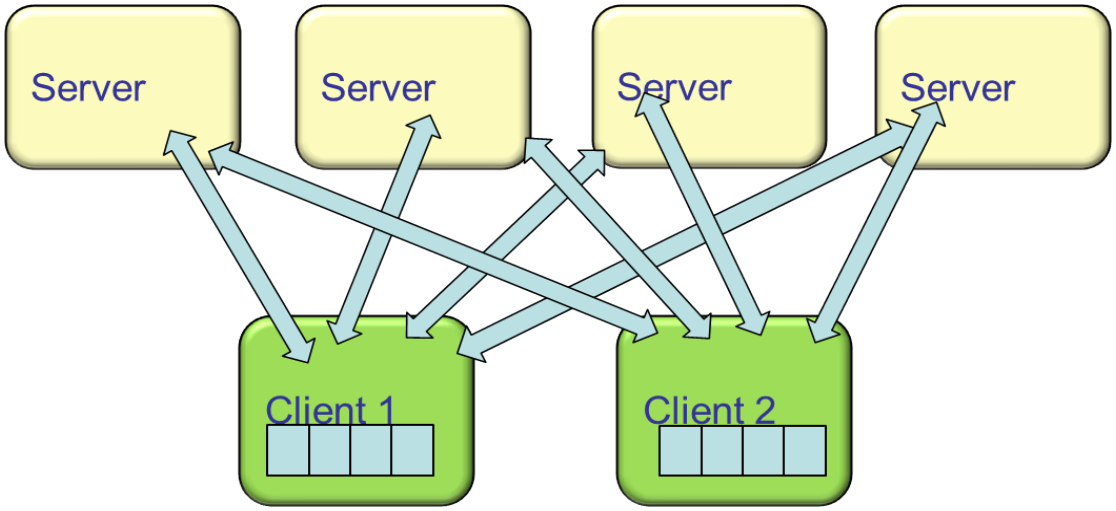


Figura 5.4 Caso II: En modo lectura todos los clientes activan sus caches

5.5.3 Caso III: Si un cliente intenta acceder a un fichero para escritura

En este caso un cliente intenta acceder a un fichero que ha sido creado previamente por otro cliente (Figura 5.5).

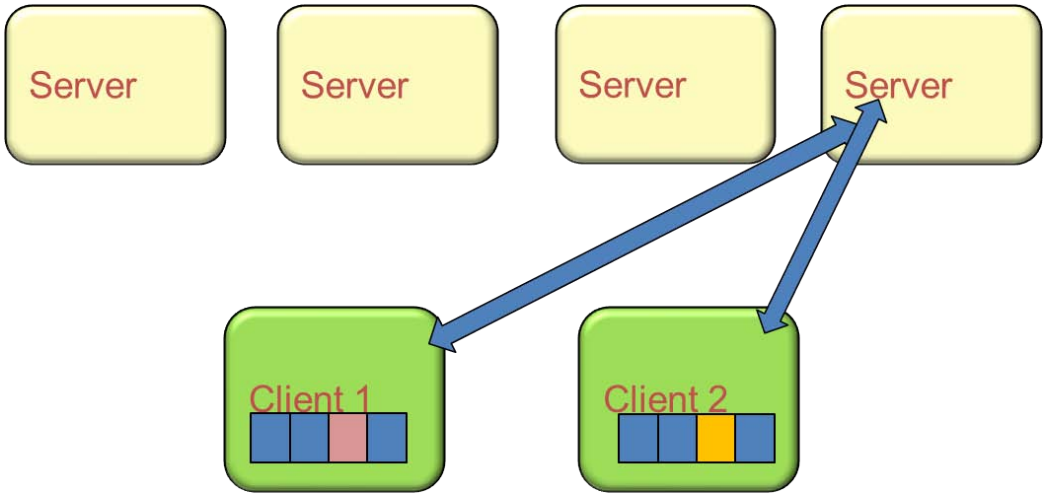


Figura 5.5 Caso III: Si un cliente intenta acceder a un fichero para escritura

En este caso el nuevo cliente toma la propiedad del fichero una vez que el anterior propietario sincroniza sus buffers en el servidor.

5.5.4 Caso IV: Otros nodos intentan acceder al nodo propietario

Cuando un cliente tiene la propiedad de un fichero toma el control completo y los demás clientes deben acceder a través de dicho nodo. Esto ralentiza las operaciones del resto de clientes pero garantizan el estándar POSIX de acceso a ficheros. (Figura 5.6)

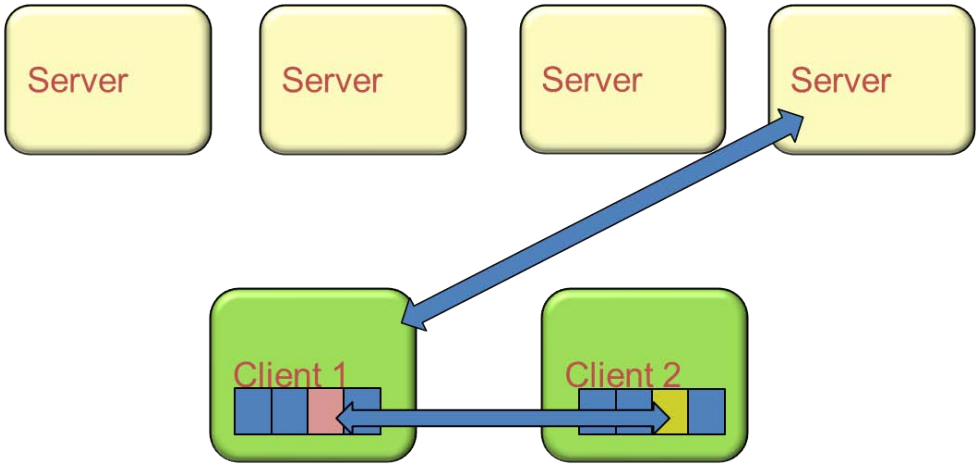


Figura 5.6 Caso IV: Otros nodos intentan acceder al nodo propietario

El cliente propietario mantiene su caché local y el resto de clientes desactivan sus caché.

5.6 Control de coherencia de cache interna de AbFS

La información que se guarda de cada elemento y las condiciones para mantener o descargar cada elemento en dichas tablas cambia de clientes a servidores. El problema de la coherencia no aparece sólo en el caso de los ficheros abiertos sino también en los inodos y entradas de directorios (*dentry*) utilizadas por clientes y servidores ya que también incluyen cache.

Los recursos almacenados internamente en el kernel en dichas caches incluyen elementos de bloqueo que garantizan el acceso a datos válidos y no en transiciones de estados que podrían provocar incoherencias. Se trata que haya gran cantidad de elementos independientes lo que permite ofrecer mejores tiempos de respuesta. Es el caso del acceso a dos inodos de forma simultánea en un SMP, el VFS sólo hace bloqueo en el caso del acceso al mismo fichero, pero en el resto no hace bloqueos. En AbFS el acceso para creación a dos inodos corresponderá, por probabilidad, a entradas distintas en la tabla de delegación, por lo

que prácticamente no hay conflicto en las dos operaciones y pueden realizarse simultáneamente en un servidor. A pesar de ello son necesarios los elementos de bloqueo que evitan incoherencias en dichos accesos, ya que aunque la probabilidad es baja cuando se crean gran cantidad de ficheros simultáneamente sí ocurre condiciones de riesgo.

Para simplificar el modelo de coherencia se trata de unificar estados en lugar de que existan estados clientes y servidor. Aunque cada estado puede tener información que le permita realizar acciones con distintos argumentos.

Se dice que un nodo es servidor cuando es responsable de un inodo. Dicho servidor puede abrir ficheros y en ese caso cuando actualiza lo hace sobre el disco. Cuando lo hace cualquier otro nodo, la actualización se hace al nodo que corresponda.

Los inodos tienen una variable, *last_jiffies*, que indica para clientes la última vez que recibieron información válida de un servidor o se revalidó. Esto permite que los inodos "mueran" transcurrido un tiempo y los servidores no tengan que preocuparse por todos los inodos que están en los clientes. Esta variable, en el caso de un servidor indica la última vez que ha sido utilizado y por lo tanto debe estar el inodo disponible por un tiempo antes de que desaparezca.

Se ha tratado de aproximar a un modelo en el que se fija cuando un cliente toma el control de un fichero (abierto para escritura) no se lo cederá a otro hasta que no se cierre el fichero (o todas las estructuras *file* relacionadas con el fichero abiertas para escritura).

A continuación se indica la lista de estados:

- **(C) (*Clear*) Vacío:** El inodo no existe en memoria (puede existir en disco) y por lo tanto no hay información en tiempo de ejecución válida. Corresponde con el estado inicial de todos los inodos.
- **(E) Exclusivo:** Los datos válidos sólo están disponibles en el nodo propietario. Los demás nodos no tienen copias válidas y estarán en estado Inválido.
- **(S) (*Shared*) Compartido:** Puede estar compartido por otros nodos.
- **(I) Inválido:** Los datos y metadatos no están válidos en la copia local.*I_owner* debe indicar el propietario, o bien 0 para indicar desconocido y preguntar al servidor (*i_server*).

Los directorios siempre están en estado (S) *shared* y no cambian de estado, aunque puede ocurrir que hayan expirado y por lo tanto su información no es válida. Si un directorio cambia sus atributos se envía un mensaje de invalidación. Este mensaje no cambia de estado el inodo sino que cambia *last_jiffies*, indicándole que ha expirado su información válida y por lo tanto si lo necesitara deberá volver a requerirlo al servidor.

Generalmente los inodos están en estado S. Los servidores guardan el ID de los nodos que han requerido el inodo (por defecto hasta 4 nodos). Cuando dicha lista se llena se activa una bandera indicando que hay más de 4 clientes que han requerido recientemente dicho inodo.

El estado (I) se utiliza para indicar que el inodo ya no tiene datos válidos y por lo tanto la función de revalidación devolverá falso. Si requiere el inodo en este estado los datos debe solicitarlo al propietario o al servidor.

El estado (E) se utiliza por el cliente que está accediendo a un fichero en modo escritura y por lo tanto tiene acceso exclusivo. Si un segundo nodo trata de abrir el fichero todas las operaciones se redirigen al nodo que tenga la propiedad del fichero.

Si el cliente ha modificado el fichero se activa la bandera para indicar que deberá actualizarlo al final, al cerrar el fichero. En el caso de un cliente, cuando se cierra el fichero se mantiene la propiedad por un tiempo hasta que se sincroniza el inodo y en ese momento el servidor retoma el acceso al fichero. Si un cliente tenía el fichero abierto en escritura al acceder a una siguiente lectura o escritura y comprobar que el nodo responsable ya no es el antiguo propietario, le puede solicitar al servidor el modo (E) exclusivo.

El estado (S) es el más frecuente ya que generalmente todos los nodos comparten información y las lecturas de inodos son muy frecuentes.

El estado (E) sólo pueden ocurrir en accesos a ficheros en escritura en clientes ya que es cuando el servidor puede delegar el inodo. En el caso que coincide que el cliente es el propio servidor del inodo aparecen dichos estados.

En el resto de casos, los servidores sólo tendrán los estados (S) o (I).

El estado (I) puede ocurrir tanto porque un cliente requiere el modo exclusivo o porque se ha borrado un inodo.

La siguiente figura 5.7 muestra los estados y transiciones entre estados:

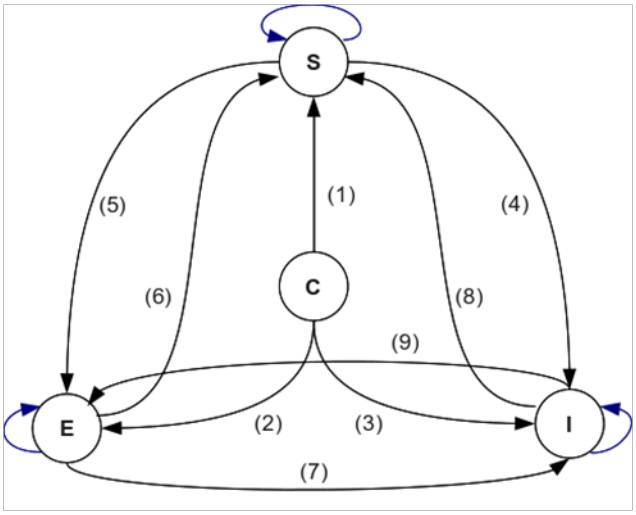


Figura 5.7 Esquema de los estados y transiciones entre estados

La figura 5.7 y la tabla 5.1 resumen el protocolo de coherencia de cache en el cliente. Aunque los bloques de la cache de buffer de dispositivo son de 4KBYTES, el bloque en el protocolo de mantenimiento de coherencia es un fichero (se aplica al inodo).

Tabla 5.1 Resumen del protocolo de coherencia de cache basado en propietario

Nº	Estado	Evento	Comentario	Sig. estado
1	(C)	Interno al nodo: lookup o dir-create	Inodo compartido	(S)
2	(C)	Interno al nodo: fichero create	Nodo gana la propiedad del inodo	(E)
3	(C)	Interno al nodo: lookup, o abre el fichero y está ya abierto para escritura en otro nodo	Nodo no gana la propiedad del inodo	(I)
4	(S)	Externo al nodo: invalidación. Interno al nodo: lookup, o abre fichero y otro nodo tiene la propiedad del mismo	Nodo no gana la propiedad del inodo	(I)
5	(S)	Interno al nodo: abre el fichero para escribir (1er nodo que lo abre para escribir)	Nodo gana la propiedad del inodo	(E)
6	(E)	Interno al nodo: cierra el fichero abierto para escritura	Inodo compartido y escritura al servidor	(S)
7	(E)	Servidor deniega propiedad	Excepción	(I)
8	(I)	Interno al nodo: lookup, o abre el fichero para leer	Inodo compartido	(S)
9	(I)	Interno al nodo: abre el fichero para escribir (1er nodo que lo abre para escribir)	Nodo gana la propiedad del inodo	(E)

A continuación se describe la implementación de la gestión de metadatos y de las caches en el cliente.

5.7 Gestión de metadatos en AbFS

En [Diaz11a, Diaz11b] se describe AbFS como un sistema de ficheros simétrico; es decir, los computadores en una plataforma con AbFS pueden ser a la vez clientes, servidores de datos y servidores de metadatos. No obstante, podría configurarse para que algunos nodos no compartan sus dispositivos de almacenamiento, podrían ser sólo clientes. Igualmente puede haber nodos que actúen sólo como servidores.

5.7.1 Creación y destrucción de inodos en memoria

Los inodos pueden crearse por petición de VFS de dos formas:

- `Lookup`
- `create_inode`

Hasta el kernel 2.6.24 había una petición adicional

→*`read_inode()`*

Así, cuando AbFS requiere crear una estructura `inode` llama a la función `ilookup()` de VFS. Esta comprueba en el hash de inodos si está presente devolviendo el puntero a la estructura o bien crea una estructura nueva llamando a `abfs_s_alloc_inode()`.

Los inodos persisten en memoria mientras su `i_count > 0`. Cuando `i_count` vale 0 pueden seguir por un tiempo. La función que decrementa el contador es `iput()`. (Ver más adelante).Si `i_count >0` AbFS permite que los nodos permanezcan en memoria, debiendo revalidar el inodo al servidor cada determinado tiempo. Si `i_count` vale 0, no lo revalida. Esto realmente no plantea un problema, ya que VFS nunca hace una búsqueda directa de un inodo en la tabla hash, antes accede por AbFS de forma que valida el inodo.

AbFS puede retener "artificialmente" un inodo válido en memoria incrementado `i_count`. Esto es necesario por los servidores para mantener los inodos en memoria, aunque ellos mismos (desde el VFS) no necesiten mantenerlo.

La caché dentry también incrementa `i_count` cuando necesita mantener un inodo en memoria. En esta versión de AbFS cuando un cliente solicita un inodo, el servidor lo incluye en una lista de clientes, de forma que todos los clientes están controlados.

5.7.2 Borrado de entradas en el sistema de ficheros

El borrado es un proceso en dos fases:

I) Eliminar la entrada de directorio

II) Eliminar el inodo cuando todos dejan de utilizarlo (`i_count ==0`).

En la fase I hay que eliminar la entrada en el servidor y en las caché dentry de aquellos nodos que la contienen. La fase II elimina del servidor todos los extents asociados a dicho inodo y finalmente libera el inodo para que pueda ser utilizado. Analizando el flujo de datos hay que avisar a un número limitado de nodos, aunque el inodo esté en la cache de varios nodos. Esto es debido a que VFS tiene "limitado" el acceso a los inodos. Veamos en qué casos puede estar un inodo en memoria:

- El fichero está abierto y por lo tanto hay operaciones de lectura/escritura.
- El inodo está utilizado por la cache dentry y en este caso sí hay que eliminarlo.
- El inodo está en cache e `i_count=0`. En este caso no lo utiliza nadie. Si hay una operación de readdir, devuelve además del inodo, el número de generación del inodo. Esto permite comprobar cuando se devuelve un readdir, que si el inodo que está en caché no coincide con el número de generación, hay que descartarlo.

Así, el servidor sólo debe guardar una lista (enlazada o como árbol rojo-negro) para cada inodo para los siguientes casos:

- Nodos con ficheros abiertos.
- Nodos que pueden tener referencias al inodo en su caché dentry.

Cuando se hace una operación *sync* del inodo, también se libera la entrada dentry. Esto permite "acelerar" el acceso a un fichero creado y si está todavía en cache, por lo que cuando se cierra el fichero, no habrá referencias al inodo en el nodo que lo creó. Así, el problema sólo se reduce a las entradas de tipo directorio ya que los clientes que guardan referencias en su caché de dentry.

Cuando se hace una operación *readdir*, se hacen *lookups* por cada nombre recibido, ya que es la única forma de que VFS consiga un inodo. En ese caso no se queda en *dentry*. El problema está en que cuando se crea un fichero (o directorio) el directorio padre sí queda en *cache dentry*.

Cuando se crea un fichero en el servidor se busca hacia arriba los directorios y se añaden como nodos que pueden tener en su *dentry* la referencia al directorio. Así, al borrarlo se envía aviso sólo a esos nodos. Dichos inodos permanecen en el servidor de forma que, cada vez que transcurre el tiempo de revisión del inodo, se eliminan las entradas a nodos que han expirado en el tiempo.

5.7.3 Destrucción de inodos

Tabla 5.2 Información interna en tiempo de ejecución

Información	Nivel
Páginas cache (sólo fich.)	4
AbFS: Arbol de extents (sólo fich.)	3
AbFS: arbol RB clients	3
AbFS: Referencias válidas a BH.	2
AbFS: S_I (Server inode)	1
AbFS: Entrada válida en tablas temporización.	1
AbFS: child_list (hijo)	1
AbFS: child_list (padre)	1
Info VFS: Fechas, atributos	1
Espacio struct server_inode	0
Espacio struct ab_inode	0

La tabla 5.2 muestra información interna en tiempo de ejecución que cualquier inodo puede contener. Existen varias funciones (`abfs_clear_inode_l`...) que liberan dicha información. Estas se pueden llamar desde distintas partes del código, en función de la versión de kernel.

A partir del kernel 2.6.35, las funciones →`drop()` pueden decidir si un inodo puede permanecer en memoria o no. En versiones anteriores se puede jugar con el flag MS_ACTIVE del sb, pero por defecto quedan en memoria. Por defecto en ambos casos quedará con `i_count=0` y tener páginas de caché válidas.

AbFS implementa mecanismos de expiración de tiempo para los inodos, éstos se pueden mantener durante un tiempo adicional en memoria. Si se desea "forzar" inodos durante un tiempo extendido se debe incrementar `i_count` para garantizar la revalidación de un inodo. En la tabla de temporización puede haber inodos con `i_count=0` pero tan pronto se ejecute su función de temporización, liberan la mayor parte de la información.

Así, AbFS utiliza un modelo donde la información activa de un inodo se puede mantener un tiempo más, pero al cabo de un tiempo limitado dicha información desaparecerá si el inodo no se ha revalidado. La regla para saber si debe liberarse o extenderse el tiempo es en función del *last_jiffies* (y no *i_count*). Esto es porque puede haberse accedido recientemente a un inodo y haberse liberado, por lo que el tiempo es algo mayor aunque `i_count =0`.*I_count* determina si se revalida o no. `I_count >0` es la forma de saber que un inodo es útil.

Para evitar conflictos en el acceso a los inodos, las funciones de temporización sólo se utilizan para revalidar inodos. Para ello, `i_count` debe ser `> 0`. Así, cuando se crea un inodo en memoria, no se añade a los temporizadores. Cuando se ejecuta →`drop()`, ésta función debe ser rápida y en AbFS no libera nada. Así, las funciones `clear_inode()` y `evict_inode()` se utilizan para liberar los recursos de un inodo, incluidos los de AbFS. Finalmente, →`destroy_inode()` libera el `struct ab_inode`.

5.7.4 Posibles conflictos por uso simultáneno de los inodos

Un inodo puede accederse de forma simultánea desde varias funciones. La función `abfs_delayed_inode()` se mantiene sólo como función “de emergencia” por si se recibe un →`evict()` y todavía no se había liberado, ya que se encargará la función de temporización.

El problema puede venir por un lookup y simultáneamente otra hebra está liberando recursos del inodo. Para ello se utilizan las banderas de I_FREEING, I_WILL_FREE e I_CLEAR que permiten controlar estas situaciones de riesgo. La ventaja es que ésta se libera sólo si `i_count=0`. Las funciones del servidor hacen `iget()` del inodo por lo que `i_count<>0` y por lo tanto no se liberará el inodo y tendrá información válida. Esta función tiene dos variantes, dependiendo de si la versión del kernel es >= 2.6.35. Para versiones hasta 2.6.34 véase la figura 5.8.

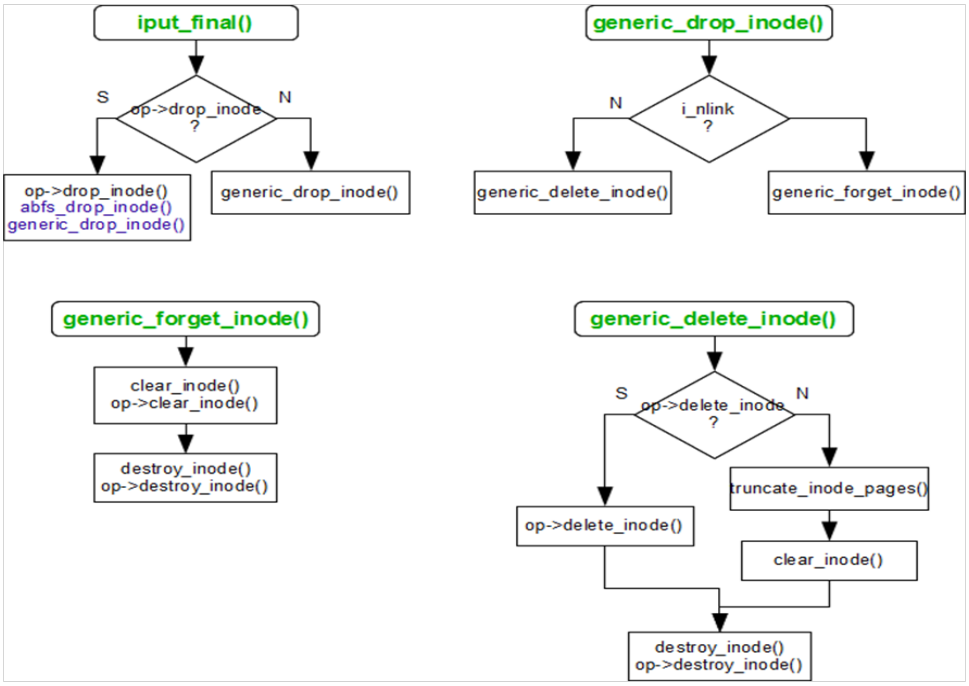


Figura 5.8 Organigramas para las funciones de creación o destrucción de inodos hasta la versión del kernel 2.6.34

Bajo ese modelo, VFS asume que cuando el contado de uso de un inodo vale 0 dicho inodo debe olvidarse o borrarse (`i_nlink=0`). Esto obliga a eliminar toda la información en tiempo real que hay sobre el nodo (páginas en cache). En realidad el inodo permanece en cache, manteniendo sólo la información de metadatos. Esto es útil, pensando que después de un acceso a un fichero se puede hacer un readdir y ya se tiene información válida (fechas, atributos, tamaño, entre otras). Los clientes deben avisar a los servidores si se abre un fichero tanto para lectura como para escritura. Esto permite resolver dos problemas:

- Los servidores deben mantener el `i_count` incrementado para garantizar que la información en tiempo real permanece activa.
- Si hubiera un borrado se sabe qué nodos mantienen ficheros abiertos y por lo tanto hay que esperar a que cierren el fichero para borrar el inodo. También se mantiene información de la última actualización.

La figura 5.9 muestra el organigrama para las funciones de creación o destrucción de inodos para la versión del kernel mayor o igual que 2.6.35.

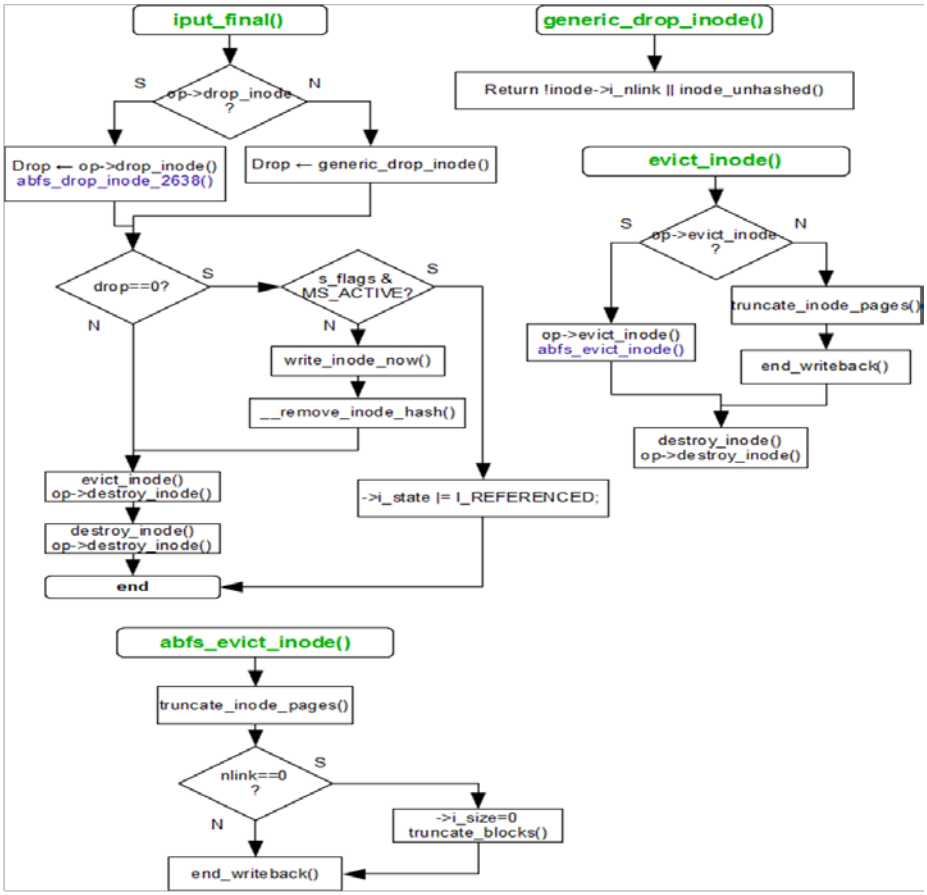


Figura 5.9 Organigramas para las funciones de creación o destrucción de inodos para la versión del kernel mayor o igual que 2.6.35

Los clientes informan sólo cuando se abre por primera vez, se cierra a la última o hay un cambio de estado.

En este modelo, la función drop sólo decide si el inodo debe quedar en la cache de inodos (no realiza directamente la eliminación). De esta forma todavía se mantiene la información completa del inodo. De forma normal, éstos quedarán en caché salvo que se decida borrar el inodo. Ahora la función que realmente "desalojará" el inodo es →`evict_inode()` que finalmente acaba sincronizando el inodo al disco con `end_writeback()`.

5.7.5 Elementos internos de un inodo

La información que guarda un inodo está dividida en tres partes:

- Atributos del inodo
- Nombre
- Relación entre inodos

La función propia del VFS, `iget e iput` sólo gestionan la primea de ellas ya que el resto de información no es formalmente un inodo. Para guardar coherencia existen dos bloqueos que garantizan que no se vaya a cambiar información de una parte que pueda afectar a la otra. El problema aparece cuando se lee un inodo y por lo tanto está en memoria. Los campos que pertenecen al inodo en memoria no pueden cambiarse accediendo directamente al disco ya que provocarían posibles incoherencias.

El bloqueo está a nivel de subtabla de inodos y otro en cada inodo.Una función de *lookup* bloquea para lectura la subtabla para poder buscar en los nombres directamente en los bloques que contienen la información. Esto evita tener que leer cada inodo si la información sólo se obtuviera con iget.

Cuando se envía un inodo remotamente se envía toda la información, para que pueda retenerla y utilizar posteriormente, pero sólo lo utilizará como lectura.

Así, un `iget()` no lee la información de relación entre inodos, ya que iget no nos garantiza cuándo se lee del disco/buffer donde es el único sitio donde sí está actualizada. Es

posible que el inodo estuviera en caché. Por lo que las funciones de lectura y escritura de dicha información van directas sobre los bloques.

Es las estructura de inodo está esa información para utilizarla posteriormente. Los enlaces sólo sirven para las operaciones `readdir`.

Cuando se crea un inodo se bloquea la subtabla y el inodo, se realizan los cambios, y se liberan ambos. En la implementación está dividida en dos estructuras, una directamente con el inodo VFS y otra, abfs_st_priv_inode con el resto de información. Está así dividida ya que a veces es posible hacer cambios de un elemento sin que tengan que estar presentes el conjunto de elementos de inodo.(ej. Cuando se cambian los punteros prev y next de relación entre inodos).

5.7.5.1 Coherencia de dentry

Ya que la cache de dentry mantiene sólo los nombres de los elementos a los que se ha accedido, sólo necesita un mecanismo de invalidación de entradas cuando se borra un fichero. AbFS implementa un sistema optimizado que permite invalidar ficheros o directamente el directorio que los contiene (borrado masivo de ficheros). Cuando se crean ficheros, esta caché no se ve afectada ya que cuando un nodo quiere acceder a un fichero recientemente creado por otro nodo, el cliente accede directamente al servidor.

5.7.5.2 Coherencia de inodos

Los inodos plantean algunos problemas adicionales a los dentry ya que cuando un nodo modifica sus atributos los inodos deben reconocer dicho cambio.

5.7.5.3 Coherencia de file

Cuando se accede a un fichero se pueden establecer diversos modos (lectura, escritura, modo exclusivo,…). Estos modos también tienen que reflejarse en cada nodo cuando se accede a dicho fichero.

Para mantener la información de un inodo AbFS necesita mantener información adicional a éste. En lugar de tener dos reservas de memoria, una por el inodo del VFS y otra

para los datos realmente la función abfs_alloc_inode reserva espacio para ambas y se devuelve el puntero del inodo de VFS.

Los inodos mantienen información de un fichero dentro del VFS. Así cuando se requiere un inodo se llamará a la función abfs_read_inode que debe rellenar la estructura inode con los datos del inodo solicitado. En un sistema de ficheros basado en bloques de tipo "convencional" se solicita el bloque a la función sb_read que es una función que gestiona los buffer_head, de forma que si el bloque que contiene el inodo en disco está en memoria(caché), esta función devuelve el puntero a un buffer_head donde hay información del bloque que contiene los datos. Este mecanismo es válido cuando el almacenamiento es local, pero cuando el inodo es remoto este mecanismo no puede utilizarse.

Cuando el inodo está en otro nodo debe gestionarse una cache de inodos que permite mantenerlo durante el tiempo que le indique el servidor, unos 4 minutos.

Cuando llega un mensaje de invalidación se comprueba si el inodo está en la cache y si está se descarta.

5.7.5.4 Relación entre dentry e inodo

En VFS son independientes el nombre del fichero (*dentry*) y la descripción de un fichero (*inodo*). Así, en un enlace tipo hard puede haber dos dentry que apuntan al mismo inodo. En VFS existen estructuras de datos para ambos (*structdentry* y *structinode*). En tiempo real se crea un vínculo entre ambos con *d_instantiate(dentry, inode).* En la figura 5.10 se muestra la relación entre dentry e inodos y los campos que los enlazan.

Cuando se crea un fichero es cuando se realiza la vinculación entre el nombre y el inodo. Las funciones *add_link* son las que crean dicho enlace en una unidad de almacenamiento.

Cuando se crea un enlace *hard* se llama a la función *abfs_link* que recibe el nombre original, el nuevo nombre y el inodo del directorio que los contiene. En el caso de AbFS es el momento en el que se crea el inodo en el volumen, por lo que hasta ese momento, pueden existir estructuras inode sin presencia en el volumen.

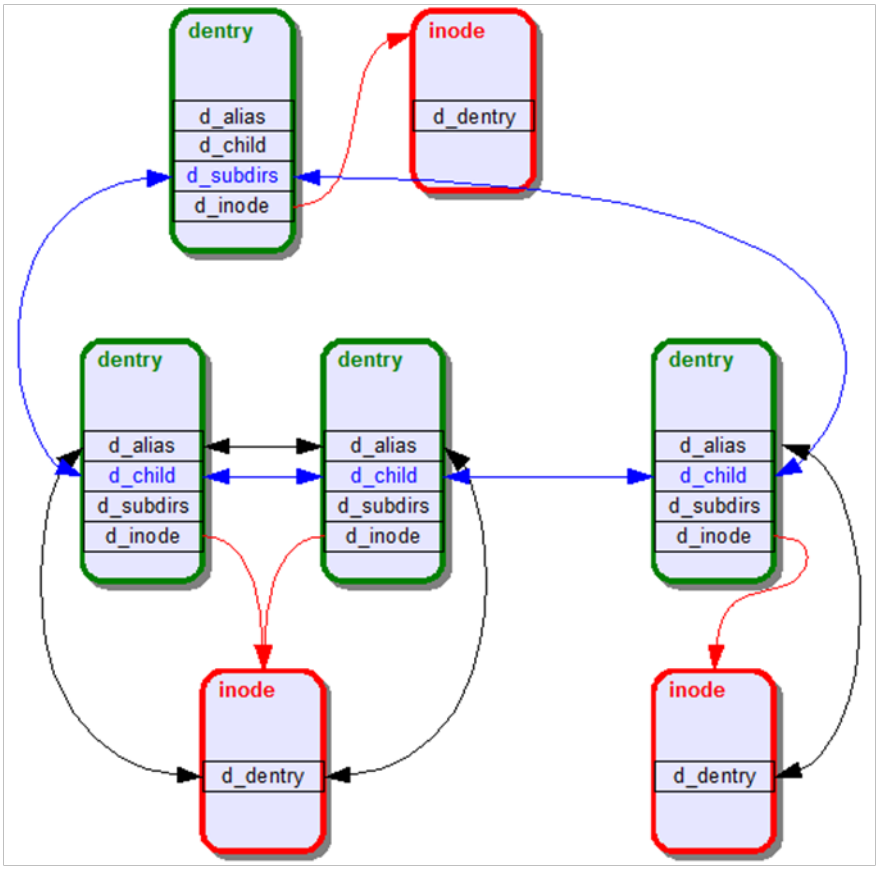


Figura 5.10 Relación entre algunos elementos de dentry e inodos

5.7.5.5 Invalidación de dentry

Cuando se requiere localizar un fichero a partir de un nombre se llama genéricamente a la función do_lookup. Ésta verifica primero en la caché dentry si está presente, si no lo encuentra trata de localizarlo llamando al sistema de ficheros correspondiente. Si lo encuentra y tiene la referencia a una función de validación la llamará para confirmar si está.

Efectivamente es una caché de nombres, si la entrada no está no significa que no exista, es simplemente que no la tiene presente. Por otra parte esta caché se va rellenando con accesos cuando se accede a directorios o cuando se abre un fichero.

Una llamada para obtener el contenido de un directorio (p.ej: originada por una orden ls), genera entradas en la caché de dentry ya que después de las llamadas a readdir, se llaman a

la función de `lookup`, por lo que se recupera también el inodo. `D_alloc` crea las entradas en esta cache. Generalmente una función lookup sí lleva asociada una entrada en esta cache.

Una entrada considerada "Negative·" es una entrada que su valor es válido pero que el inodo al que apuntaba ya no es válido. De hecho el campo `d_inode` está a NULL.

La relación entre un dentry y el número de inodo es más fuerte en AbFS que en VFS. Aunque dos dentry pueden tener el mismo número de inodo en VFS, en realidad un dentry está identificado por el número de inodo de AbFS. Así, cuando se invalida un dentry, en realidad sólo hay que enviar dicho número, mientras que cuando se invalida un inodo se envía el número de inodo que utiliza el VFS. A partir del número de inodo AbFS tiene su propia lista hash para localizar si el dentry está presente. Eso es necesario ya que si para invalidar una entrada de directorio llamamos a la función d_lookup para localizarlo podría tardar basante en el caso de directorios grandes. Para invalidar la entrada de directorio se llama a *d_invalidate*. Esta función puede devolver 0 en caso correcto pero también puede devolver EBUSY si alguien está utilizándolo. En este caso debe dispararse un temporizador cada segundo que trate de invalidarlo lo antes posible.

5.8 Gestión de extents en memoria

Los extents guardan información de los bloques ocupados por un fichero dentro del dispositivo virtual. Dicha información, además de estar almacenada en disco se mantiene en caché, tanto de cliente como servidor para gestionar más rápidamente el acceso a los bloques de datos. Los extents requieren de una estructura que permita el enlace entre nodos mediante un árbol rojo-negro, por lo que se necesita una estructura `rb_node` (red-blacknode) para cada elemento, de forma que no se puede utilizar sólo una imagen directa de los bloques que contienen los extents en el disco. Estos elementos pueden ocupar bastante espacio en memoria por lo que se utilizan páginas. En esta primera versión se van utilizando tantas páginas como sean necesarias que se liberan cuando se elimina el inodo de memoria o en el momento de cerrar el fichero.

5.8.1 Referencia a bloques, posición de un inodo en disco y referencias a extents

La figura 5.11 muestra la relación entre los distintos elementos de la cache de extents en AbFS.

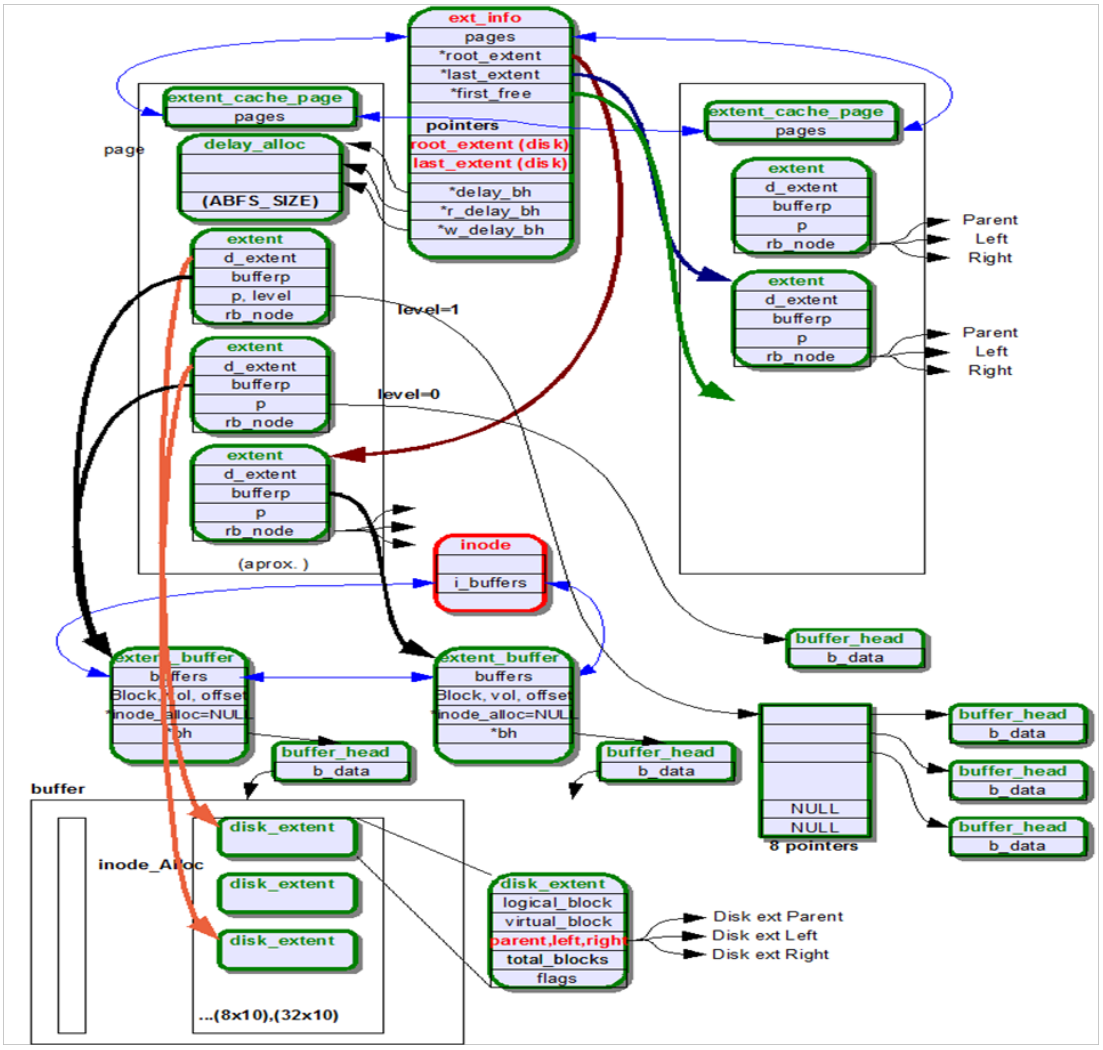


Figura 5.11 Relación entre los distintos elementos de la cache de extents en AbFS

Cada extent en memoria lleva información de:

- Puntero del extent del disco
- Estructura `rb` para acceso en memoria

- Palabra de estado
- Punteros a las estructuras `bh (buffer head)`, en un conjunto de subniveles.
- Nivel utilizado para dicho extent.

Los subniveles son necesarios porque un extent puede estar compuesto por varios `bh`, por lo que se descompone en un árbol de punteros, donde cada nivel almacena 8 punteros a `bh` o a otros subniveles.

Todos los extent en memoria deben existir en disco, pero en memoria no es necesario que estén todos los extents del disco. El orden que debe mantenerse es siempre el que existen en el disco, por lo que no pueden utilizarse las funciones `rbtree` del kernel de Linux basadas sólo para el uso de elementos en memoria. En los árboles RB puede haber rotaciones en los que cambian los enlaces entre los nodos y por lo tanto hay que cambiar también dichos enlaces en el disco. Así si un puntero es NULL, es que dicho enlace no está en memoria, pero puede estar presente en el disco y si se puede tratar de localizar.

Ya que los extent utilizan punteros sobre el espacio virtual pueden localizarse perfectamente el servidor que almacena dichos datos por lo que si un extent no está localmente se puede solicitar. Las funciones devolverán tanto dicho extent como el izquierdo y derecho, de forma que avanza de 2 en 2 niveles en cada consulta.

Cuando un cliente requiere crear nuevos extents, lo solicita al servidor correspondiente y éste crea el extent en función de su espacio disponible, devolviendo dicho extent al cliente que lo solicita. Cuando se modifica un extent del disco, primero se requiere a memoria, se actualiza en el buffer en cache, marcándose como “dirty”. En esta versión de AbFS los clientes reciben un espacio de tamaño inodo con extents. Como se envían tanto los nodos hijos como padres, si estuvieran en unidades inodo distintas se envían también. Se envía esto en lugar de un bloque ya que así los inodos quedan independientes y otros clientes pueden acceder a inodos que estuvieran presentes en el mismo bloque. Si necesitan más, realizan una nueva solicitud. La relación entre nodos es un desplazamiento relativo (Figura 5.12).

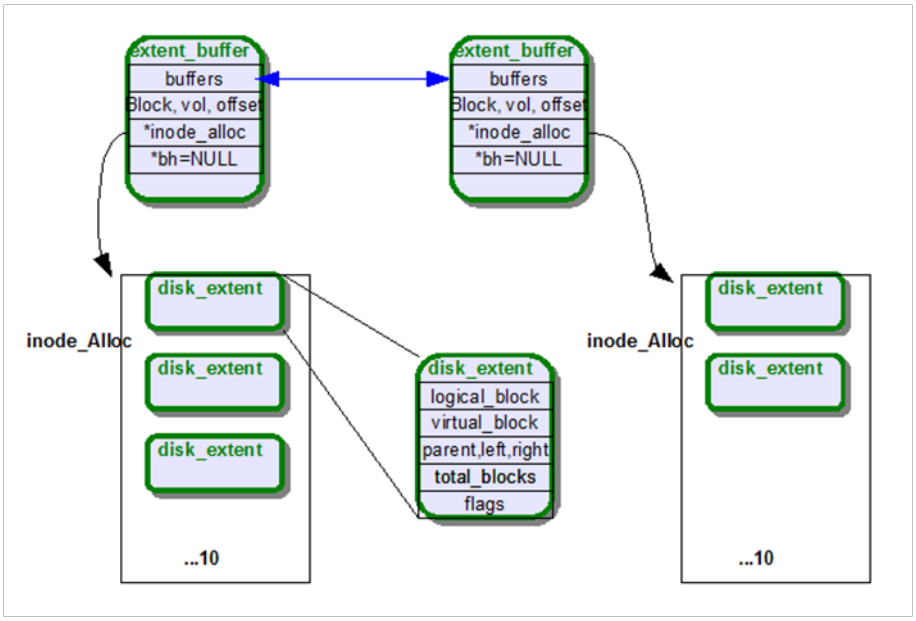


Figura 5.12 Desplazamiento relativo

El funcionamiento del sistema de extent está determinado por el modelo de coherencia de ficheros en función de los distintos estados. Básicamente hay dos operaciones: buscar/leer extent y crear un extent (Tabla 5.3).

El problema del caso (5) es que el servidor puede cambiar nodos del árbol `rb`. En este caso los clientes deshabilitan sus caches y requieren los datos directamente al servidor.

El caso (6) plantea el mismo problema, y la cache queda deshabilitada. En este caso los datos se requieren al cliente que tiene delegado el fichero, si tiene los datos se devuelven, o le informa al servidor que transmita los datos si el cliente no tiene el bloque.

El caso (7) y (8) en los que un fichero pueda estar abierto por varios clientes simultáneamente en RW básicamente consistirá en delegar extents.

El modelo cliente/servidor de extents requiere optimización pero para comprobar el funcionamiento el primer modelo de extent, el servidor realiza todas las funciones remotas y el cliente guarda copia en su memoria, por lo que si se modifica un extent se realiza en el cliente.

Tabla 5.3 Buscar/leer extent y Crear un extent

	Buscar extent		**Crear extent**	
	Cliente	Servidor	Cliente	Servidor
Cli=Serv (1) Fich. R No + clientes	Si el ext está en caché se consulta y si no, se busca el buffer y se añade el extent a la cache.		Insertar extent en el árbol en disco y añadirlo a la caché de extents.	
Cli <> Serv (2) Fich. R No + clientes	Si hay copia local del buffer lo consulta directamente o solicita el buffer al servidor	Envía la copia del buffer al cliente	Se solicita al servidor y el resultado se añade al buffer si está, o se recibe un buffer nuevo.	Se crea el extent y se envía al cliente el ext sólo o el nuevo buffer.
Cli = Serv (3) Fich. RW No + clients	Igual caso (1). El nodo accede a los datos en caché o disco.		Igualcaso (1).	
Cli<>Serv (4) Fich. RW No + clientes	Igual caso (2). No hay datos compartidos	Igualcaso (2)	Igualcaso (2)	Igualcaso (2)
Cli = Serv (5) Fich. RW + clientes R				
Cli <> Serv (6) Fich. RW + clientes R				
Cli = Serv (7) Fich. RW + clientes RW				
Cli<>Serv (8) Fich. RW + clientes RW				

5.8.2 Descripción de algunos tipos de punteros utilizados

ABVIRTUAL_BLOCK: Número de 64 bits que identifica un bloque dentro del sistema de ficheros. Coincidiendo con el tamaño de bloque, los 12 bit más significativos indican el volumen donde está almacenada la información. Actualmente este el valor que se utiliza para

almacenar el número de bloque en los extents, ya que los bloques virtuales (dispositivos) y lógicos (relacionados con los ficheros deben ser del mismo tamaño 4 KB).

LINUX_BLOCK: En este caso el volumen se desplaza 3 bits hacia la derecha ya que posteriormente se multiplicará todo el número por 8 para obtener el número de sectorLINUX_SECT: Linux utiliza el tamaño de sectores de 512 bytes (KERNEL_SECTOR_SIZE) y cuando accede al dispositivo le requiere las transferencias en dicho tamaño, por lo que el dispositivo recibe el número de bloque multiplicado por 8.

ABEXTENT_POINTER (ABEXT_PT): Los punteros utilizados en los extents para hacer referencia a otros extents utilizan 12 bits para indicar el volumen y 52 bits para el desplazamiento dentro del volumen. Como los extents están alineados a 8 bytes, los tres bits siempre estarían a 0, por lo que se aprovechan en la codificación. De esta forma se pueden direccionar 55 bits dentro del volumen. Se puede activar algún flag para indicar un modelo alternativo de direccionamiento, pero en principio no parece necesario.

5.8.3 Funciones en la implementación de los extents

Las funciones que se describen a continuación funcionan igual en cliente o servidor, salvo que se indique alguna diferencia.

- ***abfs_get_blocks():*** El VFS cada vez que requiere un `buffer_head` llama a esta función para obtener el número del bloque asignado o bien asignarle uno nuevo si lo necesita. (La versión con reserva posterior se implementará en la siguiente versión). Es el VFS el que se encarga de solicitar múltiples `buffer_heads` si una página tiene varios bloques. Esta función actúa de interfaz con VFS. Esta función llama a `abfs_get_ext_info()`.
- ***abfs_get_ext_info():*** Esta función trata de localizar el bloque en los extents. Primero lo busca en memoria con y en disco si es necesario con `abfs_rb_search_extent(,1)`. Si tampoco lo encuentra y es necesario crearlo llama a `abfs_rb_create_extent()`.
- ***abfs_rb_search_extent(, modo):*** Esta función busca un extent en caché en memoria (,0) o lo busca también en disco (,1) y lo deja en caché si lo encuentra.

Si el puntero al `root_extent` es NULL es que no hay árbol en memoria, por lo que en el caso 0 se vuelve y en el caso 1 ser requiere la información del disco `abfs_getput_file_inode(,0)` y se trata de añadir el `root_extent` en memoria como primer elemento con `abfs_get_extent( ,NULL, ).`

A partir de aquí se realiza la búsqueda en el árbol, pero si una rama no está en la búsqueda en memoria no se sigue. La búsqueda en disco intenta leer las ramas necesarias del disco para tratar de localizar el extent. Todas estas lecturas se hacen con `abfs_get_extent( ,parent,).`

- ***abfs_getput_file_inode(,modo):*** Esta función lee (modo=0) o escribe (modo=1) la información de donde están situados tanto el root_extent como el last_extent. En el proceso de escritura, comprueba si han cambiado los datos, de forma que si no han cambiado no se marca el bloque como "sucio". Si es servidor se realiza localmente, si es cliente se requiere la función de forma remota.
- ***abfs_alloc_ext():*** Esta función asigna espacio en la caché de extents. Si el puntero first_free es NULL es que no hay información de extent, por lo que se inicia la lista de extent_cache_page y se reserva una página nueva, iniciando así `first_free`.

Comprueba si hay espacio en la página actual, si no hay reserva una nueva página y la añade en la lista dejando los punteros first_free y ext actualizados.

- ***abfs_get_extent(,pt, parent, is_left):*** Esta función recupera un extent del disco y lo deja en caché si lo encuentra. Si parent es NULL considera que es el root_extent.

Se comprueba si hay algún extent_buffer que haga referencia al extent solicitado. Si está, actualiza los punteros de ext. Si no está requiere el bloque del disco o el espacio inodo del servidor, los añade en la lista de extent_buffer y devuelve los punteros actualizados de ext.

Si no hay fallos se reserva el espacio para un nuevo ext con *abfs_alloc_ext().*

Finalmente actualiza su puntero a parent y el puntero, left o right del padre según is_left. Si es servidor se realiza sólo esta función, pero si es cliente se requieren tanto el extent como los dos hijos de forma que se insertan los tres en la caché. El servidor envía los espacios de

almacenamiento inodo que guardan la información de los tres extents a petición del primer extent.

- ***abfs_rb_create_extent():***Esta función primero llama a `abfs_rb_verify_last()` para comprobar si es suficiente modificar el último extent (last_extent). Si esta función devuelve NULL o error sale, de lo contrario hay que crear un extent nuevo tanto en disco como en caché en memoria. Es función tiene que implementar el balanceo del árbol rb, cambio de colores (rojo o negro) e incluso modificar el puntero al root_extent si es necesario.

Se llama a `abfs_alloc_disk_extent()` para crear un nuevo extent en disco en la unidad de almacenamiento inodo o en alguna otra si es necesario.

Se llama a `abfs_alloc_blocks()` para determinar el espacio libre en disco a utilizar.

Después se rellena la información del extent y se realiza la inserción en el árbol rb con `abfs_rb_update_tree()`.

Esta función se realiza en el servidor y el cliente espera la respuesta. Así, el servidor requiere mantener la estructura de inodos necesaria cuando se abren los ficheros para escritura. En el caso de abierto para lectura no es necesario que el servidor mantenga esa información de extents.

- ***abfs_rb_verify_last():***Esta función comprueba si sólo es necesario modificar el last_extent para incluir el bloque solicitado.

Primero se localiza el last_extent y se comprueba si éste puede incluir el espacio requerido. Si es así sólo se modifica la información de dicho extent (tanto en memoria como en disco) y se devuelve un puntero a dicho extent. Si no puede asignarlo devuelve NULL.

Esta función se realiza sólo en el servidor.

- ***abfs_alloc_disk_extent():***Esta función trata de reservar espacio para un extent en disco a partir de la primera libre disponible en el inodo actual, o reservar espacio en la

siguiente unidad inodo. Ya que crea un extent nuevo en disco, también lo crea en memoria.

Crea un espacio nuevo en la cache de extents con `abfs_alloc_ext()`.

Esta función se realiza sólo en el servidor.

- ***abfs_alloc_blocks():*** Busca por defecto espacio libre en el volumen donde está el inodo o en algún otro volumen de los que tengan mayor prioridad para almacenamiento. Ya que existen por defcto bloques de 4 KB y de 64 KB, el algoritmo para determinar cuál se utilizará se basa en utilizar bloques de 4 KB para bloques cuyo número corresponde con espacio por debajo de 512 KB (64 KB *8). Así, el factor por "desperdiciar espacio" es relativamente bajo.

Esta función se realiza sólo en el servidor.

- ***abfs_rb_update_tree():***Esta función modifica los extent de forma que mantienen el árbol RB.

  Esta función se realiza sólo en el servidor.

- ***abfs_free_extent_buffer():*** Libera todos los buffers utilizados.
- ***abfs_free_extent_cache_page():*** Esta función libera todas las páginas.

En esta versión se implementa un modelo sencillo de reserva retrasada de forma que se retiene al menos el espacio equivalente al número de bloques de una unidad de subgrupo grande (64 KB), (ABFS_DELAY_BH_SIZE), por lo que si cada bloque es de 4 KB se retiene hasta esta cantidad. Cuando se cierra el fichero o sólo quedan poco espacio para más bhbh_libres (ABFS_TRIGGER_DELAY_BH) se realiza la reserva de dichos bloques por lo que se puede determinar correctamente bloques contiguos, tanto si es un bloque grande o varios pequeños, e incluso el almacenamiento en el espacio libre del propio inodo.

Otra versión puede implementar la reserva retrasada con más bloques de la siguiente forma: Se activa la bandera y se incrementa la variable que determina el número de bloques

reservados utilizados por el cliente que se envía en los heartbeats. El nodo administrador acumula todos y lo retransmite a todos los nodos para que sepan cuanto hay ocupado. Si un cliente comprueba que no puede reservar más desactiva su reserva retardada.

Cuando asigna su espacio, libera tantos bloques como haya asignado de sus bloques reservados.Cuando VFS llama writepage, en ese momento se realiza la localización a partir del espacio. El ext_info tiene que conocer el último bloque requerido así como del último extent asignado.

Cuando otros nodos acceden a un fichero abierto por otro nodo para escritura es este el que mantiene toda la información actualizada, por lo que el resto de nodos desactivan sus cache y realizan el acceso al nodo cliente.

5.8.4 Cómo se almacenan los extents en disco

Un fichero consta de 4 bloques de información: abfs_st_priv_inode, abfs_st_disk_inode, abfs_st_file_inode y el nombre del fichero. A partir del espacio libre que queda dentro de un espacio inodo se almacenan los extents. Los 3 primeros tienen un tamaño fijo de bytes. El espacio para el nombre queda alineado a múltiplo de 8 bytes. La figura 5.13 muestra cómo se almancena los extent en disco.

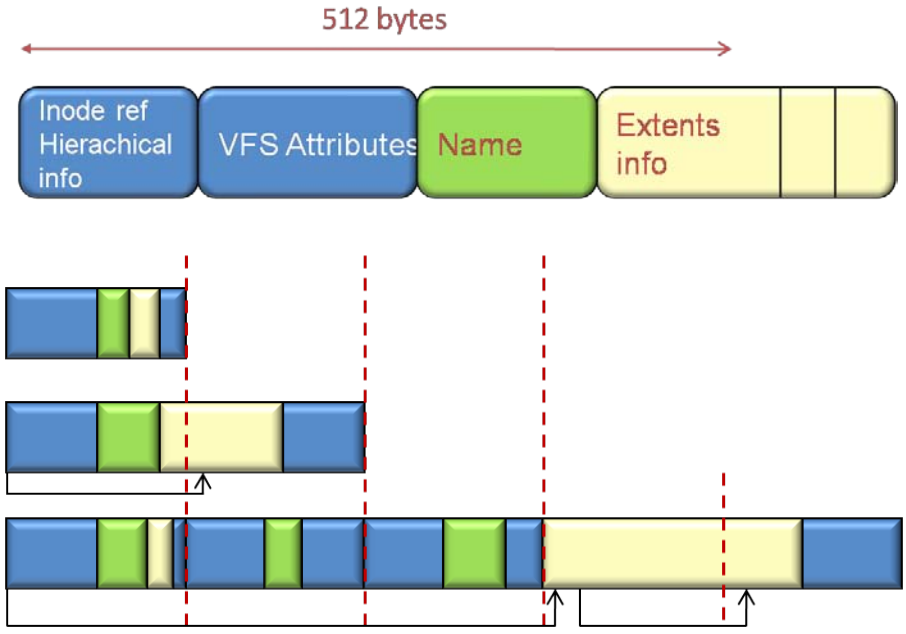


Figura 5.13 Extent en AbFS

Cuando no caben más extent en el espacio inodo original se amplía buscando el siguiente espacio inodo disponible. Los inodos incorporan i_prev_basic_ino y i_next_basic_ino para establecer una lista de inodos que mantienen información de extents. La ventaja de esta lista es que al disponer del enlace al último directamente se puede localizar un nuevo espacio para almacenar un extent en disco. Si estos punteros (prev y next valen 0) es que no hay ningún espacio inodo adicional. El primer inodo guarda dicha información en la estructura abfs_st_file_inode presente sólo en inodos de tipo fichero. El resto de inodos guardan los punteros en la estructura `abfs_st_extent_priv_inode.`

En la figura 5.14 se muestra como va creciendo el espacio de inodo necesario para almacenar los extents. El caso (a) muestra cuando se crea un fichero y se van almacenado los primeros extent en el espacio del inodo. Cuando es necesario más espacio para almacenar el se empiezan a ir enlazando nuevos espacios de inodo. Así el `prev_basic_ino` del fichero siempre localiza al último de los espacios inodo utilizados, con idea de seguir ampliándolo.

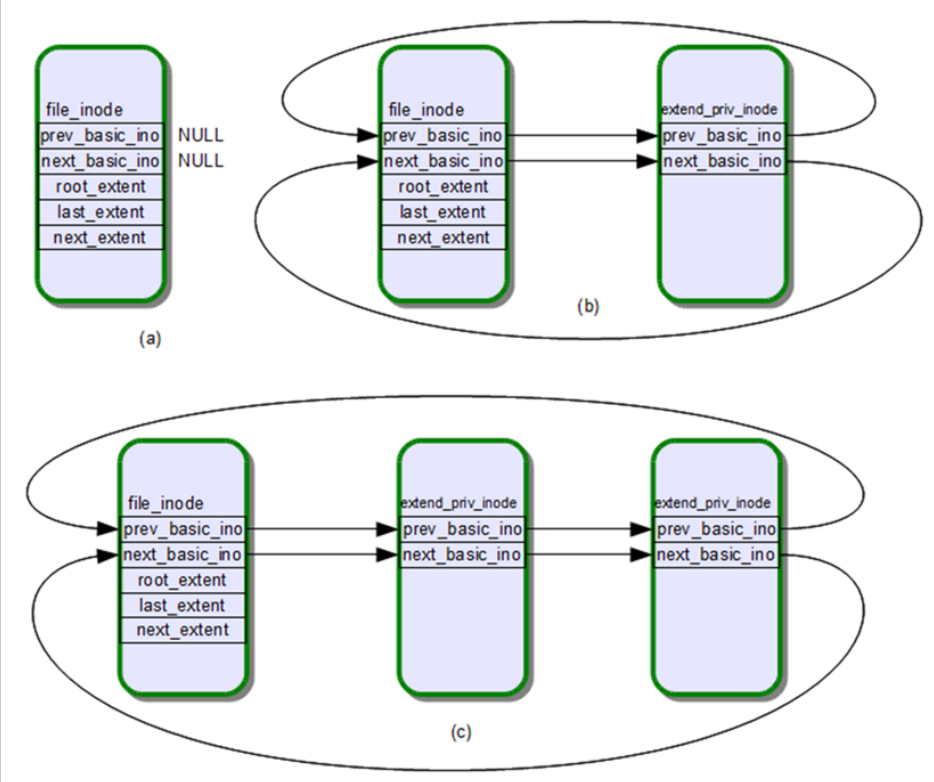


Figura 5.14 Relación entre los extents en la cache de los clientes en AbFS

5.8.4.1 Modificaciones extents

Para garantizar que cliente y servidor realizan los mismos cambios, la secuencia de cambio se codifica y se obtiene un crc de forma que si el crc recibido del servidor y el generado por el cliente no coicide se realiza un flsh de los datos del cliente.

Primero el cliente realiza los cambios y le da la orden de inserción al servidor. El servidor envía el crc de los cambios así como el número asignado como el nuevo extent de disco.

5.8.4.2 Peticiones remotas de bloques fragmentos de cliente a servidor

AbFS incorpora un sistema de transferencia de bloques de disco entre nodos basado principalmente en un dispositivo virtual que permite acceder a todos los bloques del sistema.

Tanto en el caso de accesos a dispositivos locales como al dispositivo virtual todas las peticiones de entrada/salida son gestionadas por el planificador. Las operaciones (principalmente las de escritura) no se realizan inmediatamente tras ser solicitado por algún proceso ya que esto ofrecería un rendimiento bastante reducido. Esto es debido a que, en el caso de los discos, el tiempo de posicionamiento de la cabeza penaliza enormemente el tiempo en completarse una operación ya que puede suponer varios milisegundos. Dichas operaciones se reordenan de forma que puedan optimizar el tiempo en completarse la operación.

En el caso de una lectura de un bloque (completo o parcial) se realiza un petición al nodo servidor enviándole (bloque, desplazamiento, tamaño, tag). El servidor comprueba si la petición está en buffer o la convierte en una petición al sistema operativo pasándole como función de finalización una función de procesamiento así como la página de buffer libre y dicha hebra vuelve a la gestión de peticiones preparando una nueva página (siempre hay al menos una página preparada). La función de realización envía por el canal de transferencias una cabecera con (bloque, tag y tamaño enviado) y libera la página (la pone en una cola de páginas libres) o la descarta. El cliente cuando recibe por el canal de transferencia busca el tag, si no está libera tantos bytes como indica la transferencia, descartando la petición. Si localiza el tag realiza la lectura avisando al bio o a la función que requiera el bloque (según punteros…).

En el caso de una escritura de un bloque se prepara una cabecera con (bloque, tag, -tamaño) y se envía por el canal de transferencia. El tamaño, al ser negativo, el servidor lo reconoce como una petición de escritura, por lo que realiza la petición bio. En este caso, la función de finalización envía un mensaje de vuelta indicando que se ha completado la escritura. El cliente puede reconocer así que se ha completado la operación. Ya que el envío es siempre de bloques, el tamaño tiene que ser múltiplo de ABFS_BLOCK_SIZE. Los números negativos entre 0 y –ABFS_BLOCK_SIZE se utilizan como código de error.

El cliente realiza una petición que queda almacenada en la lista situada en un knode. Se crea la petición y se añade a la lista con un *spinlock*. Cuando el cliente realiza una búsqueda utiliza la versión `for_each_list_safe` y una vez localizado utiliza el *spinlock* para eliminarlo de la lista.

El servidor, cuando recibe la petición la añade en la lista con un *spinlock*. Existe una hebra que se dedica a realizar peticiones en disco. Cuando el bloque está disponible se activa un bit de que el bloque está preparado para ser enviado y desbloquea a la función de envío (si estaba bloqueada). Esta función toma el primer bloque que esté disponible y una vez completado el envío lo elimina de la lista y trata de buscar el siguiente.

Cada volumen local tiene una hebra que gestiona lista de las peticiones pendientes de pedirse al disco. Esa lista tiene un spin_lock para controlar la lista. Cuando ésta está vacía, la hebra está dormida. Si hay peticiones, esta hebra realiza la petición y una vez actualizada, despierta a la hebra de comunicaciones.

Las peticiones de entrada/salida se basan en el modelo que implementa Linux donde cada una de ellas se identifica mediante una estructura `bio`. Éstas definen operaciones “en vuelo” de entrada/salida que puede ser re regiones no contiguas. Los algoritmos de planificación permiten reordenar las operaciones para que las transferencias se realicen de forma más eficiente a los dispositivos. La figura 5.15 muestra cómo las estructuras `bio` pueden hacer referencia a distintas páginas ya que cada petición puede estar compuesta de múltiples vectores que hacen referencia a páginas y tamaño.

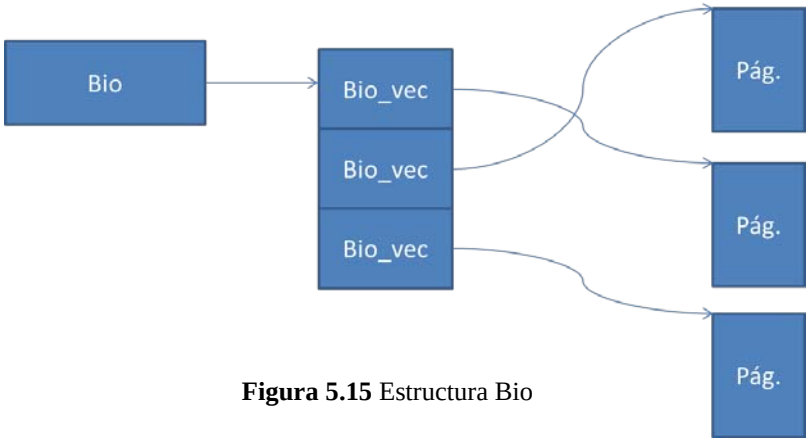


Figura 5.15 Estructura Bio

5.8.5 Transferencias cliente-Servidor

AbFS tiene tres canales principales de envío de información: Envío de órdenes (canal 1), envío de órdenes multicast (canal 2) y envío de datos (canal 3). El canal 1 está orientado a peticiones rápidas, tanto cliente-servidor, servidor-servidor o cliente-cliente. Todas las operaciones de metadatos así como las órdenes de transferencia de datos van en este canal. En la figura 5.16 puede apreciarse una traza de mensajes. En la figura 5.16 puede apreciarse la comunicación del nodo 10.129.4.14 con otros tres servidores (10.129.4.12, 10.129.4.13 y 10.129.4.15).

En el caso de la transferencia de datos (canal 3), permite transferir tanto los datos en modo bloque (transferencias directas en almacenamiento SAN) como los de modo fichero (transferencias indirectas en almacenamiento NAS) según el estado en el que se encuentre un fichero en el cliente (o clientes) y en el servidor. Además, las aplicaciones pueden requerir operaciones síncronas o asíncronas dependiendo de sus necesidades funcionales, lo que da lugar de forma global a cuatro tipos de operaciones como aparece tabla 5.4.

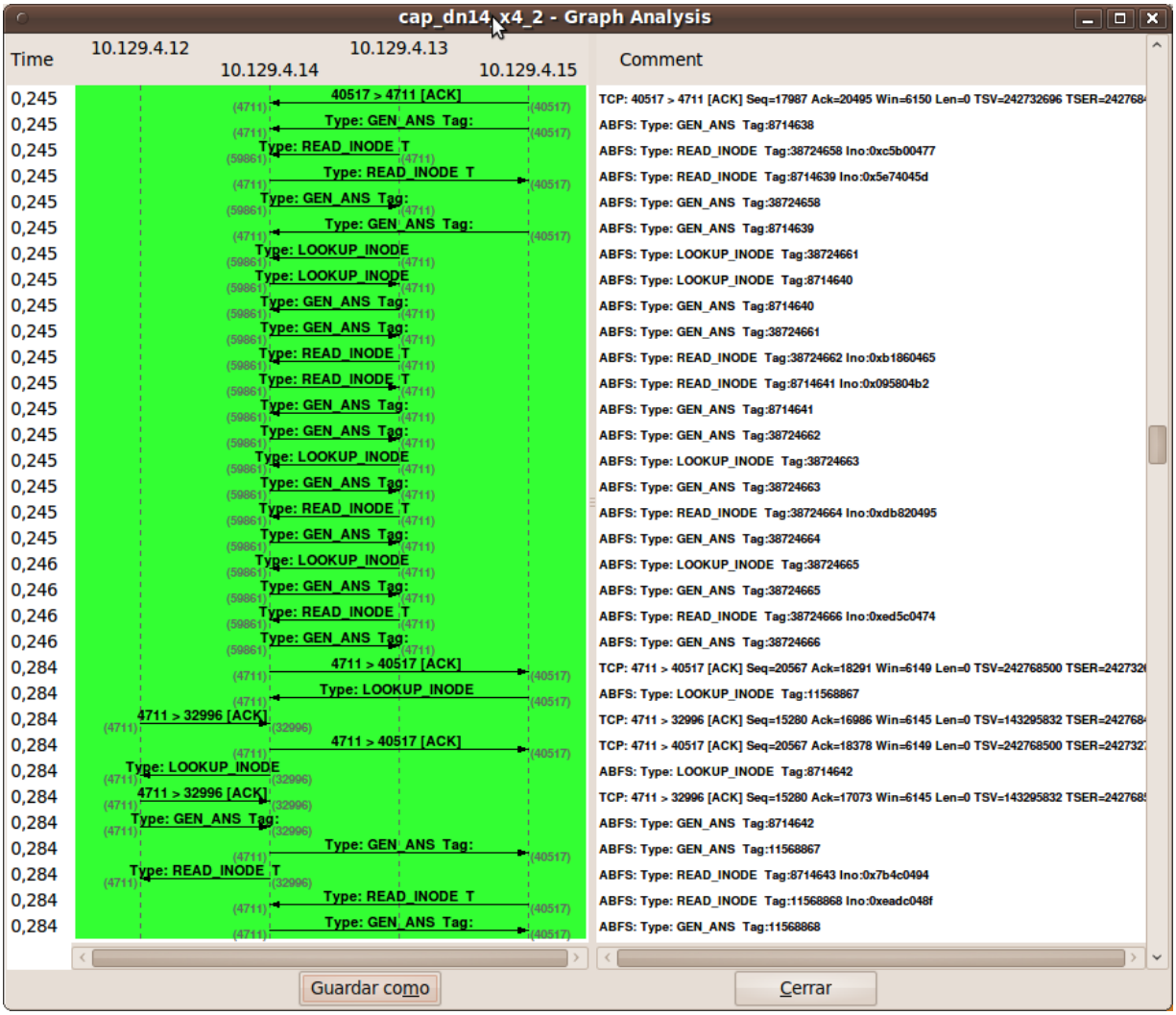


Figura 5.16 Traza de mensjes

Tabla 5.4 Operaciones síncronas o asíncronas

Transferencias	Síncronas	Asíncronas
Directas (modelo SAN)	E/S Bloques Síncronas	E/S Bloques asíncronas
Indirectas (modelo NAS)	Acceso a fichero síncronas	Acceso a fichero asíncronas

La mayor parte de las operaciones se realiza de forma directa asíncrona ya que son las que ofrecen mayor velocidad pero no resuelven todos los problemas de coherencia de cache. Es por ello que son necesarios los otros modelos que, aunque sean más lentos, garantizan el estándar POSIX de acceso a ficheros [POS01, POS08].

Para que las transferencias puedan efectuarse por canales rápidos, los datos deben estar alineados a páginas de destino, aunque tengan un tamaño distinto del tamaño de página. Eso obliga a que si los primeros datos no están alineados las transferencias se realizan de forma indirectas se realizan en varias fases para completar la parte inicial.

Los servidores mantienen en su propia caché de datos los bloques recientes tanto de lectura escritura y para ello se aprovechan los "*buffers*" del propio Sistema Operativo. Cuando se accede a datos que están en la propia caché del servidor la transferencia se acelera considerablemente ya que los medios físicos de almacenamiento son un auténtico cuello de botella.

Cuando AbFS recibe órdenes de escritura de datos a través del dispositivo virtual se crean páginas "sucias" en el servidor que deben ser escritas posteriormente en disco. Esto puede provocar que en servidor aumente significativamente el espacio asignado a *buffer*. Múltiples clientes pueden colapsar fácilmente a un servidor dejándolo sin memoria libre ya que el kernel ofrece tantos *buffers* como se soliciten y tan pronto queda sin memoria RAM libre empieza a utilizarse el mecanismo de intercambio de memoria (swap). Ésto deja prácticamente bloqueados los servidores ya que el kernel puede descartar páginas "limpias" pero no puede descartar las páginas "sucias" hasta que no se escriban en el dispositivo.AbFS supervisa el número de páginas "sucias" de forma que considera en 37,5% (1/4 +1/8 del total de páginas) el número máximoque permite (6 GBytes para un servidor de 16 GBytes de RAM).Es decir, este límite representa el tamaño máximo de datos pendiente de escribirse en el dispositivo, aunque el número de buffer puede ser superior. Este límite se utiliza como criterio para el control de flujo entre servidores y clientes, por lo que las operaciones quedan bloqueadas hasta que se reduce dicho número.

5.8.6 Transferencias cliente-cliente

AbFS permite operaciones entre clientes tanto en el canal 1 (operaciones con metadatos) como en el canal 3 (operaciones con datos). En el caso de datos las transferencias entre clientes se realizan siempre de forma indirecta ya que un cliente no "exporta" una unidad de almacenamiento y no hay referencias a bloques. La única referencia común para los ficheros es el número de inodo, así como el desplazamiento dentro de dicho fichero. Estas operaciones

garantizan el mantenimiento de coherencia de cache, pero finalmente el nodo propietario debe escribir los datos en el servidor, sincronizando y liberando páginas.

En este caso, los clientes que mantienen la propiedad al disponer de los bloques en memoria permiten transferencias entre clientes ya que la sincronización se realiza posteriormente. Estas comunicaciones también son bastante rápidas ya que en cluster de computadores los clientes también están interconectados mediante redes de alta velocidad (Gigabit o Infiniband) lo que ofrece buenas prestaciones globales.

En el caso de las transferencias cliente-cliente se mantiene el límite de páginas sucias como mecanismo de control de flujo para evitar que los clientes también puedan quedar sin memoria por un uso intensivo de las transferencias.

Con esto concluimos el capítulo 6: Implementación de cache en AbFS. Las evaluaciones de la implementación de cache en AbFS se presenta en el capitulo 7.

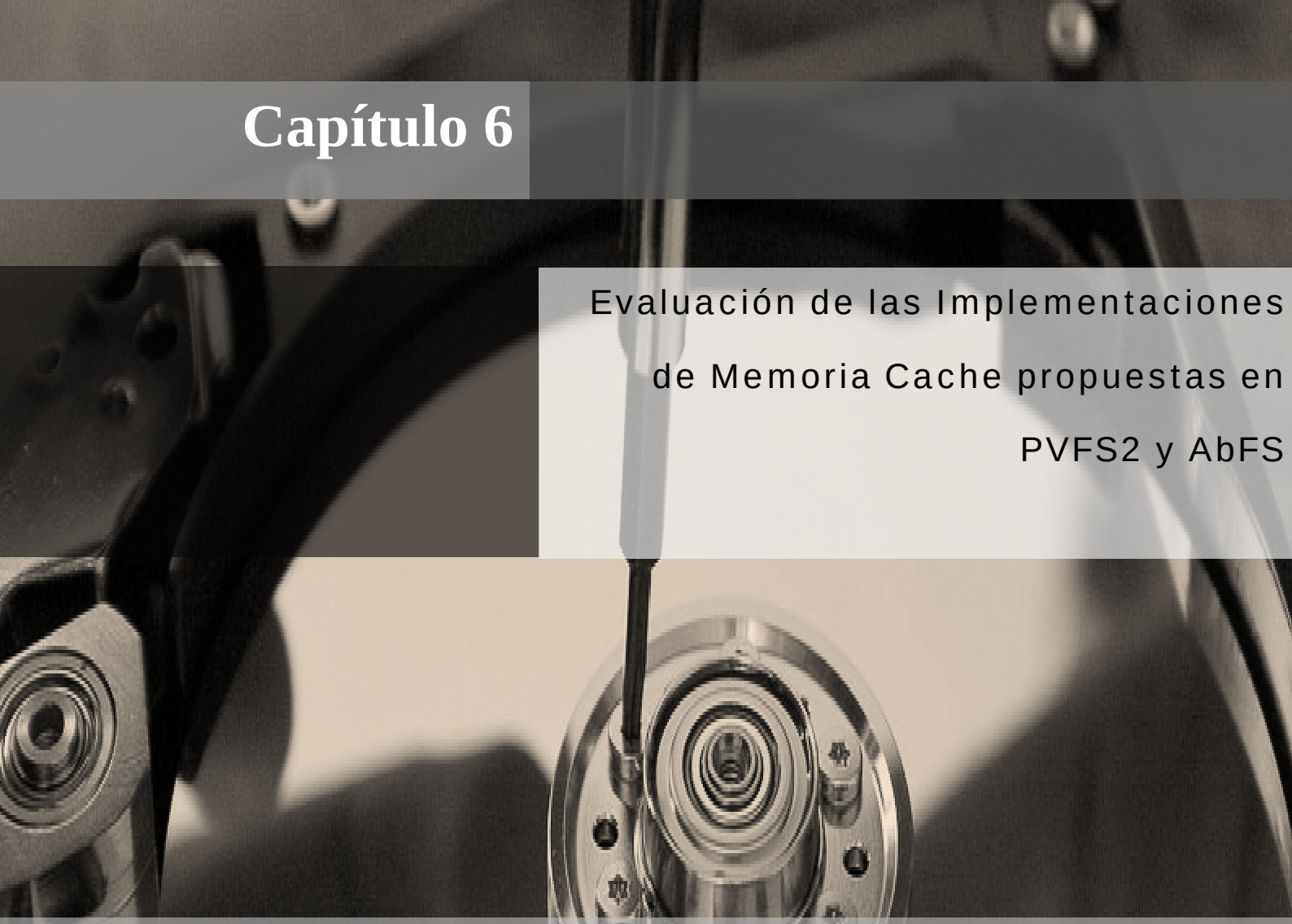
Capítulo 6

Evaluación de las Implementaciones de Memoria Cache propuestas en PVFS2 y AbFS

SUMARIO

Capítulo 6

Evaluación de las Implementaciones de Memoria Cache propuestas en PVFS2 y AbFS



En este capítulo se describen los resultados obtenidos de las evaluaciones realizadas a las implementaciones de memoria cache propuestas en los sistemas de ficheros PVFS2 (Parallel Virtual File System2) y AbFS (Abierto file System). Previamente a la implementación de cache en PVFS2, se estudió en qué medida su inclusión sería eficiente o no haciendo una comparativa entre el sistema de ficheros PVFS2 y ext3. Tras una breve introducción (Sección 6.1) se presentan los resultados de la comparativa de PVFS2 y ext3 (sección 6.2) y los benchmarks y configuraciones utilizadas para la obtención de los resultados en los sistemas de ficheros PVFS2 (sección 6.3) y AbFS (sección 6.4).

6.1 Introducción

En este capítulo se describen los experimentos realizados para la evaluación de las prestaciones de las implementaciones de cache en los sistemas de ficheros PVFS2 y AbFS. Hay que mencionar que antes de llevar a cabo la implementación de cache se estudió en qué medida su inclusión sería eficiente o no haciendo una comparativa entre el sistema de ficheros PVFS2 y ext3. Se muestran los diferentes tiempos y anchos de banda obtenidos en las operaciones de Entrada/Salida con el sistema de ficheros PVFS2, estos resultados son comparados con los obtenidos con la implementación de memoria cache en los clientes de PVFS2. El uso de cache reduce los tiempos de las escrituras en un 90% aproximadamente y el de las lecturas en un 81% aproximadamente cuando los datos están en cache. Por último, se muestra la ventaja de usar cache de datos en los clientes de AbFS. Usando cache en AbFS los tiempos de lectura y escritura se reducen en un 97% aproximadamente si los datos se encuentran en cache. Antes de presentar los resultados de evaluación se describen los benchmarks y el hardware utilizados.

6.2 Comparativa PVFS2 y ext3

Como se ha comentado, antes de empezar con la descripción de la implementación de memoria cache en los clientes de PVFS2, como paso previo a esta implementación, se hizo una comparación entre el sistema de ficheros PVFS2 y ext3. Inicialmente, se realizó un análisis y evaluación del sistema de ficheros PVFS2, cuya finalidad era la de tener la certeza de que el desarrollo de la implementación de memoria cache para datos en los clientes podía beneficiar significativamente al rendimiento del sistema, reduciendo las latencias e incrementando el ancho de banda en el envío/recepción de los datos.

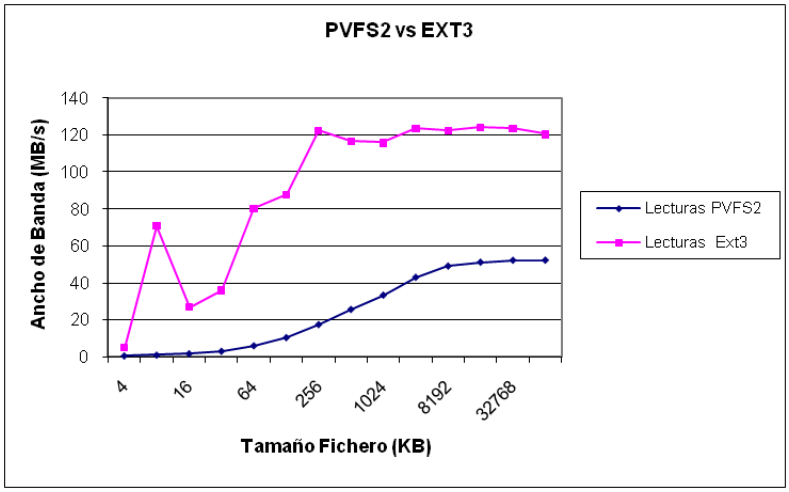
Para la evaluación que aparece en esta sección se ha usado un cluster con las siguientes características:

- Ocho nodos con: procesadores AMD Athlon MP 2400+ (frecuencia real de 2 GHz y 256 KB de cache de nivel 2), 2 GB de memoria principal y disco duro ST380011A IDE (7500 RPM, 100 MB/s de tasa de transferencia pico) en siete de

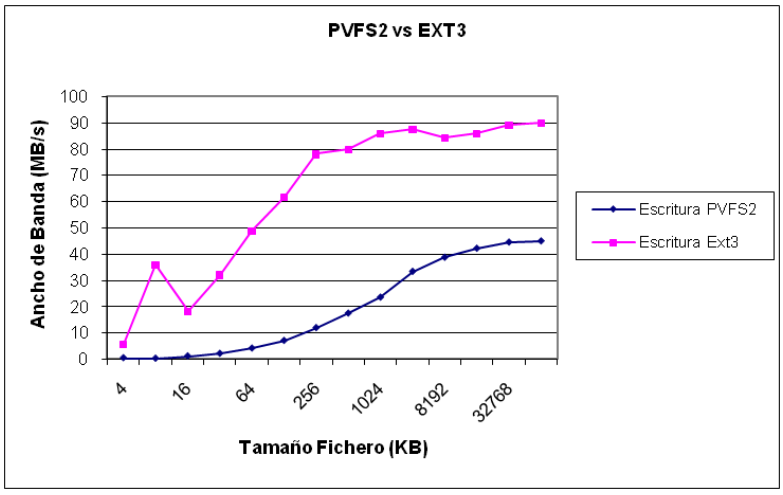
los ocho nodos y disco ST373307LW SCSI (1000 RPM, 320 MB/s de tasa de transferencia pico, 59,9 MB/s de ancho de banda sostenido) en el nodo con conexión al exterior.

- Un switch Gigabit Ethernet 3COM 4900

Se ha utilizado el benchmark *Intereaved or Random* (IOR) para evaluar las prestaciones de PVFS2 y ext3 (Figura 6.1 y Figura 6.2). Los resultados para PVFS2 se obtuvieron con la configuración siguiente: 1 servidor de metadatos, 1 servidor de E/S y 1 cliente, empleando una unidad de *striping* de 64MB y distintos tamaños de fichero desde 4KB hasta 64 MB, ficheros en los que se lee y escribe. De esta forma el fichero se encuentra almacenado en un servidor en lugar de encontrarse distribuido en distintos servidores.

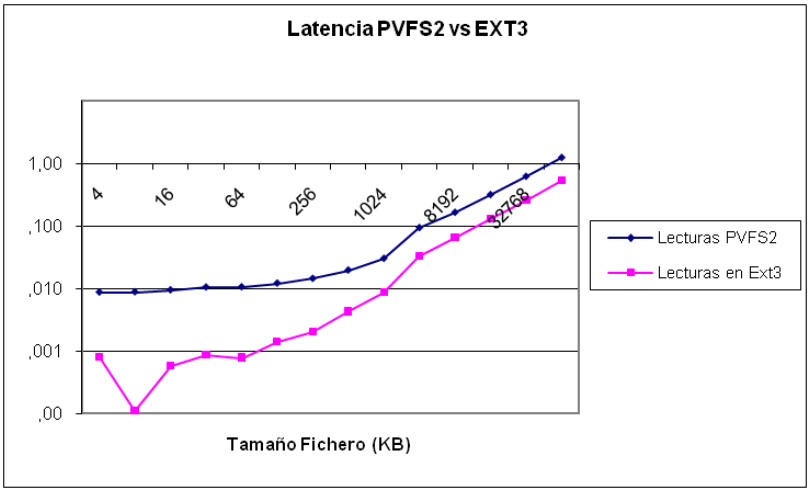


(a)

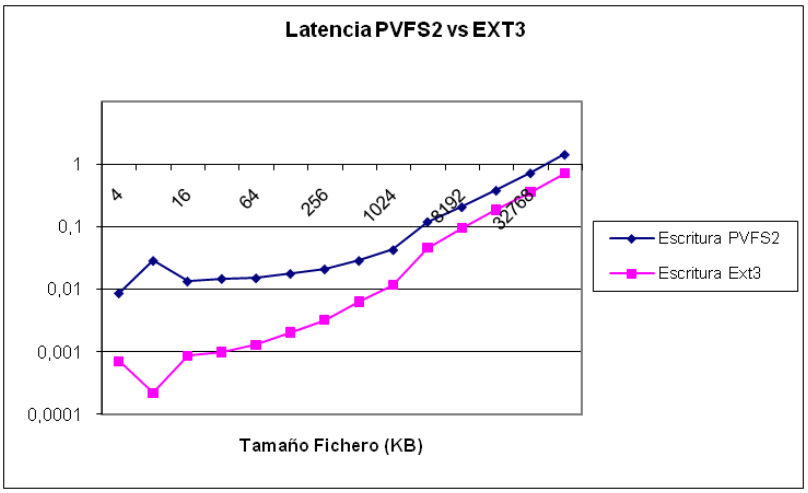


(b)

Figura 6. 1 Ancho de Banda PVFS2 vs ext3 para lectura (a) y para escritura (b)



(a)



(b)

Figura 6. 2 Latencia de PVFS2 vs ext3 para lectura (a) y para escritura (b) en segundos

En las ejecuciones realizadas con ext3, el cliente accede a su sistema de ficheros local y por tanto no hay acceso remoto. En cambio, en las ejecuciones realizadas con PVFS2, el cliente accede a un servidor remoto. Como se puede observar, los accesos con PVFS2 tanto para escritura como para lectura presentan una importante penalización frente a los accesos locales (Figura 6.1 y Figura 6.2). En los resultados mostrados en las figuras no se ve reflejado la penalización que supone acceder a disco, porque el tamaño de los accesos no supera el tamaño de la cache del sistema de ficheros local. La presencia de la cache queda reflejada en el hecho de que el ancho de banda obtenido con ext3 supera el ancho de banda pico del disco

duro (100 MB/s). El tamaño de la cache que se utiliza en el sistema de ficheros local depende de la cantidad de memoria disponible en el servidor.

En las pruebas realizadas los servidores sólo atiende a las peticiones de los clientes, no están ejecutando ninguna otra aplicación. En estas ejecuciones se ha comprobado que la cache de disco utilizada es de 1 GB. Conforme aumenta la cantidad de memoria ocupada, se reduce la cache de disco; por ejemplo, si se ejecutan en el servidor aplicaciones que ocupan el 70% de la memoria principal la cache de disco se reduce a 225 MB, y si ocupan el 80%, se reduce a 106 MB.

Estos resultados nos permitieron deducir que sería aconsejable añadir cache en los clientes de PVFS2. Con esta cache se podría esperar conseguir unas prestaciones más parecidas a las que se obtienen con ext3 cuando se accede a datos que se encuentran en cache del cliente.

Pero hay que tener en cuenta que con PVFS2 se pueden mejorar las prestaciones en el acceso a un fichero usando varios servidores de E/S (Figura 6.3 y Figura 6.4) en lugar de uno. Para estos resultados se utilizó la configuración siguiente: 1 servidor de metadatos, 2 servidores de E/S y una variedad de clientes desde 1 a 4, utilizando una unidad de *striping* de 64,128 y 256 KB. En el test se ha variado el tamaño de los ficheros que se leen y escriben desde 4KB hasta 64MB.

Se debe tener en cuenta, al interpretar los resultados, que en las lecturas se accede a los datos que previamente se han escrito; por tanto, estas lecturas se benefician, al igual que las escrituras, de la cache que incluye el sistema de ficheros local. Como se muestra en las figuras 6.3 y 5.4, la implementación de distintos servidores de E/S y la distribución de los datos (la distribución de PVFS2 puede consultarse en el apartado 3.3.7.1) a través de estos, incrementa el paralelismo. Es decir de esta forma los clientes pueden acceder a los diferentes servidores de E/S donde se almacenan parte de los datos del fichero, obteniéndose una importante mejora en las prestaciones. La latencia se reduce y el ancho de banda se incrementa pero no es comparable con la mejora que se obtendría usando cache, como se puede deducir de la

comparación de las figuras 6.3, 6.4 con las figuras 6.1, 6.2. Además, las trasferencias a las caches de los clientes también se pueden beneficiar de la distribución del fichero entre varios servidores de E/S.

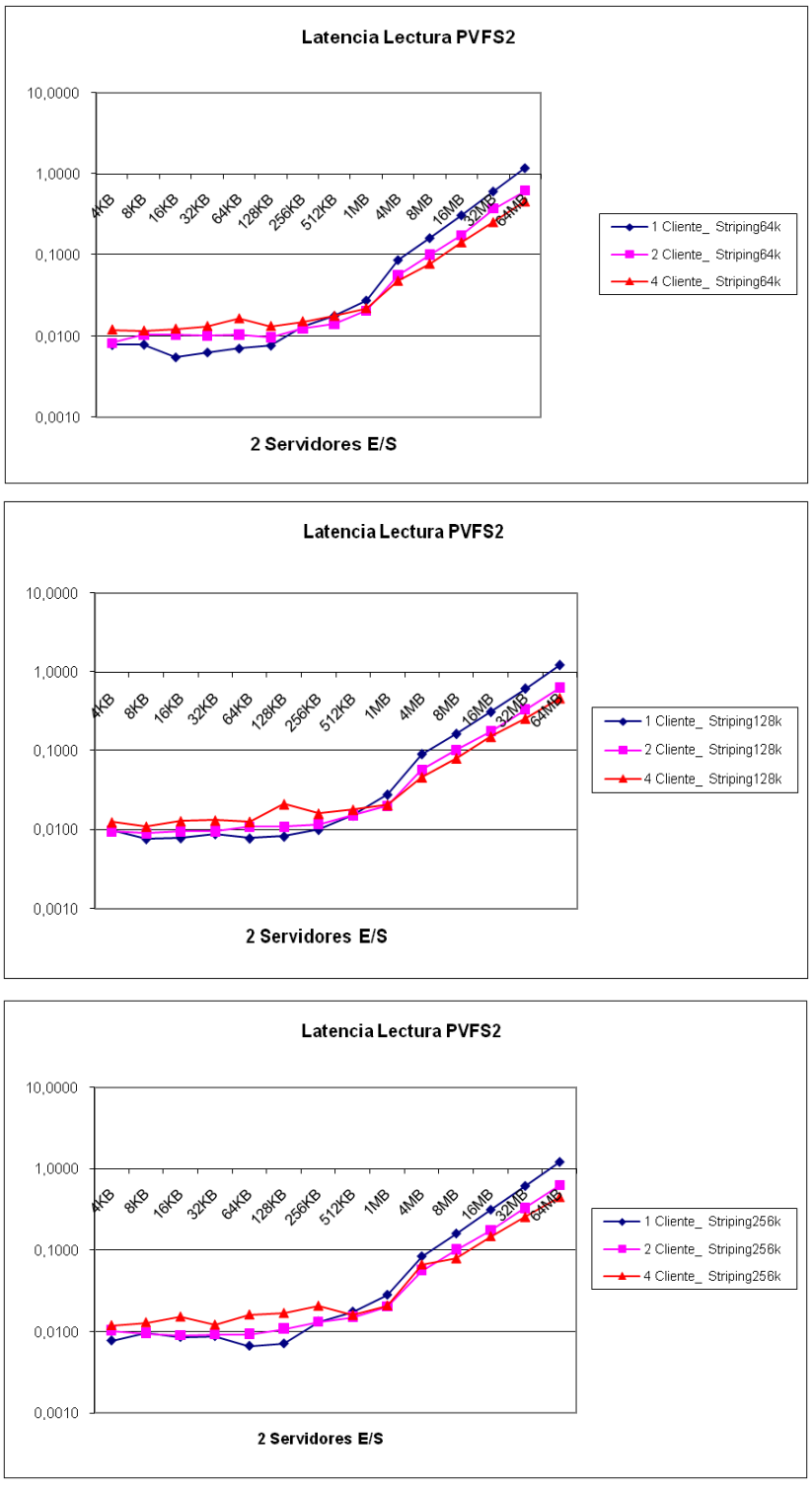
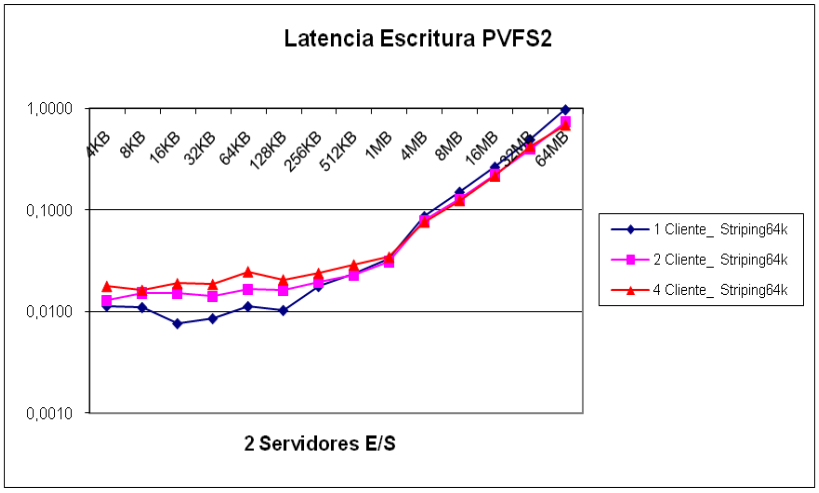
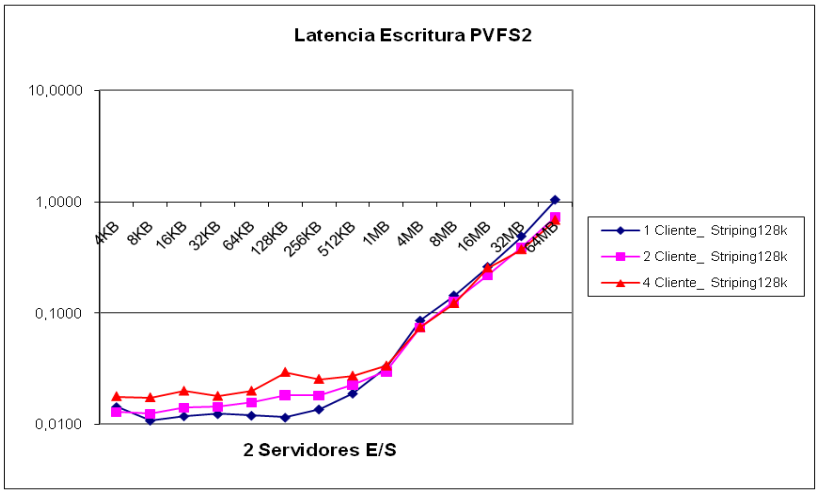


Figura 6. 3 Latencia de lecturas para PVFS2 en segundos





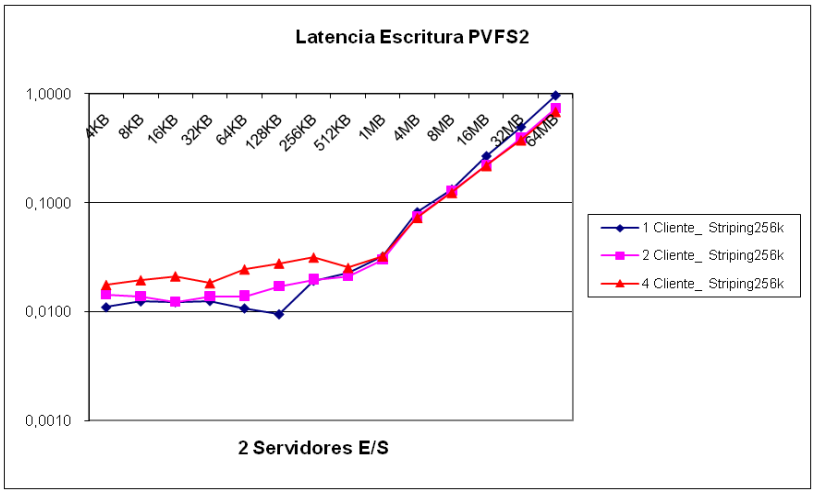


Figura 6. 4 Latencia de Escrituras para PVFS2 en segundos

En estas pruebas realizadas para comparar PVFS2 y ext3, se observa la ventaja que supone acceder a los datos localmente (con ext3 se está accediendo localmente) frente a un acceso remoto (con PVFS2 se está realizando un acceso remoto). Estos resultados incentivaron

y justificaron el esfuerzo de realizar la implementación de cache de datos en los clientes de PVFS2, debido a que se obtendrían mejores prestaciones con la inclusión de cache en los clientes de PVFS2.

6.3 Implementación de memoria cache en PVFS2

Para la evaluación se ha usado un cluster denominado gsd01 que tiene las siguientes características:

- 4 nodos con procesadores dual core
  - 2 nodos tienen 2.4 GHz Intel Core 2 Duo, 2 GB de memoria y un disco duro de 500 GB.
  - 2 nodos tienen 2.4 GHz AMD Athlon 64, 1GB de memoria, y un disco duro de 250 GB.
- Switch Gigabit Ethernet

Cada nodo tiene la siguiente configuración

- Fedora Core 9 como sistema operativo.
- PVFS ver 2.7.1 está instalado en cada uno de los nodos.
- Las pruebas utilizan varios clientes y servidores de E/S.
- Tamaño de bloque de cache es de 4KB.

6.3.1 Benchmark

Se ha implementado un micro-benchmark, llamado `io_file`, que se usa para medir el rendimiento en los accesos de una operación de lectura y/o escritura sobre PVFS2, con y sin la implementación de memoria cache. Este micro-benchmark es adecuado para obtener los tiempos de acceso y anchos de banda de una operación de E/S en la que intervienen múltiples servidores.

La sintaxis para ejecutar `io_file` es la siguiente:

```
[root@wlan1 ~]# io_file  -p /mnt/pvfs2/nombre_fichero -t read ó write
```

donde

-p (PATH) es la ruta del fichero al cual deseamos acceder.

-t (TYPE) el tipo de operación (read-Lectura, write-Escritura) que se quiere ejecutar.

Inicialmente, cuando se ejecuta el micro-benchmark, este pregunta al usuario desde y hasta que posición se requiere leer ó escribir dentro del fichero solicitado. Una vez que le hayamos indicado las posiciones requeridas, internamente se realiza una serie de comprobaciones con la finalidad de que la operación se ejecute correctamente. Por ejemplo, en el caso de una lectura se comprueba que la posición final no supere el tamaño del fichero, o bien que el valor inicial sea un número igual o mayor que 0. Una vez que se terminan estas comprobaciones, se reserva un espacio de memoria (buffer) del tamaño de la petición y posteriormente se ejecuta la operación de entrada/salida solicitada.

Otras funciones adicionales muestran en pantalla (véase la Figura 6.5) los tiempos y anchos de banda que tomó en completarse la operación.

```
[root@wlan1 ~]# io_file  -p /mnt/pvfs2/test_32KB -t read
...
...
El tamaño del fichero es 32768 bytes
Lectura de 32768 bytes en 0.000502 segundos
Ancho de Banda 62.237415 MB/Segundos
```

Figura 6. 5 Ejecución de io_file

6.3.2 Resultados

Para la evaluación de la cache implementada en PVFS2 se ha usado el microbenchmark presentado en al apartado anterior (`io_file`) y diversas configuraciones. Se ha verificado que los resultados son reproducibles. Cada valor se ha obtenido haciendo la media de tres ejecuciones distintas. Las gráficas que se muestran en esta sección permiten comparar PVFS2 con y sin cache. Para la obtención de los resultados se ha utilizado la configuración de 1 Cliente, 4 o 3 servidores de E/S, 1 servidor de metadatos y un tamaño de bloque de cache de 4KB.

Las figuras 6.6 y 6.7 muestran el ancho de banda obtenido cuando se presenta una operación de lectura (Figura 6.6) o escritura (Figura 6.7). Con la versión PVFS2 sin cache (No cache) y con la versión PVFS2 con cache (Cache) tanto en el caso en el que los datos se encuentran en cache (notado con *hit* en las figuras) como en los que hay fallo de cache (notado con *fail*) se observa que usar un mayor número de servidores de E/S favorece que las operaciones E/S se ejecuten en menor tiempo tanto en la versión PVFS2 sin cache (No Cache) como en la versión con cache (*read fail*). En la figuras 6.6 y 6.7 se ve claramente como el ancho de banda obtenido para ambas operaciones en 4 servidores mejora con respecto a utilizar 3 servidores. Esto se debe a que es mayor el número de lecturas/escrituras que pueden ejecutar en paralelo. Es decir, se reparte cada acceso entre más servidores de E/S reduciendo así la carga hacia un mismo servidor o servidores de E/S.

Por otro lado la figura 6.6 muestra sólo un ligero incremento en el ancho de banda para el caso de las lecturas en PVFS2 sin cache (No Cache) con respecto al caso en que las lecturas fallan en la cache (*read fail*) por lo que podemos deducir que la sobrecarga introducida usando la versión con cache cuando los datos no están en cache es mínimo. El rendimiento en la versión con cache mejora significativamente cuando los datos a leer se localizan dentro de esta, ya que los datos pueden ser accedidos rápidamente desde la memoria del cliente, evitando así el acceso a los servidores de E/S.

En la figura 6.7 se observa que una escritura en la cache mejora con respecto a escribir en los servidores de E/S ya que la propagación de los datos se realiza posteriormente, debido a un reemplazamiento o petición del usuario.

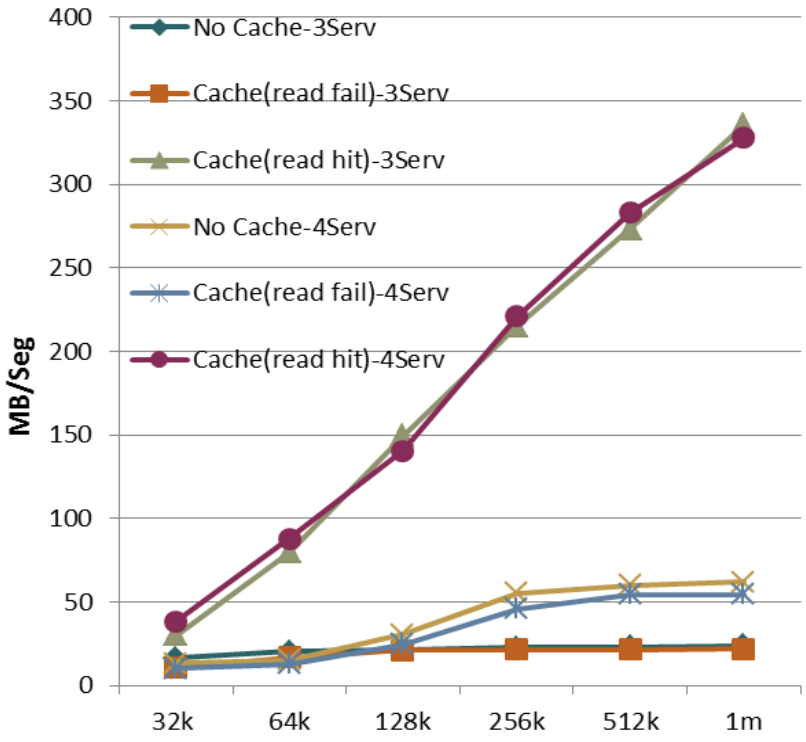


Figura 6. 6 Ancho de Banda para lecturas con 3 y 4 Servidores E/S

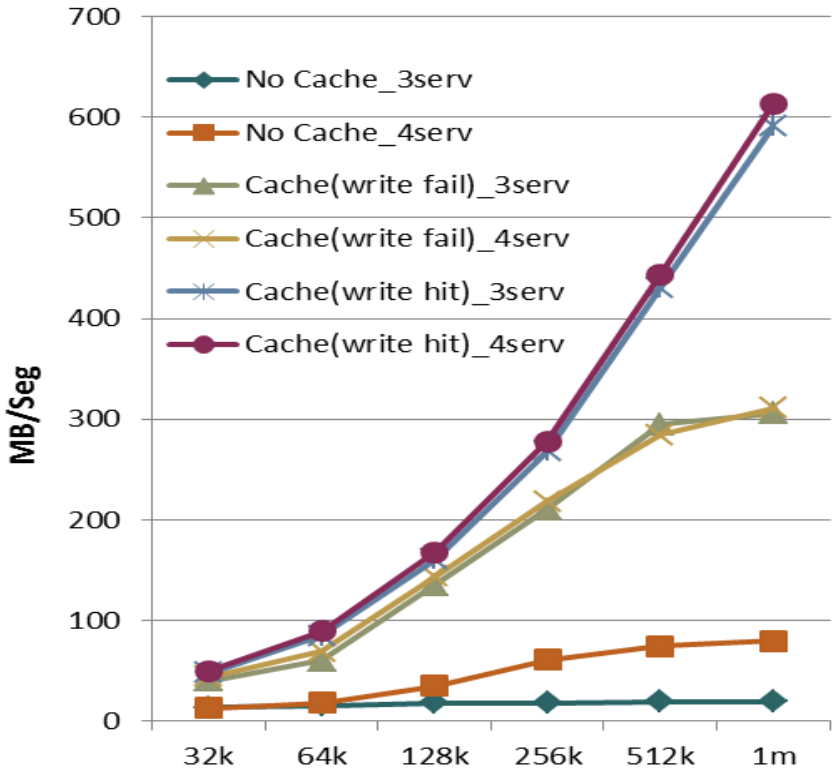


Figura 6. 7 Ancho de Banda para escrituras con 3 y 4 Servidores E/S

6.3.2.1 Sobrecarga (overhead) de la cache del cliente

Las figuras 6.8 y 6.9 muestran la sobrecarga introducida por la cache cuando se presenta una operación de lectura o escritura, respectivamente, cuando no hay localidad en los datos accedidos. Es decir, que los datos leídos o escritos no están en cache. El tiempo se representa en función a la cantidad de datos leídos o escritos por cada una de las peticiones.

La figura 6.8 muestra una sobrecarga muy baja cuando se ejecuta una operación de lectura y los datos no están en la cache. Como se puede observar en la figura la diferencia entre una operación lectura en PVFS2 sin y con cache no es significativa. La diferencia se debe a que en el caso con cache, los bloques son buscados directamente dentro de ésta, si los bloques no se localizan la máquina de estados del cliente es ejecutada enviando la información necesaria a los servidores de E/S para que estos a su vez devuelvan los datos al buffer local del cliente y posteriormente se copian a la cache del cliente.

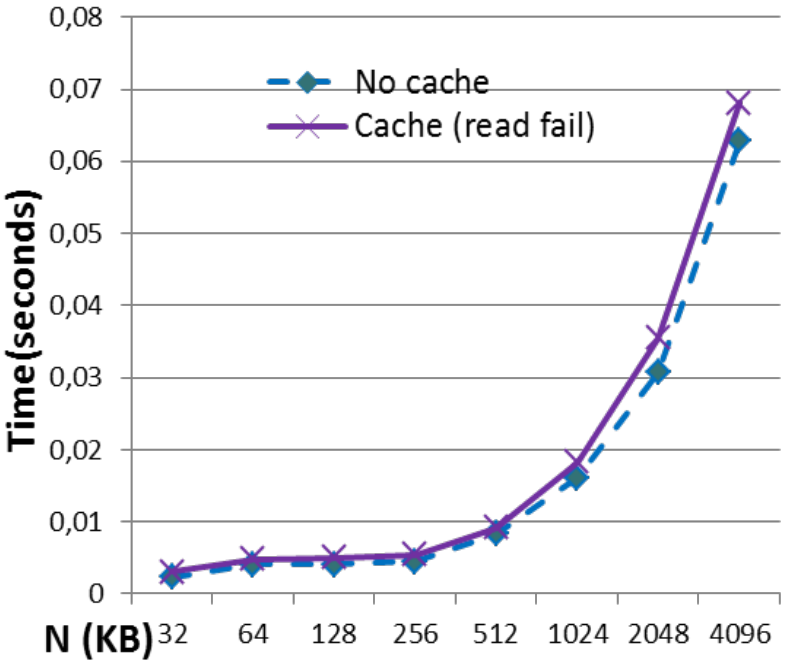


Figura 6. 8 Overhead cache de cliente, lecturas de tamaño N

En la figura 6.9 los bloques a escribir no se encuentran previamente almacenados en cache, la propagación de los bloques a los servidores de Entrada/Salida tiene lugar posteriormente cuando un bloque requiere ser reemplazado por uno nuevo o por petición del usuario. Debido a esto último el tiempo de escritura se reduce considerablemente como se puede observar en la figura aunque no llega a ser tan bajo como el tiempo que se obtiene cuando los datos están en la cache (ver Figura 6.11) porque hay que buscar espacio donde almacenarlos en la cache.

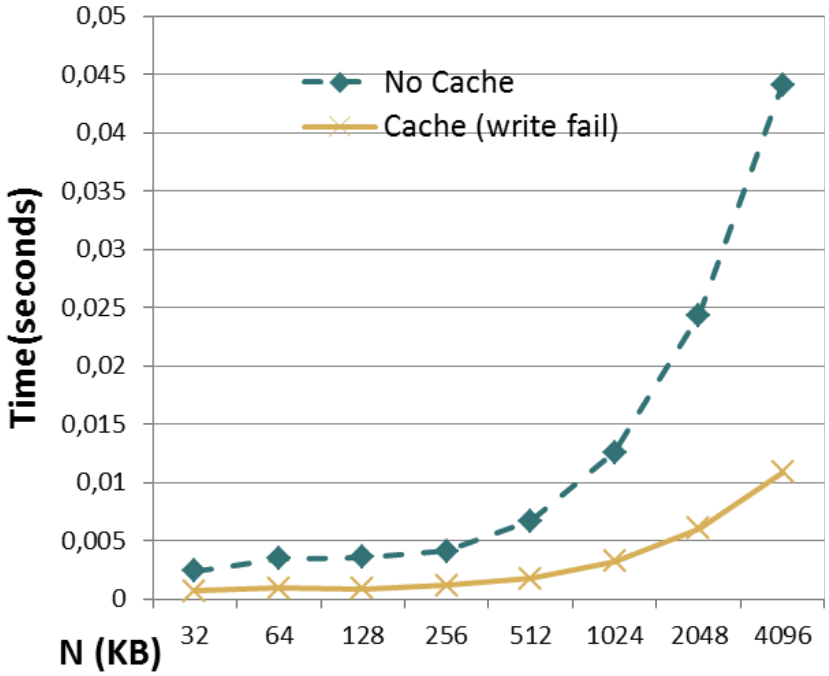


Figura 6. 9 Overhead cache de cliente, escrituras de tamaño N

6.3.2.2 Prestaciones de la cache del cliente

La figura 6.10 y 6.11 muestran la comparación de PVFS2 con y si cache cuando los datos están previamente almacenados en cache. Es decir cuando hay una máxima localidad de los datos en las aplicaciones. El rendimiento mejora cuando las peticiones incrementan su tamaño.

En la figura 6.10 se observa claramente como un acierto en la cache para una operación de lectura es muy superior en prestaciones con respecto a una operación de lectura como normalmente lo hace PVFS2, esto es gracias a que los datos están en la memoria del cliente, evitando así tener que acceder a por los datos a través de la red de interconexión a los servidores de E/S.

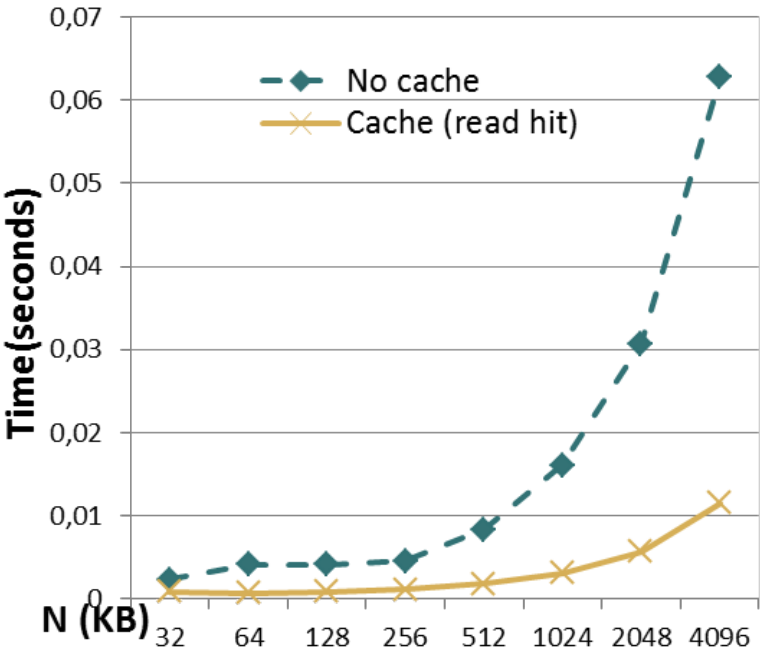


Figura 6. 10 Performance sin y con cache, lectura de tamaño N

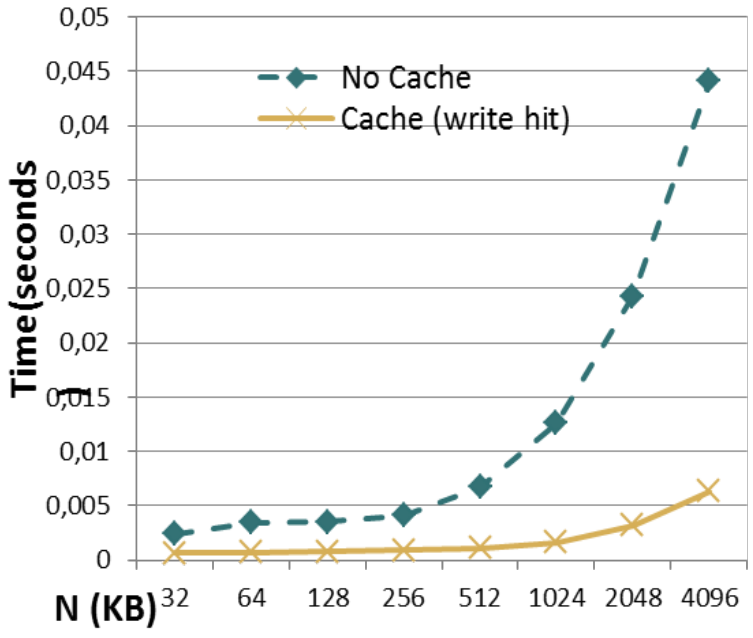


Figura 6. 11 Performance sin y con cache, escritura de tamaño N

En la figura 6.11 en el caso de las escrituras, al igual que las lecturas, se ve la diferencia significativa con respecto a los tiempos de ejecución comparados con PVFS2 sin cache. Por

tanto, se obtiene mejores resultados para este tipo de operaciones cuando los datos se encuentran en la cache del cliente.

6.3.3 Volcado de datos a los servidores de E/S.

Muchas aplicaciones leen y almacenan los datos desde/hacia disco. Estas aplicaciones pueden ser ejecutadas en paralelo en un nodo con múltiples núcleos (*cores*), distribuyendo el trabajo entre varias hebras que acceden al mismo fichero. En este caso cada una de las hebras escribe un trozo de los resultados en el fichero de salida. La cache del cliente puede mejorar el rendimiento de entrada/salida de este tipo de aplicaciones. La E/S mejora porque las hebras ejecutadas en un nodo pueden usar la cache como un buffer de escritura combinada y como un buffer de precaptación. Así las aplicaciones pueden escribir o leer desde los servidores una vez por nodo, en lugar de una vez por núcleo o varias veces por núcleo, si los núcleos escriben o leen desde un fichero varias veces. La figura 6.12, compara el rendimiento de los siguientes casos:

- Ocho escrituras en cache de N/8 bytes cada una, más un volcado o *flush* hacia los servidores de E/S.
- Una escritura sin cache de N bytes.
- Ocho escrituras sin cache de N/8 escrituras hacia los servidores de E/S.

Esta gráfica muestra la ventaja de combinar escrituras en la cache antes de enviar los datos a los servidores de entrada/salida. En una aplicación típica donde se escribe varias veces a disco, tales como aplicaciones de simulación, la propagación de las escrituras de los bloques hacia los servidores de E/S puede realizarse en un segundo plano. Así el volcado de los datos o *flush* no afecta el tiempo de ejecución de la aplicación.

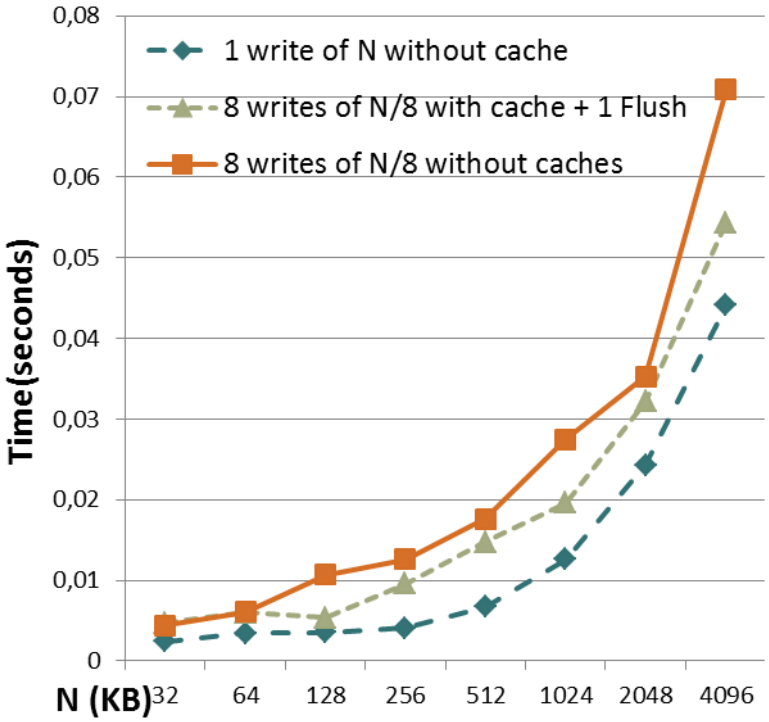


Figura 6. 12 Prestaciones de varias escrituras combinadas + un flush

6.3.4 Resumen

Los resultados muestran que la implementación de memoria cache en los clientes de PVFS2 mejora significativamente el rendimiento de las lecturas cuando los datos están en la cache del cliente, y decrecen muy ligeramente con el rendimiento de una lectura sobre la versión original de PVFS2, es decir cuando los datos no están en cache.

La cache también mejora significativamente el rendimiento de las escrituras. Si los datos no están en cache, el rendimiento de las escrituras es menor, no obstante la diferencia con respecto a tener un acierto en cache es muy baja, la penalización se debe sólo a la gestión de la cache. La cache del cliente es también útil como buffer de escritura combinada. Es decir, se puede usar para combinar las escrituras a un fichero desde los diferentes núcleos de un procesador en una sola escritura hacia los servidores de E/S. Con la cache de datos el usuario

no tiene que modificar los códigos que ejecuta añadiendo código extra que realice explícitamente la combinación de los datos que se van a escribir para lograr que se trasfieran al servidor en una trasferencia en lugar de tener que usar una por cada núcleo También es útil como precaptador porque la lectura de uno de los núcleos de un fichero puede obtener información que requieran otros núcleos y éstos al leer pueden encontrar ya los datos en cache. Los procesadores multinúcleo son actualmente comunes y es importante por tanto hacer un uso efectivo del paralelismo de estos procesadores. Si los sistemas de ficheros aprovechan el hardware usual en nuestros días, las aplicaciones pueden mejorar sus prestaciones.

6.4 Implementación de memoria cache en AbFS

Para comprobar el funcionamiento de la caché en AbFS primero se hizo una evaluación de las prestaciones con dos nodos en modo simétrico y se comparó con un sistema NAS de altas prestaciones utilizado en un cluster de supercomputación. Posteriormente se han añadido más nodos para estudiar su escalabilidad.

6.4.1 Medida de prestaciones con dos servidores en modo simétrico

Para la obtención de los siguientes resultados se ha usado un cluster con dos nodos con las siguientes características:

- 2 nodos (actuando de servidores y clientes). Cada nodo tiene:
  - Dos procesadores Xeon L5420a 2.5GHz,
  - 16GB de RAM
  - Tarjetas de InfiniBand Mellanox MT25417, para IPoIB
  - Discos locales HP GJ0120CAGSP.

Los resultados obtenidos corresponde a escrituras y lecturas de discos en RAW y otros a una NAS de NetApp FAS3140.

Para la evaluación de estas prestaciones se ha usado el benchmark IOzone [IOzone]. IOzone permite generar y medir una gran variedad de operaciones sobre los ficheros entre las que destacan lecturas, escrituras, re-lecturas, re-escrituras, lecturas/escrituras aleatorias, entre otras. Una de las características importantes de IOzone es que puede ser portado a muchas arquitecturas y se ejecuta bajo cualquier sistema operativo. La figura 6.13 muestra un ejemplo al ejecutar `IOzone:`

```
    [root@wlan1 ~]# ./iozone -Racb name_file.wks
    ...
    ...
    /* En este ejemplo, se indica que se genere un reporte en Excel
(R),  después se indica que de manera automática se generen todas la
operaciones de  4KB a 16MB  para ficheros de 64KB a 512MB (a).
Posteriormente se estable que debe cerrarse la operación (c) en el
fichero N (b) */
```

Figura 6. 13 Ejecución de mdtest

Las figuras 6.14 y 6.15 muestran las operaciones de Lectura (Figura 6.14) y Escritura (Figura 6.15) de ejecuciones locales en los clientes de AbFS. Las figuras 6.16, 6.17, 6.18 y 6.19 muestran las operaciones de Lectura/Escritura utilizando una NAS de NetApp FAS3140. En las medidas se utilizo un tamaño de fichero que va de los 4KB hasta 16MB.

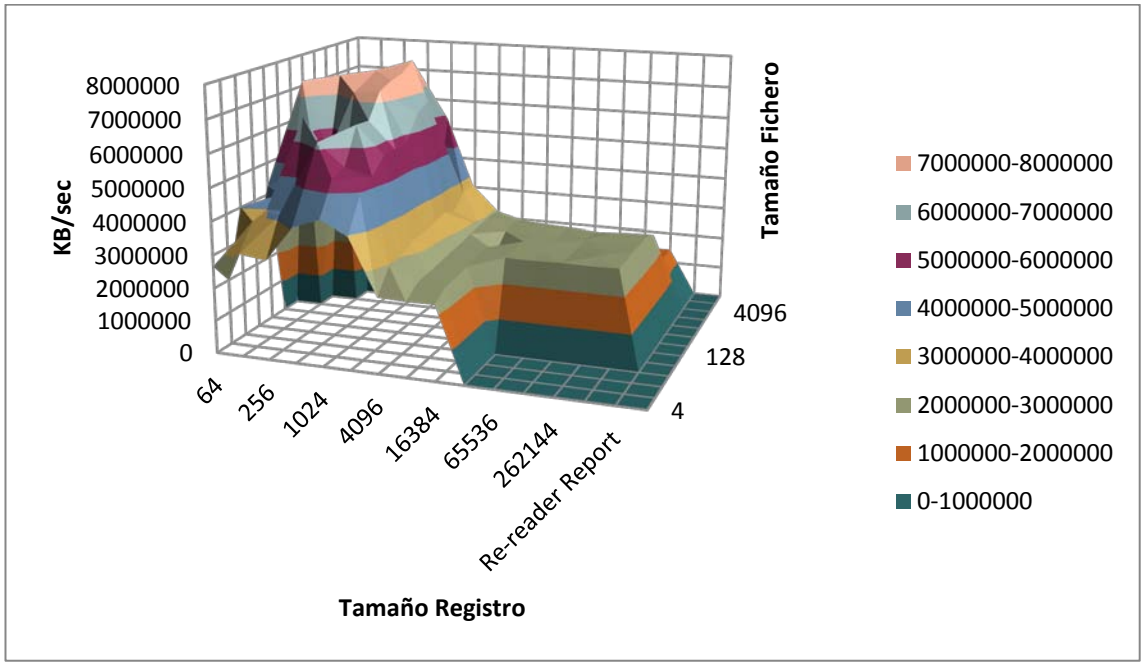


Figura 6. 14 Lectura en disco local en AbFS

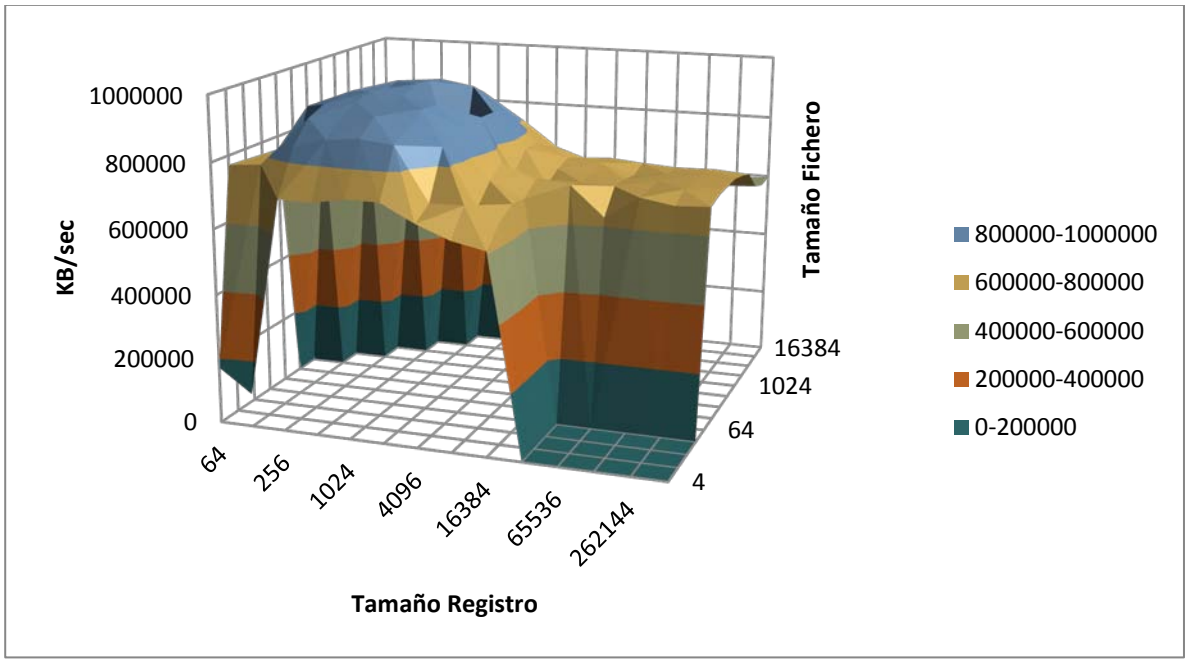


Figura 6. 15 Escritura en disco local de AbFS

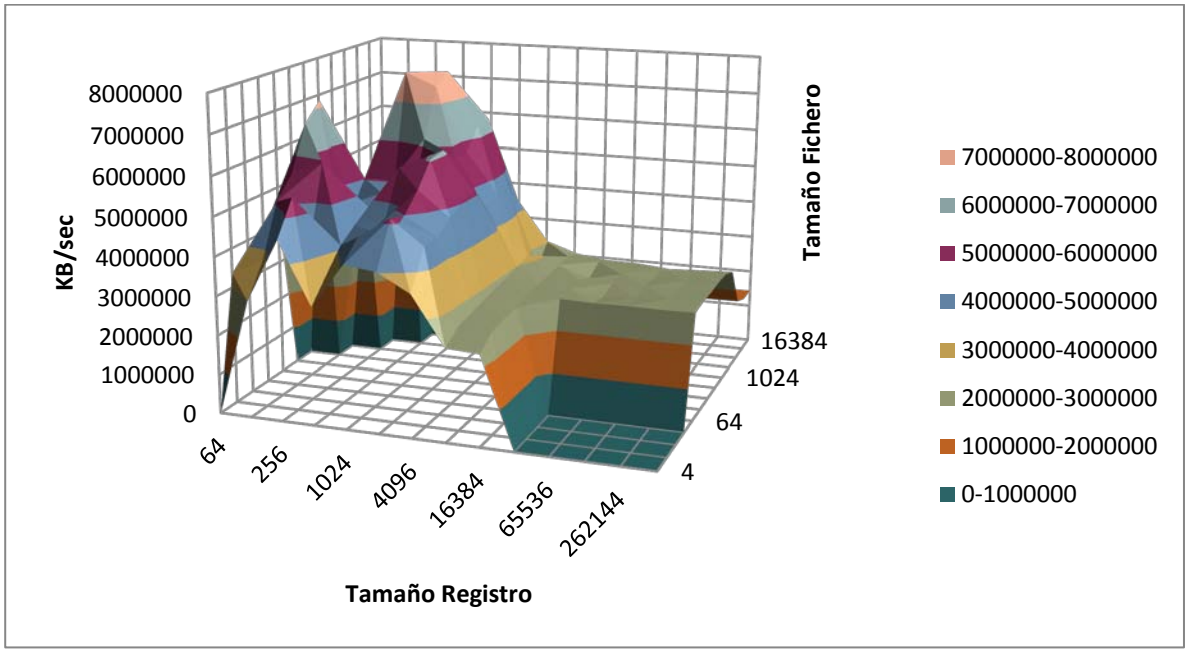


Figura 6. 16 Lectura: Acceso NFS NetApp FAS 3140

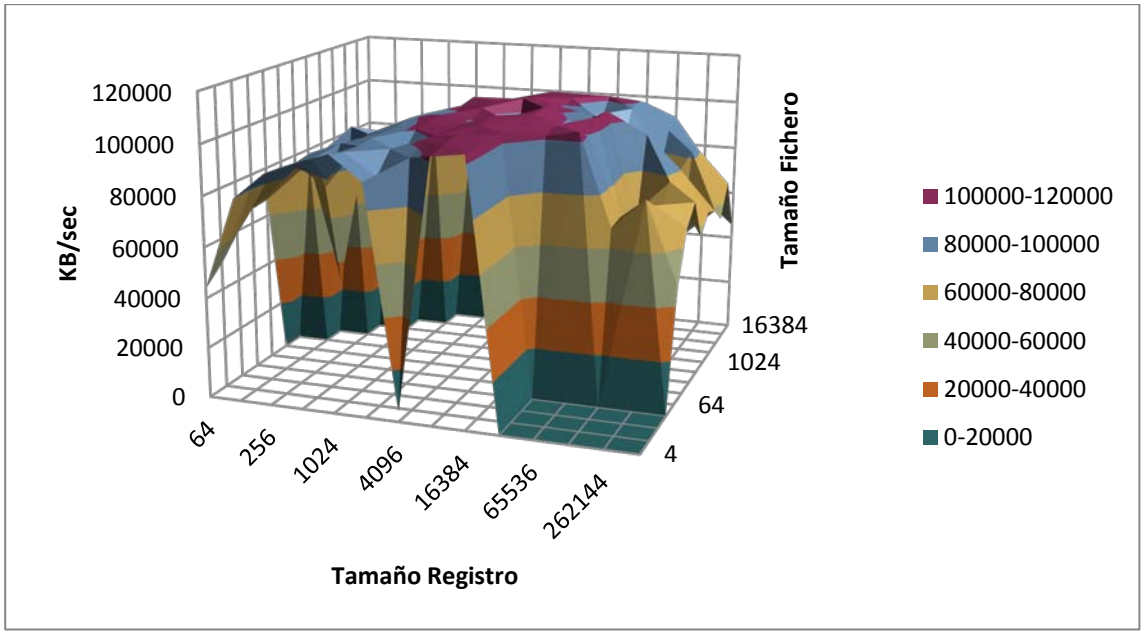


Figura 6. 17 Escritura: Acceso NFS NetApp FAS 3140

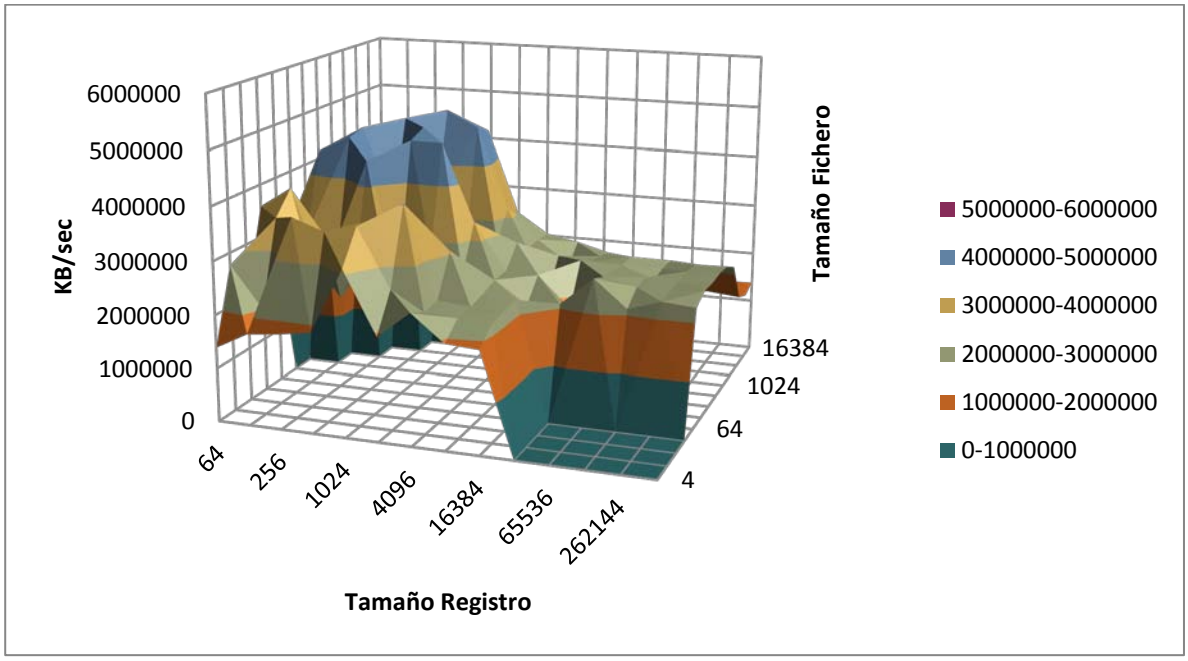


Figura 6. 18 Lectura: LUN FC 4GB en NetApp FAS 3140

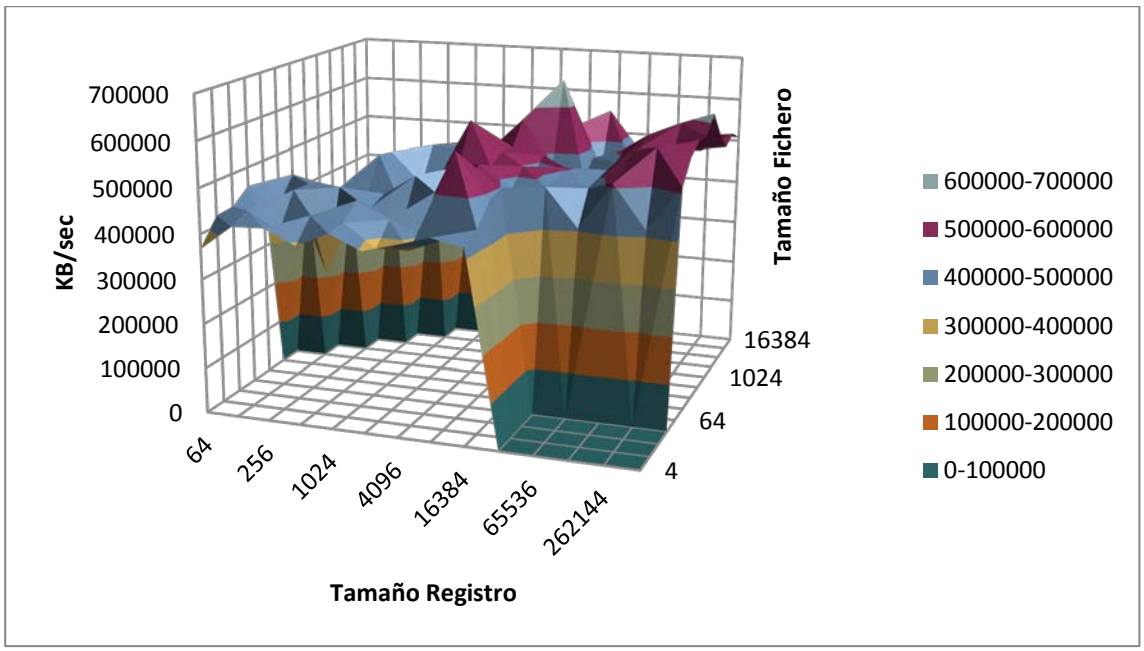


Figura 6. 19 Escritura: LUN FC 4GB en NetApp FAS 3140

6.4.2 Caché de datos en clientes

Para la obtención de los resultados experimentales mostrados en este apartado se ha usado un cluster con las siguientes características:

- Doce nodos con dos procesadores Quad-Core Intel Xeon E5450 de 3 GHz y 16GB de memoria RAM cada uno
- Tarjetas Infiniband MT25418 de Mellanox usando IPoIB para la conexión de los nodos.

La figura 6.20 [Díaz12] ilustra sobre la ventaja de usar la cache de datos. Los tiempos de lectura y escritura se reducen en un 97% aproximadamente si los datos se encuentran en la cache. Esta figura muestra las buenas prestaciones de la implementación de cache de datos en los clientes de AbFS, que aprovecha la cache de buffer de dispositivo de Linux. Las caches de metadatos y datos implementadas permiten escritura en la cache sin que el usuario tenga que intervenir para mantener coherencia ya que AbFS la mantiene.

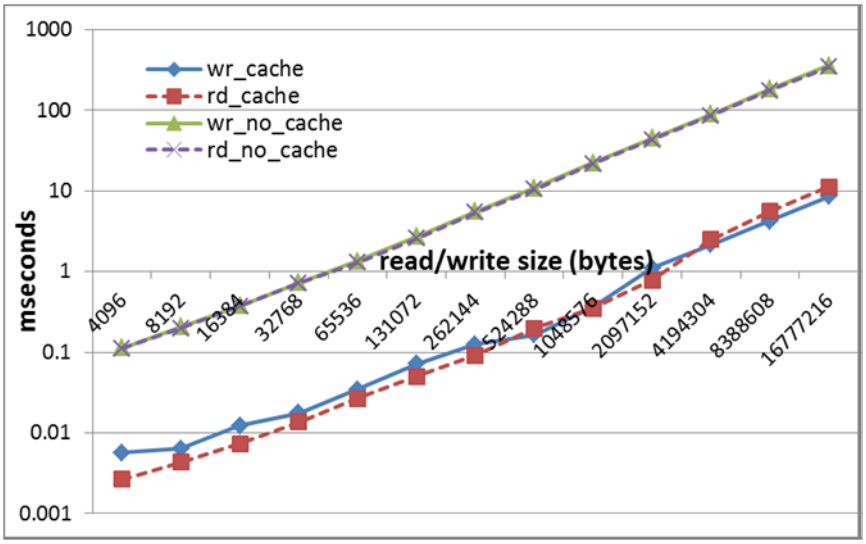


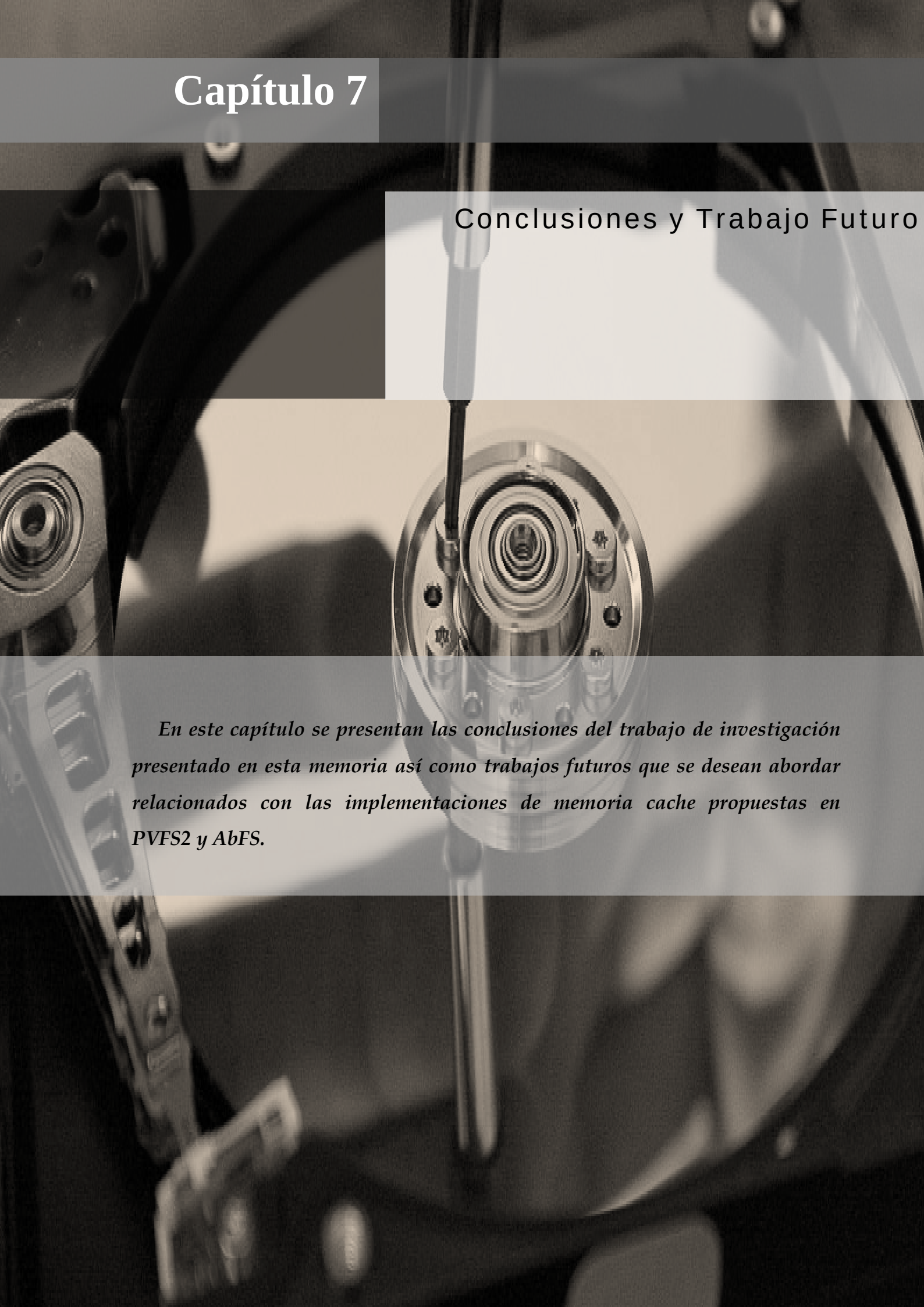
Figura 6. 20 Prestaciones de la cache de datos. Wr_no_cache y rd_no_cache son escrituras y lecturas respectivamente sin cache. wr_cache y rd_cache son aciertos de escrituras y lecturas respectivamente en la cache

Hay que destacar que los datos obtenidos con el benchmark IOzone el que exista más o menos servidores no afecta significativamente al rendimiento de la caché. Esto es gracias a que las operaciones se ejecutan de manera local en el cliente de AbFS reduciendo así el tráfico en la red. Cuando empiezan a escribir los clientes en los servidores también actúa la caché de los servidores y por ello apenas se nota el pésimo rendimiento de los discos de cada nodo (Figura 6.20).

Capítulo 7

Conclusiones y Trabajos Futuros

SUMARIO



Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo se presentan las conclusiones del trabajo de investigación presentado en esta memoria así como trabajos futuros que se desean abordar relacionados con las implementaciones de memoria cache propuestas en PVFS2 y AbFS.

7.1 Introducción

Este capítulo presenta las conclusiones obtenidas de la investigación realizada, así como los trabajos futuros que se pretenden abordar. A continuación se comentan las principales conclusiones y aportaciones que se desprenden del trabajo expuesto en esta memoria.

7.2 Conclusiones

A lo largo de este trabajo se han descrito las características, ventajas y desventajas de los sistemas de ficheros para entornos distribuidos y paralelos, y se ha centrado principalmente en la gestión de cache de datos.

Usar caches mejora las prestaciones porque reducen el tiempo de acceso a los datos por estar más cerca y, adicionalmente, en el caso de caches en clientes, también porque disminuyen los accesos al servidor y, por tanto, su congestión. En esta memoria se ha propuesto una implementación de cache de datos en los clientes para el sistema de ficheros paralelo PVFS2 y una implementación de cache de datos y de metadatos en cliente y en servidores para el sistema de ficheros distribuido AbFS.

Se realizaron pruebas, antes de iniciar las implementaciones de cache, que permitían ver la menor latencia en el acceso a datos almacenados en un sistema de ficheros local comparado con un sistema de ficheros que accede a los datos remotamente (Sección 6.2). Estos resultados permitieron deducir lo ventajoso de introducir una cache en los clientes e incentivaron la implementación de las caches propuestas en esta memoria.

Los resultados (Capítulo 6) de las propuestas de cache realizadas en los capítulos anteriores sobre el diseño de cache de datos en los clientes de PVFS2 (Capítulo 4) y AbFS

(Capítulo 5) ilustran que las implementaciones de memoria cache realizadas mejoran significativamente las prestaciones (latencia y ancho de banda) de las operaciones de Entrada/Salida de datos.

En el caso de PVFS2, los resultados muestran (Sección 6. 3) mejoras en los tiempos totales de las lecturas (y en el ancho de banda) de un 81% aproximadamente cuando hay un acierto en cache del cliente con una penalización muy reducida cuando no hay acierto en cache con respecto a la versión de PVFS2 sin cache (versión original de PVFS2). Para el caso de escrituras, al igual que en las lecturas, la cache también mejora significativamente su ancho de banda, en particular, un 90% aproximadamente. Si los datos no se encuentran ubicados en la cache, el rendimiento de las escrituras es menor, no obstante la diferencia con respecto a tener un acierto en cache es muy baja, la penalización se debe sólo a la gestión de la cache. Otro punto a destacar de la cache del cliente en PVFS2 es que es útil como buffer de escritura combinada (Sección 6.3.3). Es decir, se puede usar para combinar las escrituras a un fichero desde los diferentes núcleos (*cores*) de un nodo en una sola escritura hacia los servidores de E/S. Con la cache de datos el usuario no tiene que modificar los códigos que ejecuta añadiendo código extra que realice explícitamente la combinación de los datos que se van a escribir para lograr que se trasfieran al servidor en una trasferencia en lugar de tener que usar una por cada núcleo. También es útil como precaptador porque la lectura de uno de los núcleos de un fichero puede traer información que requieran otros núcleos, esos núcleos al leer pueden encontrar ya los datos en cache. Los procesadores multinúcleos son actualmente habituales, es importante por tanto hacer un uso efectivo del paralelismo de estos procesadores. Si los sistemas de ficheros aprovechan el hardware habitual en nuestros días, las aplicaciones pueden mejorar sus prestaciones.

En el caso de AbFS los resultados obtenidos de la implementación igualmente muestran la ventaja de usar cache para datos en los clientes (Sección 6.4). Los resultados referentes a los tiempos de lectura y escritura se reducen en un 97% aproximadamente si existe acierto de cache, es decir si estos se localizan dentro de la cache. AbFS usa, además de cache de datos, también caches de metadatos en los clientes. Las caches implementadas permiten escritura en la cache sin que el usuario tenga que intervenir para mantener coherencia ya que la implementación presentada mantiene coherencia.

La cache de datos y la cache de metadatos implementadas en AbFS comparten el mismo protocolo de mantenimiento de coherencia y aprovechan algunas de las estructuras de cache del VFS (*Virtual File System)* de Linux reduciendo, de esta forma, la complejidad añadida al sistema de ficheros implementado (es decir, no se necesitan nuevas estructuras o añadir capas adicionales para implementar las caches). Al contrario que la implementación realizada en PVFS2, en AbFS, las caches implementadas reducen la congestión de los servidores no sólo en el caso de realizar lecturas, también en el caso de las escrituras. El primer nodo que abre un fichero para escritura es el propietario del fichero y consigue reducir la congestión porque los clientes que escriben en el fichero combinan las escrituras en la cache del propietario en lugar de ir al servidor y porque los clientes leen, cuando hay propietario, de la cache del propietario en lugar de leer del servidor. En esta situación, es decir, cuando hay propietario, la cache tiene unas prestaciones similares a la cache cooperativa basada en home de [Hwang05].

El trabajo realizado en sistema de ficheros se recoge en diferentes publicaciones, que a continuación se enumeran comenzando por la más reciente:

- Antonio F. Diaz, Mancia Anguita, Hugo E. Camacho, Erik Nieto, Julio Ortega, "Two-level Hash/Table approach for Metadata Management in Distributed File Systems," accepted for publication in Journal of Supercomputing, 2012. DOI: 10.1007/s11227-012-0801-y

- Antonio F. Diaz, Mancia Anguita, Hugo E. Camacho, Erik Nieto, Julio Ortega, "AbFS: Sistema de ficheros abierto," in Actas De Las XXII Jornadas De Paralelismo, La Laguna (Tenerife, Islas Canarias), 2011, pp. 537-542.

- Antonio F. Diaz, Mancia Anguita, Erik Nieto, Hugo E. Camacho, Julio Ortega, "A metadata management implementation for a symmetric distributed file system" in Proceedings of the 11th International Conference on Mathematical Methods in Science and Engineering, Alicante, 2011, pp. 1289-1297.

- Antonio F. Diaz, Mancia Anguita, Hugo E. Camacho, Erik Nieto, Julio Ortega, "An Owner-base Cache Coherent Protocol for distributed file system" in Proceedings of the 11th International Conference on Mathematical Methods in Science and Engineering, Alicante, 2011.

- Hugo E. Camacho, Erik Nieto, Mancia Anguita, Antonio F. Diaz, Julio Ortega, "Client Cache for PVFS2" in: Proceedings of 1st International Conference on Parallel, Distributed and Grid Computing, Waknaghat, Solan, H.P., India, October 28–30, 2010, pp. 38–42.

- Erik Nieto, Hugo E. Camacho, Mancia Anguita, Antonio F. Diaz, Julio Ortega, "Fault Tolerant PVFS2 based on Data Replication," in: Proceedings of 1st International Conference on Parallel, Distributed and Grid Computing, Waknaghat, Solan, H.P., India, October 28–30, 2010, pp. 107–112.

- Erik Nieto, Raúl Hernández, Hugo E. Camacho, Antonio F. Diaz, Mancia Anguita, Julio Ortega, "Replicación de Datos en PVFS2 para Conseguir Tolerancia a Fallos". XX Jornadas de Paralelismo, en A Coruña, España celebradas del 16 a 18 de septiembre 2009.

7.3 Trabajo Futuro

En la implementación de cache en PVFS2 el mantenimiento de coherencia de la cache lo realiza con funciones invocadas por el programador. La implementación ofrece al programador una función flush o volcado para que explícitamente los bloques modificados sean enviados a los servidores de E/S. El programador debe usar esta función flush para enviar los datos a los servidores de E/S cuando existen bloques en la cache que han sido previamente modificados para que el último cliente que solicita los datos utilice la copia más actual de los mismos. En la función se establecen específicamente los datos que serán actualizados en los servidores de E/S. Al no usar protocolo de mantenimiento de coherencia las operaciones de

E/S son más rápidas que usando un protocolo. Como trabajo futuro se pretende desarrollar un protocolo de mantenimiento de coherencia para PVFS2.

En AbFS se pretende implementar distintas alternativas de gestión de coherencia y consistencia con el fin de testear sus ventajas e inconvenientes.

Apéndice

Memoria Compartida

SUMARIO

A. Memoria compartida

La memoria compartida es uno de los mecanismos agrupados bajo el nombre de Inter Process Comunication (IPC), junto con semáforos y colas de mensajes (FIFO) que se encuentran disponibles los sistemas Unix o linux. Mediante la utilización de una memoria compartida, se puede crear zonas de memoria compartidas por varios procesos. De este modo los cambios que un proceso realice a los valores almacenados en memoria compartida son visibles para los demás procesos que utilicen esa misma memoria compartida.

Este recurso IPC es el más rápido de los tres, ya que una vez conectados no necesitamos hacer mas llamadas al sistema, ni interactuar con el núcleo; directamente escribimos o leemos gracias un puntero que referencia a la memoria compartida. Para crear zonas de memoria compartida utilizamos la función **`shmget,`** la cual se encuentra definida en las cabeceras.

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

La función `shmget` se declara de la siguiente forma, **`int shmget (key_t clave, int tam, int flag);`**

Devuelve -1 en caso de error y un identificador de la zona de memoria compartida en caso de éxito. Ese identificador lo utilizaremos para trabajar con la zona de memoria compartida.

`key_t clave` – clave de la zona de memoria compartida.

`int tam` – tamaño en bytes de la zona de memoria compartida si queremos crearla. Si queremos acceder a un área ya creada, el tamaño será 0.

`int flag` – permisos.

Para la vinculación de memoria compartida, una vez que se ha creado, se requiere saber su dirección (observe que `shmget` no nos la indica). Para utilizar la memoria

compartida debemos antes vincularla con alguna variable de nuestro código. De esta manera, siempre que usemos la variable vinculada estaremos utilizando la variable compartida. Para establecer un vínculo utilizamos la función **`shmat.`**

```
void *shmat(int shmid, const void *shmaddr, int shmflag)
```

Devuelve la dirección de memoria de comienzo de la memoria compartida o -1 si hubo algún error.

`int shmid` – identificador de la memoria compartida obtenido con shmget.

`const void *shmaddr` – Dirección concreta donde reside la memoria compartida. No lo vamos a utilizar por lo que su valor siempre será NULL.

`int shmflag` – permisos. Por ejemplo. Aunque hayamos obtenido una zona de memoria con permiso para escritura o lectura, podemos vincularla a una variable para solo lectura. Si no queremos cambiar los permisos usamos 0.

El siguiente fragmento desvincula la memoria compartida vinculada.

```
r = shmdt(entero);
```

Esta función sólo libera el vínculo, pero no elimina la zona de memoria compartida. Además los vínculos se pierden al llamar a las funciones exit o exec (en sistemas Linux, en otras variantes de Unix esto puede no ser cierto). La memoria compartida reservada seguirá estando presente en el sistema hasta que no la borremos explícitamente. Para ello utilizamos la función `memctl` de una forma muy similar a como borramos grupos de semáforos o colas de procesos.

```
int shmctl(int shmid,int cmd, struct shmid_ds *buf)
```

Devuelve -1 si error, u otro valor en caso contrario.

`int shmid` – identificador de la memoria compartida obtenido con shmget.

`int cmd` - alguna de las siguientes constantes:

• **`IPC_STAT:`** Nos rellena la estructura shmid_ds() con los datos que maneja el núcleo en lo referente a la zona de memoria compartida.

• **`IPC_SET:`** Lo contrario de lo anterior establece la estructura que le pasamos como Parámetro en el kernel.

• **`IPC_RMID:`** Borra el recurso.

Algunos sistemas, como Linux o SunOS, soportan también las constantes **`SHM_LOCK`** y **`SHM_UNLOCK`** para bloquear o desbloquear el acceso a memoria compartida.

`struct sumid_ds *buf` - para el comando de borrado no es necesario el tercer argumento, por lo que utilizaremos `NULL`. La siguiente línea borra la zona de memoria compartida que hemos utilizado en el ejemplo anterior.

```
r = shmctl(shmid, IPC_RMID, NULL);
```

La operación de borrado sólo es efectiva si ningún proceso tiene vinculada la memoria compartida. Si aún existen vínculos a la memoria compartida, la operación de borrado se pospone hasta que desaparezca el último vínculo.

Bibliografía

En el siguiente aparatado, muestra la bibliografía utilizada para el desarrollado de esta investigación.

[AFS84] "What is AFS", http://www.openafsorg/

[Anguita11] Anguita L. Mancia, Diaz G. Antonio F, "Gestión de metadatos en sistemas de ficheros distribuidos v. 0.1" Informe Interno. Departamento de Arquitectura y Tecnología de Computadores. Grupo de investigación CASIP. Universidad de Granada, Granada, 2011.

[Braam02] P. J. Braam, "The Lustre Storage Architecture" Noviembre, 2002.

[Barkes98] Jason Barkes, Marcelo R. Barrios, Francis Cougard, Paul G. Crumley,Didac Marin, Hari Reddy, Theeraphong Thitayanun: "GPFS: A Parallel File System", http://www.redbooks.ibm.com/, Abril 1998.

[Brados98] Grez J. Brados: "An Enhanced Disk-Caching NFS implementation for Linux", 23 Abril 1998.

[Brados99] Grez J. Brados: "A Caching NFS Client for Linux", 27 Noviembre 1999.

[Brandt03] S. A. Brandt, E. L. Miller, D. D. E. Long and Lan Xue, "Efficient metadata management in large distributed storage systems," Proceedings 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, MSST, 2003, pp. 290-298.

[Camacho10] Camacho, Hugo E., Nieto Tovar, Erik., Diaz, Antonio F., Anguita, Mancia., Ortega, Julio, "Client Cache for PVFS2,"in: Proceedings of 1st International Conference on Parallel, Distributed and Grid Computing, Waknaghat, solan, H.P., India, Octubre 28–30, 2010. pp. 38-42.

[Carns00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, y R. Takhur: "PVFS:A Parallel File System for Linux Cluster", Extreme Linux Workshop, Atlanta, Octubre 2000.

[Carretero95] Diseño de un Sistema de Ficheros Paralelo para Maquinas Masivamente ParalelasJ_ CarreteroTR Number FIM_ 91.1/datsi/95, capitulo 4 "Mecanismos de incremento de prestaciones para sfp", 1995, pp. 140.

[Carretero96] J. Carretero, F. Pérez, P. De Miguel, F. García, L. Alonso: "ParFiSys: A Parallel File System for MPP", ACM SIGOPS, , Abril 1996, pp. 74-80.

[Ceph11] Ceph file system, web. http://ceph.newdream.net/

[Corbett96] Peter F. Corbett ,Dror G. Feitelson, The Vesta parallel file system, ACM Transactions on Computer Systems (TOCS), v.14 n.3, Agosto 1996, pp.225-264.

[Corbett01] P. F. Corbett and D. G. Feitelson, "The vesta parallel file system," in High Performance Mass Storage and Parallel I/O: Technologies and Applications, H. Jin, T. Cortes and R. Buyya, Eds. New York, NY: IEEE Computer Society Press and Wiley, 2001, pp. 285-308.

[Clusterfs07] "Lustre - Scalable Storage", http://www.clusterfs.com/

[Dahlin94] Dahlin, M., Wang, R., Anderson, T., and Patterson, D. "Cooperative Caching: Using remote client memory to improve file system performance", In Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation. USENIX Assoc., Berkeley, CA, , 1994, 267-280.

[Díaz12] Diaz, Antonio F., Anguita, Mancia, Camacho, Hugo E., Nieto, Erik., Ortega, Julio, "Two-level Hash/Table approach for Metadata Management in Distributed File Systems," accepted for publication in Journal of Supercomputing, 2012.

[Díaz11a] Diaz, Antonio F., Anguita, Mancia., Camacho, Hugo E., Nieto Tovar, Erik., Ortega, Julio, "AbFS: Sistema de ficheros abierto," in Actas De Las XXII Jornadas De Paralelismo, La Laguna (Tenerife, Islas Canarias), 2011a, pp. 537-542.

[Díaz 11b] Diaz, Antonio F., Anguita, Mancia., Nieto Tovar, Erik., Camacho, Hugo E., Ortega, Julio,"A metadata management implementation for a symmetric distributed file system," in Proceedings of the 11th International Conference on Mathematical Methods in Science and Engineering, Alicante, 2011b, pp. 1289-1297.

[Díaz11c] Diaz, Antonio F., Anguita, Mancia., Camacho, Hugo E., Nieto Tovar, Erik., Ortega, Julio, "An Owner-base Cache Coherent Protocol for distributed file system" in Proceedings of the 11th International Conference on Mathematical Methods in Science and Engineering, Alicante, 2011c.

[ext11] EXT3. http://es.wikipedia.org/wiki/Ext3, Agosto 2011

[Fagin79] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, "Extendible hashing---a fast access method for dynamic files," ACM Trans.Database Syst., vol. 4, 1979, pp. 315-344.

[Farazdel99] Abbas Farazdel, Gonzalo R. Archondo-Callao, Eva Hocks, TakaakiSakachi, Federico Vagnini: "Understanding and Using the SP Switch", Abril 1999. http://www.redbooks.ibm.com/

[García96] F. García, J. Carretero, F. Pérez, P. de Miguel, L. Alonso: "Coherencia de Cache en Sistemas de Ficheros Paralelos", VII Jornadas de Paralelismo. Santiago de Compostela. . 1996, pp. 215-228.

[García98] Félix García Carballeira: "Diseño de Protocolos de coherencia de cache en sistemas de ficheros distribuidos y paralelos", TR. Number FIM/107.1/DATSI/98.

[García03] Félix GarcíaCarballeira, Alejandro Calderón, JesúsCarretero, Javier Fernández, José M. Perez: "The Design of the EXPAND Parallel File System" in The International Journal of High Performance Computing Applications, Volume 17, No. 1, pp. 21–37, Primavera 2003.

[García04] Félix GarcíaCarballeira, JesúsCarretero, Alejandro Calderón, José M. Perez, José D. García: "An Adaptative Cache Coherence Porotocol Specification for Parallel Input/Output System". IEEE Transactions on Parallel and Distributed System, Volume 15, No. 6, Junio 2004. pp. 533–545.

[Google03] Sanjay Ghemawat, Howard Gobioff y Shun-Tak Leung: "The Google File System". Bolton Landindg, New York, USA. SOSP´03, 19-21 Octubre 2003.

[HFS04] HFS PLUS Volume Format, reportetécnico(Technical Note TN1150)http://developer.apple.com/library/mac/#technotestntn1150.html, marzo 2004

[Hwang04] In-Chul Hwang, Hojoong Kim, Hanjo Jung, Dong-Hwan Kim, HojinGhim, SeungRyoulMaeng, Jung Wan Cho: "Design and Implementation of the Cooperative Cache for PVFS", International Conference on Computational Science 2004, pp 43-50.

[Hwang05] In-Chul Hwang, Seung-RyoulMaeng, Jung-Wan Cho, Home-based Cooperative Cache for parallel I/O applications; in Future Generation Computer Systems 22, 25 October 2005 pp 633-642.

[IOzone] http://www.iozone.org/

[Jones00] Terry Jones, Alice Koniges, R. Kim Yates: "Performance of the IBM General Parallel File System", Parallel and Distributed Processing Symposium 2000.

[Kernel] http://www.linux-es.org/kernel

[Kon96] Fabio Kon: "Distributed File System Past, Present and Future A Distributed File System for 2006", 6 Marzo 1996.

[Kondekar09] P. Kondekar, MDS Performance Analysis. Sun Microsystems, 2009.

[Kuhn07] Kuhn,“Directory-Based Metadata Optimizations for Small Files in PVFS”.Parallele und VerteilteSysteme.Institut fur Informatik. Ruprecht-Karls-Universit at Heidelberg, 2007.

[Kunkel06] Julian Martin Kunkel: “Thesis: Performance Analysis of the PVFS2 Persistency Layer”, Febrero 2006.

[Kunkel07] Julian Martin Kunkel. “Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration Master´s thesis”, Ruprecht-Karls – Universität Heidelberg Institutfür Informatik Arbeits gruppe Parallele und Verteilte Systeme, Agosto 07, pp. 14-16

[Kunkel07b] J. M. Kunkel and T. Ludwig, "Performance evaluation of the PVFS2 architecture," in 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, 2007, pp. 509-516.

[Lafuente11] Alberto Lafuente, “Sistemas de fichero distribuidos”, capítulo 1, pagina 5-9. http://www.sc.ehu.es/acwlaroa/SDI/ Apuntes/Cap1.pdf

[Ligon99] W. B. Ligon III, R. B. Ross, D. Becker, P. Merkey, "Beowulf: Low-Cost Supercomputing Using Linux," IEEE Software magazine special issue on Linux, January, 1999, Volume 16, Number 1, pp 79.

[Ligon03] Walt Ligon and Rob Ross, "Parallel I/O and the Parallel Virtual File System," Beowulf Cluster Computing with Linux, 2nd Edition, William Gropp, Ewing Lusk, and Thomas Sterling, editors, MIT Press, Noviembre 2003, pp 489-530.

[Linux] http://www.linux-es.org/sobre_linux

[Lustre07] “What is Lustre?”,http://wiki.lustre.org/

[mdtest] http://sourceforge.net/projects/mdtest/

[Microsoft05] Introducción a los sistemas de archivos FAT, HPFS y NTFS. Id. de artículo: 100108 - Versión: 5.1. http://support.microsoft.com/kb/100108/es, Agosto 2005.

[MPICH2] http://www.mcs.anl.gov/research/projects/mpich2/

[Nelson88] Michael N. Nelson, Brent B. Welch, John K. Ousterhout: “Caching in the Sprite Network File System”, Febrero 1988.

[Nils96] Nils Nieuwejaar, David Kotz: “Performance of the Galley Parallel File System”, IOPADS´96, Philadelphia PA, USA, Mayo 1996, pp. 83-94.

[Oracle11] Oracle OCFS file system. Ultimo acceso (mm/dd/aaaa): 09/04/2011. http://oss.oracle.com/projects/ocfs2/.

[OrangeFS12] http://www.pvfs.org/cvs/pvfs-2-8-branch.build/doc/pvfs2-faq/

[Ousterhout87] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, Brent B. Welch : " The Sprite Network Operating System", Noviembre 1987.

[Pawlowski00] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, Robert Thurlow: "The NFS Version 4 Protocol", Marzo 2000,http://www.nfsv4.org

[Patterson88] David A Patterson, Garch Gibson y Randy H Katz: " A Case for redundant Arrays of Inexpensive Disks (RAID)", pp.109-116, Junio 1988.

[Polyserve10] "HP PolyServe matrix server 4.1.0 administration guide," Hewlett-Packard Development Company, L.P., Tech. Rep. Part Number: T5392-96073, 2010.

[POS01] POSIX-1-2001 (IEEE Std 1003.1-2001)

[POS08] POSIX.1-2008 (IEEE Std 1003.1-2008)

[PVFS2] PVFS2 Team: "Parallel Virtual File System, Version 2", Septiembre 2003, http://www.pvfs.org/pvfs2-guide.html.

[Ramachandran02] Harish Ramachandran. "Design and implementation of the system interface for PVFS2". Master's thesis, Clemson University, Clemson, SC, Diciembre 2002.

[Ross02] Robert B. Ross, Philip H. Carns, Walter B. Ligon III, Robert Latham: "Using the Parallel Virtual File System", Julio 2002. http://www.parl.clemson.edu/pvfs/

[Russel85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, Bob Lyon: "Design and Implementation of the Sun Network File System", Osenix´85, Verano 1985, pp 119-130.

[Satyanarayanan89] M. Satyanarayanan: "Coda: a Highly Avaliable File System for a Distributed Work Environment", Proceedings of the Second IEEE Workshop on Workstation Operating Systems, Pacific Grove, CA , Septiembre 1989.

[Schmuck02] Frank Schmuck y Roger Haskin: “GPFS: A Shared-Disk File System for Large Computing Clusters”, FAST´02, Monterrey CA, 28-30 Enero 2002, pp 231-244.

[SGI11] SGI CXFS file system web. Ultimo acceso (mm/dd/aaaa): 09/04/2011.http://www.sgi.com/products/storage/software/cxfs.html.

[SGI Corp.11] "CXFS 6. SGI InfiniteStorage shared filesystem: High peformance shared filesystem," Silicon Graphics International Corp., 2011.

[Shirriff96] Ken Shirriff: “The Sprite Operating System”, http://www.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html, Marzo 1996.

[SL] http://www.hispalinux.es/SoftwareLibre

[Stolarchuk92] Michael T. Stolarchuk: “Faster AFS”, CITI Technical Report 92-93, 22 Junio 1992.

[Soltis96] "The global file system," in Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems and Technologies, 1996, pp. 319-342.

[Tanenbaum02] Andrew S. Tanenbaum, Maarten Van Steen: “Capítulo 10, Distributed File Systems”, Prentice Hall, 2002.

[Top500] “The 29th TOP500 List was released in Dresden, Germany during ISC07”. Junio 2007

[Turek10] W. Turek and P. Calleja, High Performance, Open Source, Dell Lustre Storage System. White Paper. Dell - University of Cambridge, 2010.

[Vilayannur02] M.Vilayannur, M. Kandemir, A. Sivasubremaniam: “Kernel-Level Caching for Optimizing I/O by Exploting Inter -Application Data Sharing”, IEEE International Conference on Cluster Computing (Cluster´02), Septiembre 2002.

[Weil06] Sage A. Weil ,"Ceph: A scalable, high-performance distributed file system," in OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, Washington, 2006, pp. 307-320.

[Weil07] Sage A. Weil, “CEPH: Reliable, scalable, and high performance distributed storage”, Doctor Thesis, in the University of California Santa Cruz, chapter. 3 page 18. December 2007

[Xiong06] M. Xiong, H. Jin and S. Wu, "FDSSS: An efficient metadata management scheme in large scale data environment," in Fifth International Conference on Grid and Cooperative Computing Workshops, GCCW '06, 2006, pp. 71-77.