PROYECTO FIN DE CARRERA

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE FICHEROS EN MPI SOBRE FUSE

Autor: Javier Escobar Guisado

Tutor: Francisco Javier García Blas

Leganés, Octubre de 2010

Título: DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE FICHEROS EN MPI SOBRE FUSE
Autor: Javier Escobar Guisado
Director: Francisco Javier García Blas

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de _____ de 2010 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Resumen

En la actualidad la mayoría de los supercomputadores pertenecientes al Top500 utilizan sistemas de ficheros como PVFS, GPFS o Lustre. Estos sistemas de ficheros son usados mayoritariamente en aplicaciones de procesamiento de datos y visualización, intensas en datos. Sin embargo, estos sistemas presentan numerosas complicaciones a la hora de instalar y administrar.

Este Proyecto Fin de Carrera presenta el diseño, implementación y evaluación de un sistema de ficheros basado en AHPIOS y FUSE. AHPIOS es el primer sistema de E/S salida paralela implementado en MPI. FUSE ofrece a los usuarios la posibilidad de desarrollar nuevos sistemas de ficheros basados en POSIX.

En este trabajo se muestra como partiendo de un sistema de E/S paralela basado en MPI, se puede ofrecer una interfaz de fichero basada en POSIX. Para este fin, se ha usado la herramienta FUSE. En el capítulo de evaluación se demuestra que el prototipo implementando obtiene buenos resultados en la mayoría de los casos. Para comprobar el rendimiento del sistema propuesto se han usado distintas herramientas, basadas en aplicaciones usadas frecuentemente por la comunidad científica.

Palabras clave: Entrada/salida paralela, computación de altas prestaciones, paralelismo.

Abstract

Recent advances in storage technologies and high performance interconnects have made possible in the last years to build, more and more potent storage systems that serve thousands of nodes. The majority of storage systems of clusters and super-computers from Top 500 list are managed by one of three scalable parallel file systems: GPFS, PVFS, and Lustre.

The following Master Thesis presents the design, implementation, and evaluation of a parallel file system based on AHPIOS file system and FUSE. AHPIOS is the first I/O system implemented in MPI. FUSE offers users the possibility to develop new file systems based on the POSIX interface.

In this dissertation we show how starting from an existing MPI-based parallel I/O system, we are able to provide a POSIX-based interface through the FUSE tool-kit. The evaluation section shows that our implemented prototype outperforms the rest of solutions in most of the cases. In order to perform the experimental evaluation, we have been used different well-known *benchmarks*, which are used by the scientific community.

Keywords: Parallel I/O, high performance computing, parallelism.

Agradecimientos:

En primer lugar quiero agradecer a mi tutor Javier, por haber estado siempre que le he necesitado y por toda la ayuda e interés que ha mostrado, ya que sin él no hubiera sido posible la realización de este Proyecto Fin de Carrera.

Agradecer también a Elena, por estar ahí siempre que la he necesitado, liberándome de multitud de tareas que me permitieron compatibilizar familia, casa, trabajo y Universidad durante estos últimos años. Ante mis hijas Candela y Rocío siempre supo explicarles el tiempo que las robaba y por ello también las quiero agradecer su incondicional comprensión.

A mis padres por la motivación que siempre me brindaron para que continuase con este segundo ciclo y que en los momentos más duros siempre me recordaba que de alguna forma les debía el terminar estos estudios.

A mis compañeros y amigos de clase, que me han permitido descubrir nuevas amistades que he mantenido incluso fuera del ámbito Universitario.

Por último y no menos importante, a todos mis amigos por su apoyo y comprensión durante el tiempo empleado en los estudios del segundo ciclo.

No puedo olvidar a mis hermanos que siempre me han estado alentándome, aunque eso sí, cada uno a su manera.

A todos los que me he dejado sin mencionar y que también les tengo muy presentes, les agradezco todo su apoyo y la alegría mostrada cuando ven que finalmente dejo terminado este capítulo de mi vida.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

En este apartado se describe la motivación y los objetivos del Proyecto Fin de Carrera, aquellos términos y definiciones del ámbito del proyecto, así como la estructura que seguirá el documento.

1.1 Motivación

En la actualidad, cada vez es más necesaria la ejecución paralela de aplicaciones con el objetivo de aumentar su rendimiento. De esta forma se consigue resolver problemas grandes de una forma más rápida y económica. Para poder cumplir con las exigencias de cálculo actuales, es necesario abandonar el clásico sistema centralizado de computación por un sistema de computación paralela. Los sistemas cluster permiten que

las computadoras que lo integran trabajar en paralelo como si fuera una sola, aumentando considerablemente la capacidad de cómputo.

Uno de los problemas más importantes a los que se enfrenta la computación paralela de hoy en día, es la llamada "crisis de E/S". Este problema viene dado por el cuello de botella que producen los grandes sistemas de E/S. Este cuello de botella puede afectar considerablemente al tiempo de ejecución de las aplicaciones paralelas como han demostrado distintas publicaciones científicas.

Existen distintas soluciones para reducir la latencia en los accesos de E/S. Una de ellas, y sobre la que se centrará este Proyecto Fin de Carrera, es el sistema escalable de E/S AHPIOS. AHPIOS permite virtualizar sistemas de ficheros paralelos y locales. El mayor problema que presenta actualmente este sistema es que solo se ofrece una interfaz de acceso basada en MPI-IO. No todas las aplicaciones intensas en datos acceden a los ficheros mediante la interfaz MPI-IO, sino que usan otras interfaces muy extendidas como *POSIX [40]*. A pesar de que la interfaz MPI-IO ofrece ventajas frente al resto de interfaces, POSIX sigue siendo la solución más usada en la comunidad científica.

El objetivo principal de este Proyecto Fin de Carrera será el desarrollo de un sistema que permita de una forma sencilla y con las menores restricciones posibles, explotar al máximo la E/S paralela para los supercomputadores mediante la utilización del estándar *POSIX*, facilitando el acceso a los usuarios finales con la utilización de la herramienta FUSE[1]. El sistema tendrá que hacer totalmente transparente a los usuarios de MPI [17] el acceso de una interfaz paralela mediante *POSIX*, además de incorporar todas las ventajas que ofrece el sistema de E/S paralela AHPIOS.

1.2 Objetivos

A continuación se describen brevemente los objetivos que se han de alcanzar para la realización del presente proyecto:

- Adaptación de un sistema de E/S paralela sobre MPI a la interfaz *POSIX* mediante la herramienta FUSE.
- Estudio de FUSE y de las distintas posibilidades de implementación que posee. Estudiar y analizar el sistema de E/S paralela AHPIOS y conocer todas sus posibilidades de funcionamiento y posibles métodos de adaptación.
- Ampliación del conjunto de metadatos existente, manteniendo en la medida de lo posible, la compatibilidad con la interfaz de *POSIX*.
- Diseño e implementación del despliegue de los servidores de E/S de forma estática.
- Estudiar y cuantificar la sobrecarga del sistema basado en FUSE sobre el sistema original de E/S AHPIOS.
- Evaluar el sistema implementado y averiguar cuál es la configuración más recomendable en función de las necesidades del usuario.

1.3 Definiciones y acrónimos

A continuación, se definen los tecnicismos y la nomenclatura utilizada.

Definiciones

- *Benchmark*: es una técnica utilizada para medir el rendimiento de un sistema o componente de un sistema, frecuentemente en comparación con el cual se refiere específicamente a la acción de ejecutar un *benchmark* [24].
- *Kernel:* hace referencia al núcleo de un sistema operativo.
- *Cluster*: conjuntos o conglomerados de computadoras construidos mediante la utilización de componentes de *hardware* comunes y que se comportan como si fuesen una única computadora.

- *Middleware*: es un *software* de conectividad que ofrece un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas.
- *Round-Robin*: es un método de planificación para seleccionar todos los elementos en un grupo de manera equitativa y en un orden, comenzándose normalmente por el primer elemento de la lista hasta llegar al último y volviendo a empezar desde el primero.

Acrónimos

Acrónimo	Significado
ADIO	*Abstract-Device Interface for E/S.*
API	*Application Program Interface.* Interfaz de aplicación.
AVFS	*A Virtual FileSystem.* Un Sistema de Ficheros Virtual.
DRAM	*Dynamic Random Access Memory.*
ENOENT	*Error NO ENTry (No such file or directory).* Fichero o directorio no encontrado.
EOF	*End Of File.* Fin de fichero.
EXT3	*Extension 3.* Sistema de ficheros Extension versión 3.
ETYPE	*Elemental Type.* Tipo de datos elemental.
GPFS	*General Parallel File System*. Sistema de archivos virtuales paralelos.
HPCBP	*High Performance Computing Basic Profile.*
GPL	*General Public License.*
MPI-IO	Interfaz de E/S de ficheros paralelos para MPI.
MPICH	Implementación portable y gratuita de MPI.
NFS	*Network File System*. Sistema de archivos de red.
OGF	*Open Grid Forum.*
POSIX	*Portable Operation System Interface.*

Acrónimo	Significado
PVFS	*Parallel Virtual File System*.
PVM	*Parallel Virtual Machine.*
ROMIO	Implementación portable de MPI-IO, de altas prestaciones.
RPC	*Remote Procedure Call*. Llamada de procedimientos remotos.
SDRAM	*Static Random Access Memory.*
SO	Sistema Operativo.
VFS	*Virtual File System.* Sistema de ficheros virtual que permite unificar de cara al usuario todos y cada uno de los posibles tipos de estructuras de almacenamiento de datos.

Tabla 1: Definiciones y Acrónimos

1.4 Estructura del documento

En este apartado se detalla cada uno de los capítulos de los que se compone este documento.

En el Capítulo 1, *INTRODUCCIÓN*, se presenta el proyecto, proporcionando un breve resumen y los objetivos del mismo.

En el Capítulo 2, *ESTADO DE LA CUESTIÓN*, se proporciona una visión general de la tecnología que engloba el proyecto.

En el Capítulo 3, *FUSE*, se proporciona una visión general sobre este sistema de ficheros en el espacio de usuario.

En el Capítulo 4, *SISTEMA DE E/S PARALELA AHPIOS*, se proporciona una visión sobre el funcionamiento, desarrollo e implementación del sistema de ficheros.

En el Capítulo 5, *ANÁLISIS Y DISEÑO*, se realiza un estudio preliminar del sistema que se desea crear atendiendo a las necesidades del usuario.

En el Capítulo 6, *DETALLES DE IMPLEMENTACIÓN* se explica cómo se ha realizado la adaptación e implementación del nuevo sistema (pasos, dificultades encontradas, etc.).

En el Capítulo 7, *VERIFICACIÓN DEL SISTEMA*, se especifica un conjunto de pruebas básicas que se deben realizar una vez desplegado el nuevo sistema para verificar su correcto funcionamiento.

En el Capítulo 8, *EVALUACIÓN*, se especifican las técnicas utilizadas para realizar los test de rendimiento al sistema. Además, se estudiará el rendimiento del sistema y se cuantificará su rendimiento en factor de los resultados obtenidos.

En el Capítulo 9, *CONCLUSIONES Y TRABAJOS FUTUROS*, se realizará un balance de los objetivos logrados y se expondrán las conclusiones obtenidas de la realización del proyecto.

En el Capítulo *BIBLIOGRAFÍA*, se expondrá la bibliografía usada en el proyecto.

En el *Apéndice I*, titulado *INSTALACIÓN Y CONFIGURACIÓN DE FUSE*, se especificarán los requisitos mínimos para la instalación y ejecución de la herramienta FUSE.

Finalmente, en el *Apéndice II*, titulado *CONFIGURACIÓN DE NFSSTONE*, se especificarán los requisitos mínimos para la instalación y ejecución de la herramienta de *benchmark* NFSSTONE.

1.5 Características del documento

En este punto es necesario entablar las particularidades de este documento. Este documento tiene las siguientes características:

- Los títulos vendrán en mayúsculas, negrita y con letra Times New Roman, con un tamaño de 16, excepto los títulos de primer nivel que tendrán un tamaño de 24.
- El resto de texto vendrá dado en letra Times New Roman con un tamaño de 12 puntos y un interlineado de 1,5.
- Todas las figuras vendrán numeradas y con su correspondiente título descriptivo que será el que aparezca en el índice correspondiente.
- Todas las tablas vendrán numeradas y con su correspondiente título descriptivo que será el que aparezca en el índice correspondiente.

2 ESTADO DE LA CUESTIÓN

En este apartado se proporciona una visión general de las tecnologías que engloba el proyecto. Para ello se describe de forma detallada las distintas tecnologías disponibles actualmente con el fin de utilizar la información y funcionalidad proporcionada por las diversas tecnologías para realizar un diseño eficiente del proyecto.

En primer lugar se realiza una exposición general de la tecnología cluster, estudiándose posteriormente los sistemas de ficheros distribuidos y más concretamente los sistemas de ficheros paralelos. Al tratarse el tema de paralelismo se analizará el estándar MPI (*Message Passing Interface*) y las posibles herramientas utilizadas para analizar las aplicaciones paralelas.

2.1 Clusters

El origen del término y uso de este tipo de tecnología es desconocido pero se puede considerar que comenzó entre finales de los años 50 y principios de los años 60.

La base formal del término, en cuanto a Ingeniería Informática se refiere, está relacionado con trabajos en paralelo de cualquier tipo. En 1967, Gene Amdahl publicó lo que ha llegado a ser considerado como el inicio del procesamiento paralelo: la Ley de Amdahl [2]. Esta ley que describe matemáticamente el aceleramiento que se puede esperar paralelizando cualquier serie de tareas al ser aplicadas sobre una arquitectura paralela.

La historia de los primeros grupos de computadoras está fuertemente ligada a la historia de los inicios de las redes. Una de las principales motivaciones para el desarrollo de una red era enlazar los distintos recursos que ofrece un conjunto de computadoras.

Utilizando el concepto de red de conmutación de paquetes, el proyecto ARPANET [3] logró crear en 1969 lo que fue posiblemente la primera red de ordenadores básica basada en un cluster de ordenadores formado por cuatro tipos de centros informáticos. El proyecto ARPANET creció y se convirtió en lo que es ahora Internet, que se puede considerar como "la madre de todos los clusters" (como la unión de casi todos los recursos de cómputo, incluidos los clusters, que pasarían a ser conectados).

También estableció el paradigma de uso de computadoras cluster en el mundo de hoy que consiste en el uso de las redes de conmutación de paquetes para realizar las comunicaciones entre procesadores localizados en los marcos de otro modo desconectado.

La construcción de PC's por los clientes y grupos de investigación se desarrolló a la vez que la organización de las redes y el sistema operativo Unix desde principios de la década de los años 70. Estos desarrollos dieron sus frutos, sacando a la luz proyectos

como TCP/IP [4] y el proyecto de la Xerox PARC [7] formado por protocolos basados en redes de comunicaciones. El núcleo del SO fue construido en 1971.

Sin embargo, no fue hasta alrededor de 1983, cuando los protocolos y herramientas para el trabajo remoto, comenzaron a facilitar la distribución y uso compartido de archivos y recursos (en gran medida dentro del contexto de BSD Unix [5], e implementados por Sun Microsystems).

El primer producto comercial de tipo cluster fue ARCnet [6], desarrollado en 1977 por Datapoint. No obtuvo un éxito comercial y los cluster no consiguieron tener éxito hasta que en 1984 VAXcluster [8] fue desarrollado con el sistema operativo VAX/VMS. El ARCnet y VAXcluster no sólo son productos que apoyan la computación paralela, sino que además también comparten los sistemas de archivos y dispositivos periféricos. La idea era proporcionar las ventajas del procesamiento paralelo, al tiempo que se mantiene la fiabilidad de los datos y el carácter singular.

Otros dos principales clusters comerciales desarrollados más adelante fueron el Tandem Himalaya [9] (alrededor de 1994 con productos de alta disponibilidad) y el IBM S/390 Parallel Sysplex [10] (también alrededor de 1994, principalmente destinado a empresas).

La historia de los clusters de ordenadores no estaría completa sin señalar el papel fundamental desempeñado por el desarrollo del *software* de Parallel Virtual Machine (PVM) [11]. Este *software* de código abierto y basado en comunicaciones TCP/IP permitió la creación de un superordenador virtual, un cluster de computación de alto rendimiento (High Performance Computing). De forma libre los clusters heterogéneos han constituido la cima de este modelo logrando aumentar rápidamente en Flops y superando con creces la disponibilidad incluso de los más caros superordenadores.

En 1995, la invención de la arquitectura de tipo "*Beowulf*"[12], una granja de computación diseñado en base a un producto básico de la red con el objetivo específico de "ser un superordenador" capaz de realizar de forma eficiente cálculos paralelos. Esta invención estimuló el desarrollo independiente de la computación Grid como una

entidad, a pesar de que el estilo Grid giraba en torno al del sistema operativo Unix y el Arpanet.

Beneficios de la Tecnología Cluster

Las aplicaciones paralelas escalables requieren: buen rendimiento, comunicaciones que dispongan de gran ancho de banda y baja latencia, redes escalables y acceso rápido a los archivos. Un cluster puede satisfacer estos requisitos usando los recursos que tiene asociados a él.

Los cluster ofrecen las siguientes características a un costo relativamente bajo:

- Alto Rendimiento.
- Alta Disponibilidad.
- Alta Eficiencia.
- Escalabilidad.

La tecnología cluster permite a las organizaciones incrementar su capacidad de procesamiento usando tecnología estándar, tanto en componentes de *hardware* como de *software*, que pueden adquirirse a un costo relativamente bajo.

Clasificación de los Clusters

El término cluster tiene diferentes connotaciones para diferentes grupos de personas. Los tipos de clusters, establecidos en base al uso que se dé a los clusters y los servicios que ofrecen, determinan el significado del término para el grupo que lo utiliza. Los clusters pueden clasificarse en base a sus características. Los tres tipos de cluster que se pueden tener son:

Alto rendimiento: Son clusters en los cuales se ejecutan tareas que requieren de gran capacidad computacional, grandes cantidades de memoria, o ambos a la vez. El llevar a cabo estas tareas puede comprometer los recursos del cluster por largos periodos de tiempo.

Alta disponibilidad: Son clusters cuyo objetivo de diseño es el de proveer disponibilidad y confiabilidad. Estos clusters tratan de brindar la máxima disponibilidad de los servicios que ofrecen. La confiabilidad se provee mediante *software* que detecta fallos y permite recuperarse frente a los mismos, mientras que en *hardware* se evita tener un único punto de fallos.

Alta eficiencia: Son clusters cuyo objetivo de diseño es el ejecutar la mayor cantidad de tareas en el menor tiempo posible. Existe independencia de datos entre las tareas individuales. El retardo entre los nodos del cluster no es considerado un gran problema.

Los clusters pueden ser clasificados también como Clusters de IT Comerciales (Alta disponibilidad, Alta eficiencia) y Clusters Científicos (Alto rendimiento). A pesar de las discrepancias a nivel de requerimientos de las aplicaciones, muchas de las características de las arquitecturas de *hardware* y *software* que están por debajo de las aplicaciones en todos estos clusters, son las mismas. Más aún, un cluster de determinado tipo, puede también presentar características de los otros.

2.1.1 Cluster de tipo beowulf

El cluster encuadrado en la tipología *Beowulf* posee una arquitectura basada en multicomputadoras y suele ser utilizado para computación paralela. Este sistema consta de un nodo maestro y uno o más nodos esclavos conectados a través de una red Ethernet u otra topología de red. Esta construido con componentes de *hardware* comunes en el mercado, similar a cualquier PC, y adaptadores de Ethernet y switches estándares. Como no contiene elementos especiales es fácilmente reproducible, y por tanto es sencillo implementarlo con unos mínimos recursos básicos.

Una de las diferencias principales entre un cluster *Beowulf* y un cluster de estaciones de trabajo (COW, *cluster of workstations*) se basa en que la visión que ofrece un cluster *Beowulf* se similar a una sola máquina que muchas estaciones de trabajo. En la mayoría de los casos, los nodos de cómputo no tienen dispositivos de E/S como monitores o teclados y son accedidos solamente vía remota o por terminal serial.

El nodo maestro controla el cluster y presta servicios de sistemas de archivos a los nodos esclavos. Es también la consola del cluster y la conexión hacia el mundo exterior. Los sistemas grandes *Beowulf* pueden tener más de un nodo maestro, y otros nodos dedicados a diversas tareas específicas, como por ejemplo, consolas o estaciones de supervisión. En la mayoría de los casos los nodos esclavos de un sistema *Beowulf* son estaciones simples. Los nodos son configurados y controlados por el nodo maestro, y hacen solamente lo que éste le indique.

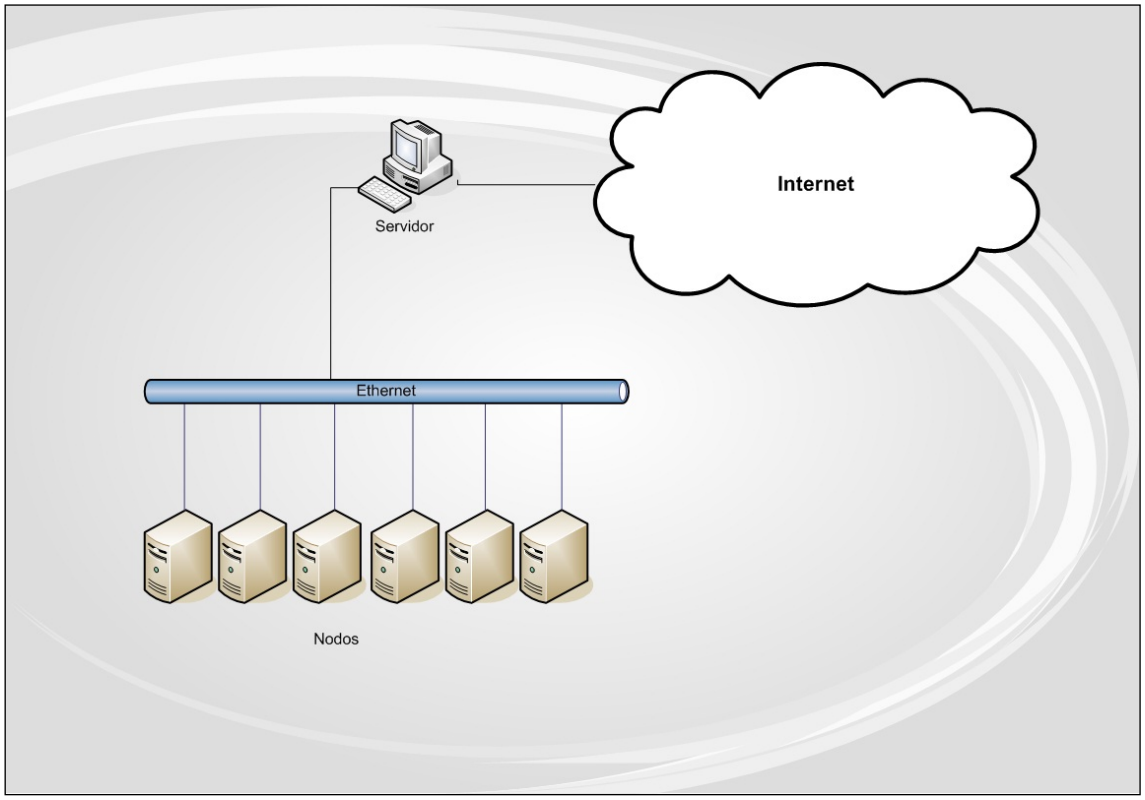

Figura 1: Esquema general de un cluster

2.2 Niveles de optimización en la E/S paralela

Los sistemas de ficheros distribuidos ofrecen un espacio global de almacenamiento que posibilita a múltiples clientes compartir los datos. En estos sistemas, cada fichero se almacena en un servidor, y el ancho de banda de acceso a un fichero se encuentra limitado por el acceso de un único servidor. Esta característica

convierte a los servidores de datos en el cuello de botella en el sistema de E/S. Este problema, conocido como la crisis de la E/S, sigue sin estar resuelto en sistemas distribuidos de propósito general, pero se han propuesto diferentes soluciones para resolver este problema, entre las que cabe destacar las siguientes:

- Utilización de paralelismo en los sistemas de E/S, con la distribución de los datos de un fichero entre diferentes dispositivos y servidores (en este apartado solo trataremos esta solución).
- Empleo de sistemas de almacenamiento de altas prestaciones (redes de almacenamiento).

La idea original de la E/S paralela viene ligada junto a la aparición del sistema RAID (Redundant Array of Inexpensive Disks) [13] a principios de los años 80 por un grupo de investigadores de UC Berkeley. Este sistema de almacenamiento hace uso de varios disco duros entre los que distribuyen o replican los datos, ofreciendo con respecto al clásico almacenamiento con un disco duro (dependiendo de la configuración realizada), mayor integridad, tolerancia a fallos, rendimiento y mayor capacidad de almacenamiento.

En la actualidad existen principalmente dos alternativas para el empleo de paralelismo en el sistema de E/S:

- Bibliotecas paralelas de E/S.
- Sistemas de ficheros paralelos.

2.2.1 Bibliotecas paralelas de E/S

La computación paralela se utiliza en diferentes tipos de aplicaciones, donde cada una de ellas refleja requisitos muy diferentes. Por este motivo, han surgido diversas bibliotecas de E/S paralela que ofrecen a los programadores de aplicaciones un conjunto de funciones de E/S altamente especializadas, y que obtienen el máximo rendimiento y flexibilidad para cada clase de aplicación. Algunas de estas bibliotecas más conocidas

son: PASSION (*Parallel And Scalable Software for I/O*), Panda, Jovian, DRA (*Disk Resident Arrays*), MIO S, VIP-FS, ChemIO o *VIPIOS*.

El principal problema que reportan las bibliotecas paralelas de E/S es que al estar concebidas fundamentalmente para el desarrollo de aplicaciones paralelas, no ofrecen una solución genérica para su uso en sistemas distribuidos. Esto es debido a que cada una de ellas proporciona un API diferente a las aplicaciones, produciendo una falta de estandarización de la E/S paralela.

2.2.1.1 MPI-IO

Dentro de las librerías de E/S paralela destaca MPI-IO. MPI-IO [18] constituye el único intento real de estandarización de la interfaz paralela de E/S. MPI-IO comenzó como un proyecto de investigación de IBM en 1994, dando soporte a múltiples operaciones de E/S paralelas y optimizaciones, tales como acceso a ficheros no contiguos, E/S colectiva, E/S asíncrona, pre-asignación de ficheros o punteros compartidos, incluyéndose en 1997 dentro del estándar MPI-2. MPI-IO debe implementarse sobre un determinado sistema de ficheros a fin de garantizar un buen rendimiento en el acceso en paralelo a los ficheros, no obstante, MPI-IO es sólo una especificación, por lo que es necesario utilizar una determinada implementación de la misma. ROMIO [19] es una implementación portable de MPI-IO, realizada en el *Argonne National Laboratory*.

El componente que permite la portabilidad de esta implementación es ADIO (*Abstract Device Interface for Parallel E/S*) [20], que ofrece una interfaz abstracta para E/S paralela. ROMIO es implementado encima de ADIO y sólo ADIO debe ser implementado de forma separada para los diferentes sistemas de ficheros subyacentes.

En la Sección MPI-IO se dan más detalles sobre la arquitectura y funcionamiento de MPI-IO.

2.2.1.2 HDF5

HDF5 (Hierarchical Data Format) es una librería de propósito general y a la vez un formato de ficheros para el almacenamiento de datos científicos. Fue creado para atender las necesidades de científicos e ingenieros trabajando en entornos de computación de altas prestaciones, con un uso intensivo de datos.

HDF5 almacena dos tipos de objetos principales: "datasets" y grupos. Un "dataset" es, esencialmente, una matriz multidimensional de datos, y un grupo es una estructura para organizar los diferentes objetos en un fichero HDF5. Usando estas dos estructuras básicas se puede crear y almacenar casi cualquier tipo de estructura de datos científica, tales como imágenes, vectores, matrices, así como retículas estructuradas y no estructuradas. También se pueden combinar estos elementos en ficheros HDF5 de acuerdo con las necesidades del caso.

Como resultado, HDF5 hace énfasis en la eficiencia del almacenamiento y de la entrada-salida de datos. HDF5-1.8.4 patch 1, es la última versión oficial disponible.

2.2.2 Sistemas de fichero paralelos

Un sistema de ficheros paralelo es aquél que elimina el problema del cuello de botella de E/S, agregando de forma lógica múltiples dispositivos de almacenamiento independientes y los nodos de E/S, mediante un sistema de almacenamiento de alto rendimiento.

En los sistemas de ficheros paralelos, el ancho de banda observado puede incrementarse mediante:

- **Direccionamiento de disco independiente**, mediante el cual el sistema de ficheros puede acceder a datos de diferentes ficheros de forma concurrente.

- **Reparto de los datos en los nodos**, mediante el cual un solo fichero se puede acceder en paralelo mediante varios clientes.

Un sistema de ficheros paralelo opera, al igual que los sistemas de ficheros distribuidos, independientemente de las aplicaciones ofreciendo una mayor flexibilidad y generalidad de las bibliotecas.

Los sistemas de ficheros paralelos más populares son:

- ***GPFS*** (*General Parallel File System*) [14] sistema de ficheros desarrollado por IBM. GPFS es particularmente apropiado en entornos donde los sistemas distribuidos no ofrecen suficiente rendimiento. GPFS ofrece un entorno que permite a los usuarios compartir el acceso a los datos a través del cluster, posibilitando la interacción a través de las interfaces estándar de UNIX.

  La fortaleza de GPFS se basa en:

  - Mejora el rendimiento del sistema: permite que múltiples nodos accedan simultáneamente a los datos utilizando llamadas estándar del sistema, balanceando la carga y por lo tanto, incrementando el ancho de banda disponible por cada nodo.
  - Asegura la consistencia de los datos: utiliza un sofisticado sistema de administración que provee la consistencia de los datos mientras permite múltiples e independientes rutas para archivos con el mismo nombre.
  - Alta recuperación y disponibilidad de los datos: crea registros o trazas independientes para cada uno de los nodos que intervienen en el sistema. Permite administrar el número de replicas que se desea manejar.
  - Alta flexibilidad del sistema: los recursos no se encuentran congelados, se puede añadir o quitar discos al sistema mientras este se encuentra montado. Cuando la demanda es muy baja se puede reconfigurar la cargar del sistema a través de todos los discos

configurados. También se puede agregar nuevos nodos sin que el sistema sea detenido y puesto en marcha nuevamente.

- Administración simplificada: los comandos de GPFS guardan la configuración en más de un archivo, conocido colectivamente como "cluster de datos". Los comandos de GPFS están diseñados para sincronizar los datos en cada uno de los nodos del sistema. De tal modo que asegura una exacta configuración de los datos.

- ***LUSTRE [15]*** es un sistema de ficheros paralelo diseñado por Cluster File Systems. En Lustre se considera a cada archivo almacenado en el sistema de archivos Lustre un objeto. Lustre presenta a todos los clientes una semántica *POSIX* estándar y acceso concurrente de lectura y escritura para los objetos compartidos. Un sistema de archivos Lustre tiene cuatro unidades funcionales. Estas son: *Meta data server* (MDS) para almacenar los metadatos; un *Object storage target* (OST) para guardar los datos reales; un *Object storage server* (OSS) para manejar los OSTs; cliente(s) para acceder y utilizar los datos. Los OSTs son dispositivos de bloques. Un MDS, OSS, y un OST pueden residir en el mismo nodo o en nodos diferentes. Lustre no administra directamente los OSTs, y delega esta responsabilidad en los OSSs para asegurar la escalabilidad para grandes clusters y supercomputadores.

  En un *Massively Parallell Processor (MPP)*, los procesadores pueden acceder al sistema de archivos Lustre redirigiendo sus peticiones I/O hacia el nodo con el servicio lanzador de tareas si está configurado como un cliente Lustre. Aunque es el método más sencillo, en general proporciona un bajo rendimiento. Una manera ligeramente más complicada de proporcionar un rendimiento global muy bueno consiste en utilizar la biblioteca *liblustre*. *Liblustre* es una biblioteca de nivel de usuario que permite a los procesadores montar y utilizar el sistema de archivos Lustre como un cliente, sorteando la redirección hacia el nodo de servicio. Utilizando *liblustre*, los procesadores pueden acceder al sistema de archivos Lustre, incluso si el nodo de servicio en el

que se lanzó el trabajo no es un cliente Lustre. *Liblustre* proporciona un mecanismo para mover datos directamente entre el espacio de aplicación y los OSSs de Lustre sin necesidad de realizar una copia de datos a través del núcleo ligero, logrando así una baja latencia, y gran ancho de banda en el acceso directo de los procesadores al sistema de archivos Lustre.

- ***PVFS*** (*Parallel Virtual File System*) [16] es un sistema de ficheros paralelo que permite a las aplicaciones, indistintamente de si estas se ejecutan en forma paralela o secuencial, almacenar y acceder a ficheros cuyos datos se encuentran distribuidos a través de un conjunto de servidores de E/S. PVFS ha sido desarrollado para *clusters* Linux, proporcionando un gran ancho de banda en operaciones de lectura y escritura concurrentes realizadas desde múltiples procesos o *threads* a un fichero común. Los datos de los ficheros se distribuyen a través de los discos de las maquinas del cluster Linux y proporcionando la apariencia de un único sistema de ficheros Unix.

  En 2001 arrancó el proyecto PVFS2, en el que se ha reescrito el código y se ha modificado el núcleo de la versión 1 de PVFS para que sea más apropiado para el entorno en el que los sistemas de ficheros paralelos son desplegados. De esta forma se ha conseguido un sistema de ficheros paralelos más adecuado a las necesidades, robusto y de alto rendimiento.

  Las características que posee la versión 2 de PVFS son:

  - Distribución de datos flexible. PVFS1 utiliza el patrón de distribución *Round-Robin* para realizar la distribución de datos de los ficheros entre los servidores. PVFS2 no solo permite el uso del patrón de distribución *Round-Robin* sino que incluye un sistema modular para añadir nuevos patrones de distribución en el sistema y el uso de ello para los nuevos ficheros.

- Información de metadatos distribuida. PVFS1 contemplaba un único servidor de metadatos, pero se convierte en un único punto de fallos, puesto que los datos no se encuentran replicados, y es un cuello de botella. En PVFS2 los metadatos se encuentran distribuidos en un conjunto de servidores, que pueden coincidir con los servidores de datos.
- Proporciona múltiples interfaces, incluso un interfaz de MPI-IO vía ROMIO y otra para acceder al sistema de ficheros tradicional mediante el uso de un módulo del kernel de Linux.
- Utiliza servidores y clientes sin estado, lo cual, facilita la recuperación de fallos en el sistema.
- Soporte de concurrencia explícito. En PVFS2 los hilos o procesos ligeros son usados cuando son necesarios para proporcionar acceso no bloqueante a todos los dispositivos y evitar la serialización de las operaciones independientes.
- Soporte de redundancia de datos y metadatos. PVFS1 no soporta redundancia de datos. En PVFS2 utiliza el enfoque RAID para proporcionar tolerancia de fallos de disco, pero si un servidor desaparece, los datos que contenía son inaccesibles hasta que el servidor se recupera.
- Soporta clusters heterogéneos.
- Presenta un diseño modular.

En cuanto a los sistemas de ficheros paralelos, su principal problema es que están especialmente pensados para máquinas paralelas y no se integran adecuadamente en entornos distribuidos de propósito general. Además, cada sistema de ficheros paralelo utiliza una estructura de fichero paralelo diferente, incompatible con la de otros sistemas.

2.3 MPI

La Interfaz de Paso de Mensajes (*Message Passing Interface*), es una biblioteca considerada en la actualidad un estándar, el cual, especifica el protocolo de comunicación entre procesos que trabajan en un programa paralelo y forman parte de un sistema paralelo mediante el paso de mensajes.

Las características de MPI son:

- Estandarización.
- Rendimiento: optimización de las librerías para cada uno de los sistemas en los que puede ser ejecutado.
- Escalabilidad: gestión de los distintos tipos de tamaño y formato de los mensajes de forma totalmente transparente.
- Portabilidad: existencia de multitud de librerías o implementaciones como MPICH o OPEN-MPI, disponible para distintos lenguajes de programación como C, C++ o Fortran, por poner algunos ejemplos.

En la actualidad existen dos versiones principales del estándar: la versión 1.2 (llamada MPI-1), la cual aporta el paso de mensajes y un entorno de ejecución estático y la versión 2.1 (llamada MPI-2), la cual además incluye operaciones de E/S paralela, gestión dinámica de procesos y operaciones de memoria remota (RMA) [22].

La comunicación mediante paso de mensajes puede ser punto a punto o multipunto, para las cuales MPI proporciona una serie de funciones. La comunicación punto a punto, es la establecida entre dos nodos, pudiendo ser síncrona (el remitente se bloquea hasta que el destinatario empieza a recibir el mensaje) o asíncrona (las llamadas no son bloqueantes y realizan peticiones de envió), mientras que la comunicación multipunto es en la que interviene varios nodos a la vez.

2.3.1 Fundamentos de MPI

Con MPI el número de procesos necesarios se asigna antes de la ejecución del programa, y no se crean procesos adicionales mientras la aplicación se ejecuta. A cada proceso se le asigna una variable que se denomina *rank*, la cual identifica a cada proceso, en el rango de 0 a p-1 donde p es el número total de procesos, realizándose el control de ejecución del programa mediante el uso de esta variable, ya que posibilita controlar que proceso ejecuta una determinada sección de código. Además en MPI se define un comunicador como una colección de procesos, los cuales pueden enviar mensajes entre ellos y permite abstraerse de la topología a nivel físico de las computadoras, el comunicador básico se denomina MPI_COMM_WORLD el cual agrupa a todos los procesos activos durante la ejecución de una aplicación.

Las llamadas de MPI se pueden clasificar en:

- Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones: permiten inicializar la biblioteca de paso de mensajes, identificar el número de procesos (*size*) y el rango de los procesos (*rank*), donde las funciones principales son:
    - *MPI_Init*: primera llamada para cada proceso que establece un entorno.
    - *MPI_Comm_size*: devuelve el número de procesos del comunicador.
    - *MPI_Comm_rank*: devuelve el identificador del proceso dentro del comunicador.
    - *MPI_Finalize*: Termina la ejecución en MPI y libera los recursos utilizados.
- Llamadas utilizadas para transferir datos entre un par de procesos: operaciones de comunicación punto a punto, para envío y recepción, consiguiéndose mediante las llamadas:
    - *MPI_Send*: envía información de un proceso a otro.
    - *MPI_Recv*: recibe información desde otro proceso.

Existe también la versión asíncrona o no bloqueante de estas llamadas las cuales son MPI_Isend y MPI_Irecv.

- Llamadas para transferir datos entre varios procesos: llamadas de comunicación grupales, donde alguna de estas llamadas son:
    - MPI_Barrier: sincronización global entre los procesadores del comunicador.
    - MPI_Bcast: envío de datos desde un proceso (*root*) a todos los demás.
    - MPI_Gather: recolección de datos de todos los procesos en uno de ellos.
    - MPI_Scatter: distribución de datos de un proceso entre todos los demás.
    - MPI_Reduce: reducción de los datos de cada procesador, dejando el resultado en uno de ellos (*root*).

2.3.2 Tipos de datos en MPI

Los tipos de datos MPI son patrones de acceso de datos en memoria o en fichero. Pueden expresar patrones regulares o irregulares, con o sin huecos entre los datos.

Los tipos de datos básicos son los usados en los lenguajes de programación tradicionales como C: *char* (MPI_CHAR), *byte* (MPI_BYTE), *integer* (MPI_INT), *float* (MPI_FLOAT), etc. Los tipos de datos complejos son construidos a partir de tipos de datos básicos o recurrentemente de otros tipos de datos complejos. Como tipos de datos complejos podemos tener vectores y tipos estructurados.

Los tipos de datos complejos pueden ser representados como un árbol: los nodos internos son otros tipos de datos complejos usados en el proceso de construcción y las hojas son tipos de datos básicos. Los tipos de datos MPI son objetos opacos, p.ej. su estructura interna es directamente accesible al usuario. Una vez construido, el usuario

puede tener acceso a los tipos de datos por medio de un manejador. MPI ofrece un mecanismo para descifrar la composición de los tipos de datos (MPI_Type_get_envelope).

2.3.3 MPI-IO

MPI-IO se desarrolló en 1994 en el laboratorio Watson de IBM con el fin de proporcionar paralelismo a la E/S en aplicaciones desarrolladas en MPI. La Figura 2 muestra la arquitectura de ROMIO[19], la implementación más extendida de MPI-IO.

Las características más importantes de MPI-IO son:

- Portabilidad
  - Proporciona una interfaz estándar para las E/S, algo que no existía.
  - Interfaces UNIX tradicionales no estaban bien adaptados al paralelismo de E/S.
  - Plataforma/proveedor específico paralelo para E/S.
- Facilidad de uso
  - Escribir es como enviar y leer es como recibir.
  - Versatilidad de datos de MPI.
- Eficiencia/rendimiento potencial
  - Operaciones colectivas para permitir una mejora del potencial y del rendimiento.
  - Proporciona "hints" para posibles optimizaciones de E/S.
- Archivo de interoperabilidad
  - Proporciona un archivo estándar para la representación de datos.
  - Definición de un tratado universal de representación de datos externos.
- Apoyo a las representaciones definidas por los usuarios (tipos de datos, vistas, etc.).

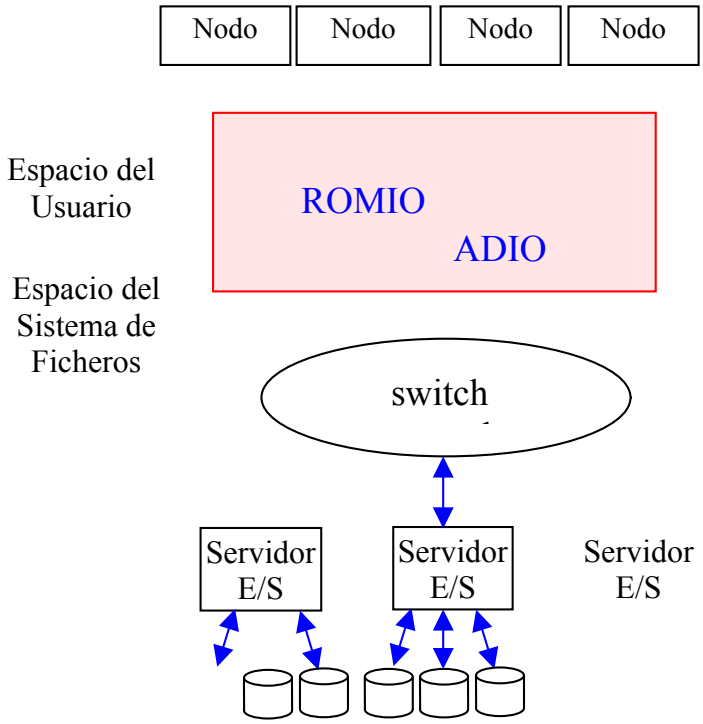

Figura 2: Arquitectura MPI-IO

2.3.3.1 Modelo de fichero en MPI-IO

Un fichero de MPI-IO es una colección ordenada de datos escritos en una máquina. Un fichero es abierto por un conjunto de procesos representados por un comunicador, donde todas las invocaciones de E/S colectiva o no, acceden al fichero mediante este comunicador. El desplazamiento en un fichero, es una posición absoluta medida en bytes en relación al principio del fichero, definiendo este el principio de la vista.

Un *etype* (tipo de datos elemental) es la unidad de acceso de datos (lectura y escritura) y desplazamiento (expresado como cuenta de *etypes*) pudiendo ser cualquier tipo de MPI predefinido. Un *filetype* es la base para dividir un archivo entre procesos y

define una plantilla para tener acceso al fichero. Un *filetype* está compuesto por un *etype* o por un tipo de dato compuesto por varios *etypes*.

Una vista se define como el acceso actual de los datos visibles y accesibles de un fichero abierto como un conjunto ordenado de *etypes*. Cada proceso tiene su propia vista sobre el fichero, definida por tres valores: un desplazamiento, un *etype*, y un *filetype*. El patrón descrito por un *filetype* es repetitivo y su comienzo está definido por el valor del desplazamiento. La vista por defecto es una cadena de bytes (el desplazamiento es el cero, *etype* y *filetype* son igual a MPI_BYTE), p. ej. la vista traza una correspondencia de uno a uno al fichero. Un grupo de procesos puede usar vistas complementarias para alcanzar una distribución de datos global como un modelo dispersión/recolección. Un desplazamiento es una posición dentro de la vista, expresado como un número de *etypes*. Los huecos en el *filetype* de la vista son calculados mediante esta posición. El desplazamiento 0 es la posición del primer *etype* visible de la vista (después de saltar el desplazamiento y cualquier hueco inicial en la vista). Todas las operaciones sobre un fichero abierto usan el descriptor de fichero como una referencia al fichero.

2.3.3.2 Acceso a los datos en MPI-IO

En MPI-IO los datos se transmiten entre ficheros y procesos mediante llamadas de escritura y lectura. Las rutinas de acceso de datos pueden ser individuales o colectivas.

En la implementación de una rutina colectiva, a diferencia de las rutinas individuales, los procesos se coordinan entre sí para optimizar el acceso al dispositivo de E/S, esto se realiza mediante la combinación de pequeñas peticiones individuales en los nodos de cómputo en peticiones más grandes, optimizando así el funcionamiento de disco y red (reduciendo el número de transferencias necesarias).

Dependiendo del lugar donde se realiza la combinación, uno puede identificar dos métodos de E/S colectivas. Si la combinación ocurre en nodos intermedios o en los nodos de cómputo los llamaremos métodos E/S de dos fases.

2.4 Arquitectura de ROMIO

La implementación más extendida del estándar de MPI-IO es ROMIO el cual forma parte de distribuciones como MPICH [21], LAM, CV-MPI, NEC-MPI, y SGI-MPI. ROMIO, el interfaz de MPI-IO, es implementado sobre una interfaz abstracta llamada ADIO, siendo este el sistema de ficheros independiente. Sobre la cima de ADIO y bajo el interfaz de MPI-IO, ROMIO pone en práctica mecanismos como vistas, y varias optimizaciones de acceso como la E/S colectiva de dos fases (Two-phase I/O) [23].

El modelo de fichero de MPI-IO y las operaciones de ficheros están basados en los tipos de datos de MPI, donde una de las operaciones de fichero más interesantes es la declaración de una vista. Una vista ofrece varias ventajas: los datos no contiguamente almacenados son "vistos" contiguamente, facilitando la tarea del programador y permitiendo optimizaciones de E/S no contiguas. Al mismo tiempo una vista es una "pista" sobre el futuro modelo de acceso, pudiendo ser usado para optimizar el acceso.

La interfaz de ADIO contiene las típicas funciones para manejar ficheros: open, close, fcntl, read, write, etc. En la implementación actual de ADIO el mecanismo de vistas reside en la capa de MPI-IO, como se muestra en la Figura 3. La vista mapea sobre un espacio lineal del fichero mediante MPI-IO y, a su vez, el fichero sobre subficheros o discos por el sistema de ficheros. Los dos mapeos se realizan de forma explícita, incluso cuando la vista mapea datos contiguamente posicionados sobre un subfichero/disco. ROMIO contiene una implementación de la técnica two-phase I/O, descrita en este mismo documento. La fase de mezcla es implementada en la capa de MPI-IO. La fase de E/S transfiere el buffer colectivo del nodo de cómputo al sistema de

ficheros. La fase de E/S es implementada en una función de ADIO, que a su vez llama funciones de acceso de sistema de ficheros.

La técnica *two-phase I/O* de ROMIO puede necesitar hasta tres transferencias de red para superponer una región existente. La primera corresponde a la fase de intercambio, la segunda la transacción "leído modifica escriben" necesaria para actualizar el fichero. La parte leída puede ser optimizada cuando la región del fichero de huecos es superpuesta.

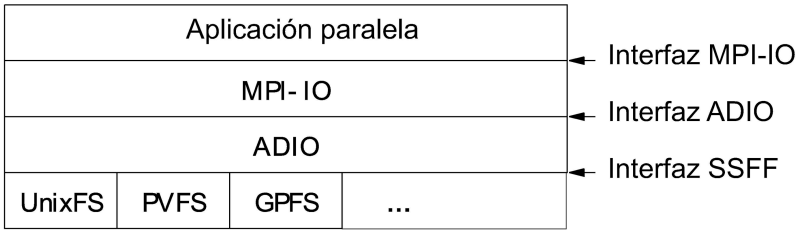

Figura 3: Arquitectura ROMIO

2.4.1 Técnica *two-phase I/O*

El uso de la técnica *two-phase I/O* es una de las técnicas usadas a la hora de distribuir las fracciones de datos que deberán ser escritas en cada uno de los procesos que quieran hacer uso del fichero distribuido. Algunos de los sistemas distribuidos que pueden hacer uso de esta técnica son AFS, xFS, GPFS, o PVFS, por poner algunos ejemplos. Además, dichos sistemas pueden usar interfaces *POSIX* o MPI-IO.

La técnica *two-phase I/O* para la escritura/lectura colectiva de datos no contiguos, debe su nombre a que se realiza en dos fases: la primera fase consiste en el intercambio de datos entre los procesos que intervienen en la operación. La segunda fase es la escritura o lectura de los datos en disco.

En primer lugar, lo que hace la técnica *two-phase I/O*, es asignar qué parte del fichero distribuido corresponderá a cada proceso que quiera tener acceso a dicho

fichero. La técnica *two-phase I/O* asignará cada parte del fichero distribuido a cada proceso de acuerdo a su posición, es decir, la primera parte será para el primer proceso, la segunda parte, para el segundo proceso… y así sucesivamente. Ejemplo. Se dará la siguiente distribución para cuatro procesos accediendo a un fichero distribuido, usando la técnica *two-phase I/O*:

n_m : Dato situado en la posición m del fichero distribuido, asignado al proceso n.

Cada proceso dispondrá de los siguientes datos:

Proceso 0: 0 0 0 6 0 10 0 13 0 14

Proceso 1: 1 4 1 8 1 9 1 15

Proceso 2: 2 1 2 2 2 3 2 12

Proceso 3: 3 5 3 7 3 11

Según el color, cada parte del fichero se asigna a un proceso:

al 0, al 1, al 2, al 3.

Así pues, cuando se llegue a la fase de intercambio de datos, para construir el fichero distribuido, se realizarán las siguientes comunicaciones:

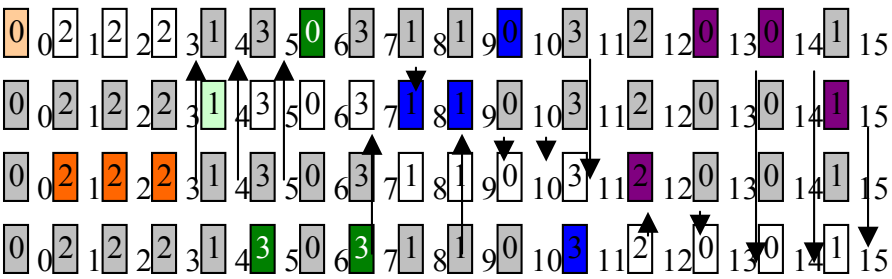

Figura 4: Ejemplo de intercambio de datos con two-phase I/O

En la primera de las fases, el intercambio de datos, todos los procesos mandarán los datos que se encuentren en partes del fichero distribuido asignadas a otros procesos, al proceso al que pertenecen. Es en este momento en el que realizan los intercambios de datos por la red, y en el que interviene la técnica para distribuir los datos correspondientes entre los distintos procesos. Así pues, tras esta fase, todos los procesos deberían tener un buffer con todos los datos que necesitan en su parte del fichero distribuido. En la segunda de las fases, la escritura de datos, todos los procesos escribirán su parte de datos del fichero distribuido en disco.

3 FUSE: FILESYSTEM IN USER SPACE

FUSE es la abreviación de las siglas de *Filesystem in USErspace.* Este sistema de ficheros es de libre distribución y se rige por los términos de *GNU General Public License* y de *GNU Lesser General Public License.* Actualmente, se encuentra disponible para los sistemas operativos: Linux, FreeBSD [29], NetBSD [30], OpenSolaris [31] y Mac OS X [32]. FUSE fue oficialmente incluido como línea principal del *kernel Linux* en la versión 2.6.14. FUSE fue originalmente desarrollado para soportar AVFS [28], hasta que se fue convirtiendo en un proyecto totalmente separado. Actualmente es usado en un gran número de proyectos relacionados con la E/S.

Como es sabido, el *kernel* de *Linux* posee un componente muy importante que es VFS [26], encargado de administrar todos los sistemas de archivos montados que tengamos. Por ejemplo, ante la ejecución del siguiente comando:

```
mount -t ext3 /dev/hdb2 /mnt/unidad
```

VFS se encarga de gestionar los sistemas de ficheros locales que se montan en una unidad. Gracias a VFS se presenta al usuario la nueva estructura del sistema de archivos. Para llevar a cabo estas acciones, cada acceso que se hace sobre */mnt/unidad* se redirige al módulo controlador del sistema de ficheros local, por ejemplo *ext3* [27].

FUSE funciona de manera similar a otros sistemas de ficheros. Hay un módulo en el *kernel* que se registra como un manejador de un sistema de archivos virtual. Entonces cuando queremos acceder a una unidad de FUSE montada, VFS le redirige la llamada al módulo de FUSE, y FUSE redirige la llamada al proceso controlador. Resumiendo, el sistema FUSE consiste en una capa entre el *kernel* y nuestro programa en el espacio de usuario. Todo esto nos permite tener nuestro propio sistema de archivos corriendo en espacio de usuario, además la interfaz (o API) de FUSE es mucho más sencilla que la del *kernel*. Podríamos tener por ejemplo un programa que utilice FUSE para mostrar el contenido de un archivo TAR como un directorio más. Por ejemplo, GmailFS es un sistema que permite usar nuestra cuenta en Gmail como un disco rígido más, y está implementado usando FUSE. También existe WikipediaFS y otros proyectos muy interesantes.

3.1 Introducción

Con FUSE es posible implementar un sistema de ficheros funcionalmente completo en el espacio de usuario de una aplicación. Las características principales del sistema FUSE son:

- Una sencilla librería API
- Instalación sencilla (no es necesario parchear o recompilar el Kernel)

- Implementación segura
- Un espacio de usuario -la interfaz del *Kernel* es muy eficiente
- Permite ser usado por usuarios sin muchos privilegios
- Funciona sobre *Kernels* de *Linux* 2.4.X y 2.6.X
- Ha demostrado ser muy estable a través del tiempo

Implementar un sistema de ficheros con FUSE es sencillo, un sistema de ficheros "hola mundo" no es más largo de 100 líneas de código. Seguidamente se muestra una sesión de ejemplo:

```
~/fuse/example$ mkdir /tmp/fuse
~/fuse/example$ ./hello /tmp/fuse
~/fuse/example$ ls -l /tmp/fuse
total 0
-r--r--r--  1 root root 13 Jan  1  1970 hello
~/fuse/example$ cat /tmp/fuse/hello
Hello World!
~/fuse/example$ fusermount -u /tmp/fuse
```

3.1 Puesta en marcha

Después de la instalación, puede realizar una verificación inicial gracias al sistema de ficheros de ejemplo existente en el directorio "*example*". Para poder ver lo que

ocurre, es posible depurar el funcionamiento con la opción "-d". A continuación se muestra la salida producida tras la ejecución de "*/tmp/fuse/hello*" dentro de otra consola:

```
~/fuse/example> ./hello /tmp/fuse -d
unique: 2, opcode: LOOKUP (1), ino: 1, insize: 26
LOOKUP /hello
        INO: 2
        unique: 2, error: 0 (Success), outsize: 72
unique: 3, opcode: OPEN (14), ino: 2, insize: 24
        unique: 3, error: 0 (Success), outsize: 8
unique: 4, opcode: READ (15), ino: 2, insize: 32
READ 4096 bytes from 0
        READ 4096 bytes
        unique: 4, error: 0 (Success), outsize: 4104
 unique: 0, opcode: RELEASE (18), ino: 2, insize: 24
```

Es posible intentar más operaciones mediante el uso del sistema de ficheros de ejemplo implementado en "*fusexmp*". Esto es similar al directorio raíz de "*mount –bind//mountpoint*". Esta aplicación no es de mucho uso en sí mismo, sin embargo, la importancia radica en que puede ser usado como una plantilla para la creación de un nuevos sistemas de ficheros.

Por defecto el sistema de ficheros FUSE es ejecutado con multi-hilos. Esto puede ser comprobado ejecutando el punto de montaje recursivamente en el sistema de ficheros "*fusexmp*". La capacidad de multi-hilo puede ser deshabitada añadiendo la opción –s.

Algunas opciones pueden ser pasadas al módulo del *Kernel* FUSE y a la librería. Para ello revise la salida que produce el comando "*fusexmp –h*" con la lista de dichas opciones.

3.2 Funcionamiento

La siguiente figura muestra el camino de la llamada a un sistema de ficheros desde unaaplicación (por ejemplo "*stat*") dentro del anterior ejemplo "hola mundo":

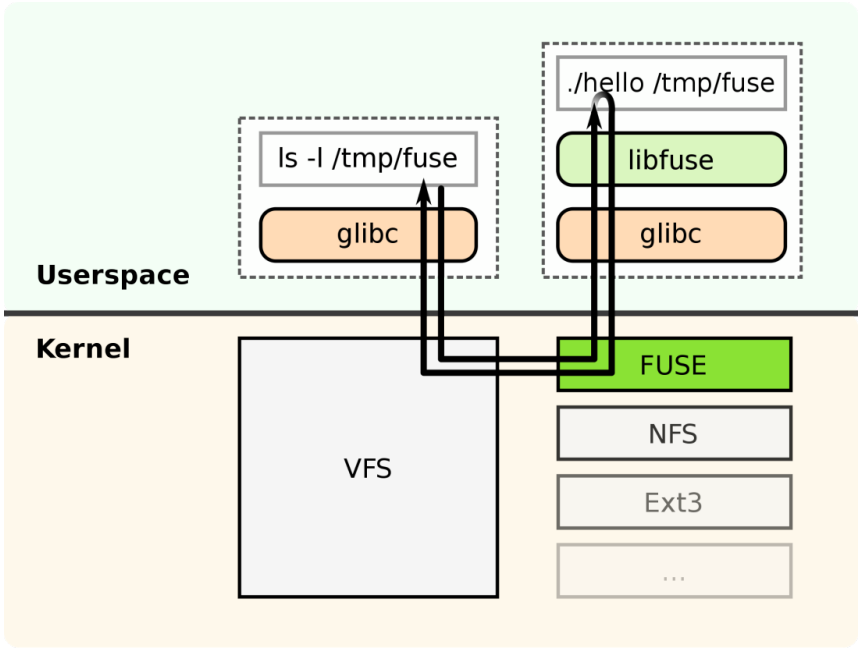

Figura 5: Arquitectura de FUSE

El módulo del *Kernel* de FUSE y la librería de FUSE se comunican vía un descriptor de fichero especial, el cual es obtenido al abrir "*/dev/fuse*". Este fichero puede ser abierto múltiples veces, y el descriptor de fichero obtenido es pasado a la llamada de sistema del montaje "*mount*", para hacer coincidir el descriptor con el sistema de ficheros montado.

La API que presenta FUSE es relativamente sencilla. Simplemente para realizar un sistema de ficheros en espacio de usuario hay que incluir el archivo de encabezado fuse.h y enlazar con la librería *libfuse*. Será suficiente declarar una estructura llamada *fuse_operations* que simplemente contiene varios punteros a funciones que serán llamadas para cada operación. Por último se terminará el programa con la función *fuse_main*. Por lo general, las funciones deben devolver 0 (cero) en caso de éxito y un número negativo indicando el error en caso de fallo. La excepción a esto son las llamadas *write* y *read* que deben devolver un número positivo representando la cantidad de bytes leídos, cero en caso de EOF o un número negativo en caso de error. Las verdaderas llamadas al sistema en *linux* devuelven -1 y establecen "errno" al valor adecuado. Lo que se debe hacer es devolver un número negativo indicando el error, y FUSE se encargará de devolver al proceso llamante -1 y rellenar *errno* con el valor absoluto de nuestra función. Es decir, si se devuelve -ENOENT, FUSE se va a encargar de devolver -1 y rellenar *errno* con ENOENT. Los miembros básicos de la estructura fuse_operations son:

- *int (*getattr) (const char *, struct stat *);*
- *int (*open) (const char *, struct fuse_file_info *);*
- *int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);*
- *int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *);*

  - **getattr:** Esta función es llamada cuando se quieren obtener los atributos de un archivo, de hecho, cuando se le hace un "*stat*" a un archivo, se llama a esta función. El primer parámetro es de entrada e indica el *path* del archivo. El segundo parámetro es de salida y es la estructura *stat* a rellenar. Una devolución de cero indica éxito.
  - **open:** Se llama cuando se quiere abrir un archivo, el primer parámetro es el *path* del archivo, el segundo parámetro es una estructura que contiene información acerca de los *flags* de apertura, y además nos permite devolver un *handler*. Como esta estructura será pasada en las funciones *read*, *write* y alguna otra, es posible utilizar el *handler* que se le pasa para saber de qué

archivo se habla, en todo caso no es un parámetro necesario para un sistema de archivos sencillo y se podría ignorar. Lo único que hay que hacer es comprobar si se puede abrir el archivo, en caso afirmativo devolvemos cero.

- **read:** Se llama cuando se quiere leer un archivo. El primer parámetro es el *path* del archivo, el segundo parámetro es el *buffer* donde almacenar los datos, el tercer parámetro es la cantidad de bytes a leer, el cuarto el desplazamiento y el quinto es el mismo que el de *open*. Se devuelve la cantidad de bytes leídos.
- **readdir:** Se utiliza para leer un directorio. El primer parámetro es el *path* del directorio, el segundo una estructura que hay que rellenar, el tercero es una función que se usa para rellenar la estructura del segundo parámetro y los otros dos se pueden ignorar.

También tenemos los siguientes miembros que quizás ya son algo más interesantes:

- *int (*unlink) (const char *);*
- *int (*rename) (const char *, const char *);*
- *int (*truncate) (const char *, off_t);*
- *int (*write) (const char *, const char *, size_t, off_t, struct fuse_file_info *);*
- *int (*mknod) (const char *, mode_t, dev_t);*

- **unlink:** Borra un archivo, el único parámetro es el *path*.
- **rename:** Renombra un archivo, el primer parámetro es el nombre viejo y el segundo el nuevo nombre.
- **truncate:** Trunca un archivo, le cambia el tamaño, ya sea a un tamaño menor o mayor. El primer parámetro es el *path* y el segundo es el nuevo tamaño que será aplicado.
- **write:** Escribe un archivo, el primer parámetro es el path. El segundo el *buffer* que contiene los datos, el tercero la cantidad de bytes a escribir, el cuarto el

offset en el cual escribir y el último es la estructura con información del archivo (lo mismo que se le pasa a *open* y a *read*).

- **mknod:** Crea un nuevo nodo o archivo. No solamente sirve para crear archivos, también sirve para crear *FIFOs*, archivos de dispositivos y demás cosas que se pueden hacer con una llamada *mknod* común y corriente. El primer parámetro es el *path* a crear, el segundo representa los permisos y el tipo de archivo, el tercero contiene el número mayor y menor en caso de que estemos creando un dispositivo.

Hay un punto a tener en cuenta, el *path* que pasa FUSE siempre es un *path* "absoluto" pero referenciado a nuestro sistema de archivos. Supongamos que tenemos nuestro sistema de archivos montado en */mnt/fuse* y tenemos un archivo llamado fuse.txt, si alguien ejecuta la siguiente llamada al sistema:

```
int fd = open ( "/tmp/fuse/fuse.txt", O_RDONLY );
```

FUSE efectuará una llamada a *open* estableciendo como *path* la cadena "/fuse.txt". Aún quedan más miembros, es recomendable revisar el archivo de cabecera fuse.h ya que el resto de los miembros se encuentran muy bien documentados.

4 SISTEMA DE E/S PARALELA AHPIOS

AHPIOS (*Ad-hoc Parallel I/O System*), es el primer sistema de E/S paralela escalable completamente implementado en MPI (*Message Passing Interface*). En la actualidad la mayoría de los supercomputadores pertenecientes al Top500 [42] utilizan sistemas de ficheros como PVFS, GPFS o Lustre, son usados mayoritariamente en aplicaciones de procesamiento de datos y visualización, pudiéndose beneficiar estas aplicaciones del sistema AHPIOS al ofrecerlas una solución transparente de E/S paralela, fácilmente instalable.

AHPIOS puede ser usado como un middleware alojado entre MPI-IO y los recursos de almacenamiento distribuido, proporcionando un alto rendimiento de acceso a los ficheros. AHPIOS puede ser usado como una alternativa ligera y barata a cualquier sistema de ficheros paralelo.

Las principales características de AHPIOS son las siguientes:

- **Alto rendimiento**: se logra a través de la estrecha integración de MPI-IO y el sistema de almacenamiento, que permite un acceso eficiente a los datos, a través de dos niveles de caché cooperativa y una estrategia de adquisición de datos asíncrona.
- **Escalabilidad**: el sistema es escalable con ambas jerarquías de memorias y con el almacenamiento. El primer nivel de caché es ajustable al número de procesos que participan en la aplicación paralela. El segundo nivel de caché y el sistema de almacenamiento global (disco) se adapta al número de servidores AHPIOS y recursos de almacenamiento.
- **Portabilidad**: logrado gracias a la completa implementación en MPI (siendo esta la primera implementación conocida de un sistema paralelo de E/S en MPI).
- **Simplicidad**: es fácil de usar y no es necesaria la modificación de las aplicaciones existentes. El funcionamiento de AHPIOS puede ser configurada por el usuario a partir de un fichero de texto plano.

4.1 Visión general

Dada una aplicación en MPI que acede a distintos ficheros a través de la interfaz de MPI-IO, y un conjunto de recursos de almacenamiento distribuidos, AHPIOS construye bajo demanda una partición distribuida, la cual puede ser accedida de forma transparente y eficiente. En cada partición AHPIOS, los usuarios pueden crear un espacio de nombres de directorio, de la misma manera que en cualquier sistema de ficheros. Los ficheros almacenados en una partición AHPIOS son listados de forma transparente sobre los recursos de almacenamiento al igual que en cualquier sistema de ficheros paralelo. Estas particiones son gestionadas por un conjunto de servidores de

almacenamiento, los cuales, se ejecutan como una aplicación MPI independiente. El acceso a una partición AHPIOS es realizado a través de la interfaz de MPI-IO. Una partición puede ser construida escalando hacia arriba o hacia abajo, bajo demanda mientras la aplicación se esté ejecutando.

El sistema maneja una jerarquía de memorias caché cooperativas, como se muestra en la Figura 6:

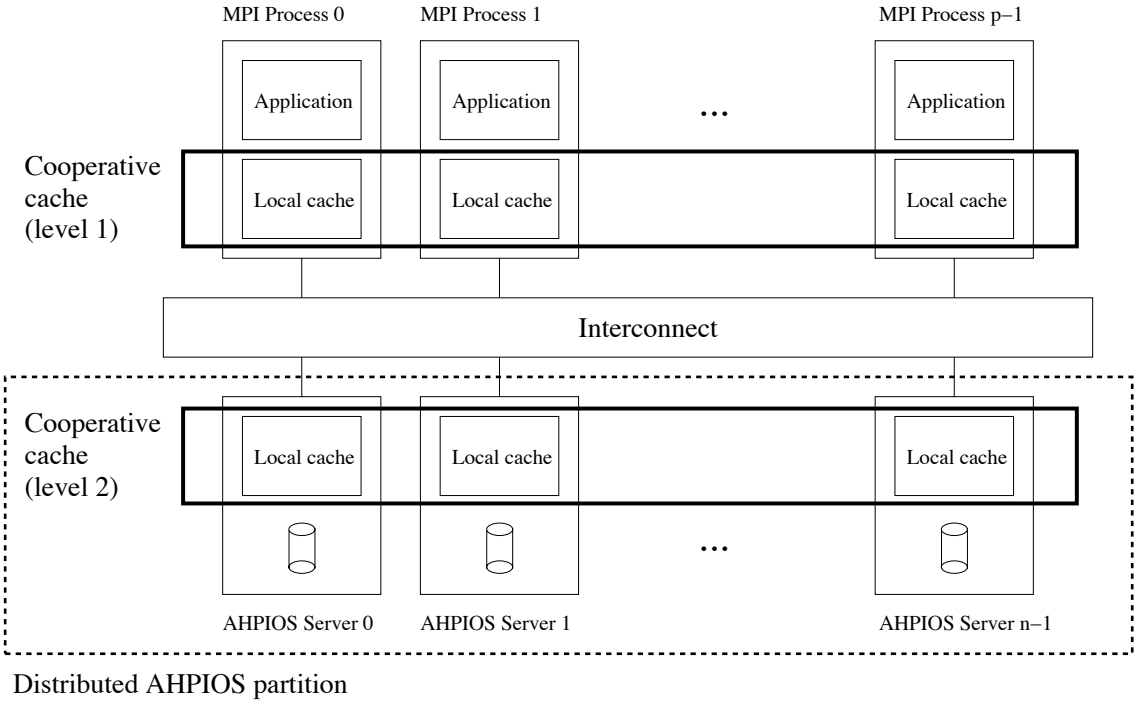

Figura 6: Esquema AHPIOS

En primer lugar las cachés de las aplicaciones clientes se agrupan en un buffer colectivo, el cual será usado por la capa de MPI-IO para reordenar y recopilar las peticiones de E/S con el fin de mejorar el rendimiento de acceso para la E/S. En segundo lugar los servidores, AHPIOS gestiona una copia de caché cooperativa. La comunicación interna y entre estas capas se realiza a través de operaciones estándar de MPI.

Una estrategia de adquisición de datos, oculta la latencia de transferencia de bloques de datos entre los niveles de la jerarquía de caches. La transferencia de datos entre la cache del primer nivel (clientes) y la cache del segundo nivel (servidores), así como entre la caché de los servidores y el almacenamiento final, se realizan de forma asíncrona.

Los mecanismos de MPI-IO y las optimizaciones como las *vistas* y el grupo de E/S están fuertemente integrados dentro del sistema de almacenamiento. Las *vistas* pueden ser aplicadas, ya sea en cliente o en el servidor y el grupo de buffers debe residir en el cliente o cerca del almacenamiento como en los nodos de los servidores.

Para cada ejecución, el usuario puede definir la configuración de los parámetros de manera centralizada, como pueden ser el número de recursos de almacenamiento, el tamaño del buffer de red y del tamaño bloque, el tamaño de la caché del cliente y del servidor, el tipo de política de planificación, etc. El usuario tiene el control de la configuración de MPI-IO y del sistema paralelo E/S desde un único fichero.

El sistema puede consistir en varias particiones independientes, donde cada una es manejada por diferentes grupos de servidores, con distintas configuraciones.

4.2 Preámbulo

El diseño y la implementación de AHPIOS están basados en la arquitectura *software* de ROMIO. En ROMIO, la interfaz de MPI-IO es implementada encima de la interfaz abstracta de dispositivos ADIO. En la Figura 7 se muestra la arquitectura *software* de ROMIO.

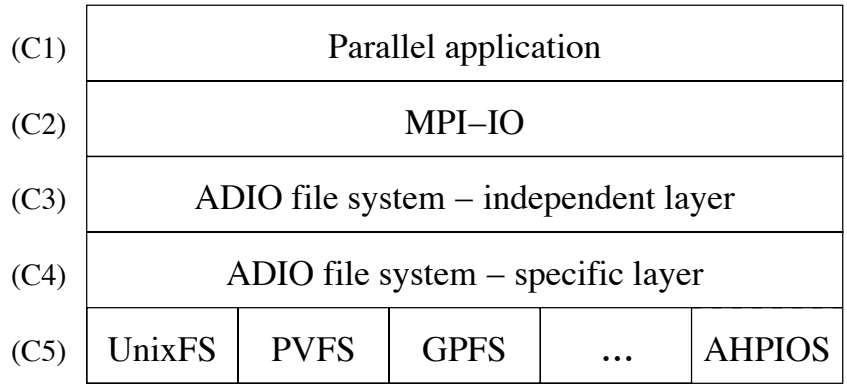

Figura 7: Arquitectura ROMIO

Como se puede observar esta divida en cinco capas. Las llamadas MPI-IO de la aplicación son traducidas en la capa (C2), en un subconjunto menor de llamadas ADIO. La capa (C3), contiene la implementación de mecanismos y optimizaciones como las vistas, acceso a ficheros no continuos y llamadas colectivas de E/S. La capa (C4) mapea a un subconjunto menor de funciones de acceso a ficheros sobre un sistema de ficheros particular. Esta capa ha tenido que ser implementada con el fin de añadir soporte ROMIO, a los nuevos sistemas de ficheros paralelos. Por último la capa (C5) consiste en las rutinas de acceso del sistema de ficheros (pudiendo ser librerías a nivel de usuario o el sistema de ficheros montado de manera local). AHPIOS ha sido integrado dentro de la pila de la arquitectura de ROMIO, mediante la implementación de las capas (C4) y (C5).

4.3 Diseño e implementación

El acceso de las aplicaciones MPI a una partición AHPIOS se hace a través de la interfaz de MPI-IO (llamándose a la implementación de la interfaz MPI-IO cliente AHPIOS). Una partición AHPIOS es gestionada por un conjunto de servidores AHPIOS, los cuales son procesos MPI. Estos procesos se ejecutan independientemente de la aplicación MPI cliente.

En la Figura 8 se muestra la arquitectura *software* del sistema AHPIOS, donde en la parte superior se encuentra la aplicación cliente y en la parte inferior se encuentra un conjunto de servidores AHPIOS.

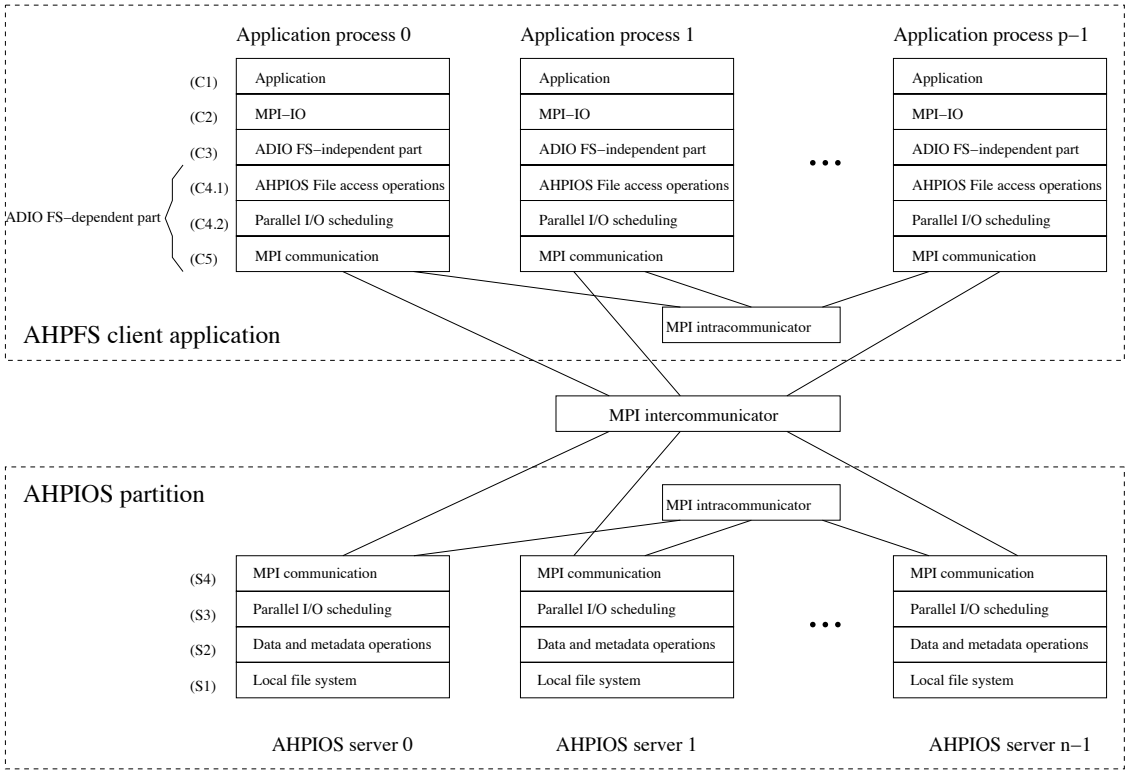

Figura 8: Arquitectura *software* de AHPIOS

Los servidores AHPIOS están comunicados entre sí a través de un intercomunicador de MPI (*MPI intracommunicator*). Un *MPI intracommunicator* es un mecanismo de MPI el cual permite a los miembros de un grupo de procesos comunicarse con otro grupo a partir de las rutinas de comunicación de MPI.

Los servidores AHPIOS están comunicados con la parte cliente AHPIOS a través de un intercomunicador de MPI.

El cliente AHPIOS forma parte de la interfaz de MPI-IO. Está integrado dentro de la pila de la arquitectura ROMIO en las capas (C4) y (C5). La capa (C4) puede ser dividida en dos subcapas (C4.1) y (C4.2). La subcapa (C4.1) mapea las operaciones de

fichero de ADIO sobre tareas que puedan ser realizadas por los servidores AHPIOS de manera individual. Estas tareas pueden ser relaciones de metadatos, como la creación o eliminación de un fichero u operaciones sobre ficheros. En la subcapa (C4.2) estas tareas son programadas para la transferencia por un modulo planificador de E/S paralela. La capa (C5) es la responsable de las comunicaciones con los servidores AHPIOS a través de las rutinas de comunicación de MPI.

Los servidores AHPIOS han sido diseñados como un programa MPI completo e independiente. Como se puede ver en la Figura 8, el diseño del servidor está estructurado en 4 subcapas. La comunicación con la aplicación cliente es realizada a través de rutinas MPI en la subcapa (S4). La subcapa (S3) es la responsable de la política de planificación de E/S paralela, orientada a la cooperación con los correspondientes módulos del lado del cliente. La gestión de los datos y metadatos se realiza en subcapa (S2). Finalmente, la subcapa (S1) transfiere los datos y metadatos al sistema de almacenamiento final.

Varias particiones AHPIOS con diferentes configuraciones pueden ser ejecutadas en paralelo en un *cluster* (después de registrarse en el registro global). En la Figura 9 se puede ver un ejemplo de dos aplicaciones utilizando dos particiones AHPIOS diferentes.

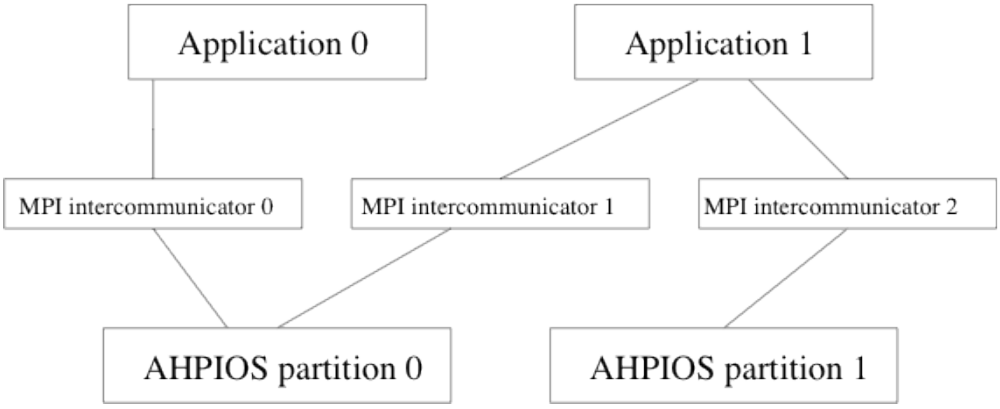

Figura 9: Diferentes particiones de AHPIOS en un mismo cluster

Para cada partición montada, se crea un intercomunicador MPI dedicado, a través del cual la capa MPI-IO se comunica con los servidores AHPIOS.

Los atributos de creación de las particiones (como la cantidad de recursos, el ancho de banda y la lista de recursos usados para datos y metadatos) son almacenados en un fichero de configuración, con la misma estructura que el ejemplo siguiente:

```
# The default stripe size of the partition
stripe_size = 64k
# Number of IOS
nr_ios = 4
# Storage resources of AHPFS servers
ahpios_server = n0:/data
ahpios_server = n1:/data
ahpios_server = n2:/data
ahpios_server = n3:/data
# Path of the metadata directory
metadata = n0:/data
```

4.3.1 Particiones

Las particiones en AHPIOS son elásticas: pueden aumentar o disminuir para incrementar el número de recursos de almacenamiento. Escalar hacia arriba solo implica el reinicio del sistema con un mayor número de servidores AHPIOS. En este caso, para los ficheros almacenados en el sistema antiguo con menor número de recursos, los usuarios pueden optar por conservar su estructura inicial o redistribuirla sobre los nuevos recursos de almacenamiento. Escalar hacia abajo implica una redistribución de los datos que ya no se encuentran accesibles desde ese momento, necesitando realizar este proceso en dos pasos: primero, todos los recursos de almacenamiento antiguos son montados al iniciar el sistema con el número de servidores AHPIOS antiguos y se realiza la redistribución; segundo, la partición antigua es desmontada y la nueva montada.

4.3.2 Cache distribuida

El acceso a los datos se realiza a través de la cooperación de las librerías de los clientes (ejecutándose en varios nodos de cómputo) y en el servidor AHPIOS. Las órdenes de transferencia de datos son controladas por un planificador de E/S paralela.

Un fichero AHPIOS debe ser particionado sobre varios servidores AHPIOS. Por defecto los ficheros son particionados sobre todos los servidores AHPIOS disponibles, pero el usuario puede controlar los parámetros de particionamiento a partir de parámetros de MPI (MPI hints).

A una partición AHPIOS se accede a través de una jerarquía de dos niveles de caches cooperativas. El usuario puede elegir si deshabilitar el primer nivel de cache cooperativa por razones de consistencia (si diferentes aplicaciones utilizan el mismo fichero, aunque los estudios han demostrado que se trata de un caso muy raro dentro de las aplicaciones científicas).

El primer nivel de cache distribuida, es gestionado a través de la cooperación de todos los procesos de la aplicación (utilizar solo un subconjunto de procesos también es posible), quien pone en común una fracción de memoria local del nodo sobre el que se está ejecutando. De esta forma, este nivel de cache es escalable con el número de procesos. Por analogía con el método *two-phase I/O* implementado en ROMIO, estos nodos se llaman agregadores, por ellos también juntan pequeños trozos de un fichero en una página cache más grande.

El primer nivel de cache distribuida funciona de la siguiente manera. Los bloques del fichero son *mapeados* mediante *Round-Robin* sobre todos los agregadores. La petición del fichero se dirige al responsable de los agregadores. El agregador apila junto con varias peticiones antes de acceder al siguiente nivel de cache. La comunicación con

el segundo nivel de cache es realizada de manera asíncrona a partir de un hilo de E/S, el cual, oculta a la aplicación la latencia de acceso al fichero.

El segundo nivel de cache distribuida es gestionado por los servidores de E/S de AHPIOS. Los bloques del fichero, son *mapeados* a los servidores AHPIOS, con el algoritmo *round-Robin*, siendo cada servidor responsable de transferir estos bloques al sistema de almacenamiento persistente. Cuando un servidor AHPIOS recibe una petición de un bloque del cual no es responsable, este servidor contesta a esta petición en cooperación con el resto de los servidores. Este enfoque es útil, al menos en dos escenarios. En primer lugar el cuello de botella de E/S relacionado con la computación puede ser disminuido de los servidores AHPIOS. En segundo lugar, en el sistema Blue Gene, un subconjunto de cómputo es asignado a un servidor de E/S, el cual es responsable de todas las peticiones sobre ficheros de este grupo, pudiéndose ahora servir al grupo en cooperación con otros servidores de E/S.

4.3.3 Acceso a los datos

El estándar de MPI-IO define dos grandes grupos de operaciones de E/S: colectivas e independientes. En el caso de AHPIOS, las operaciones independientes son realizadas de la misma manera en el *server-directed I/O* explicado más adelante en este mismo apartado, por lo que nos concentraremos en las operaciones colectivas.

Las operaciones colectivas son apropiadas para los trabajos paralelos debido a las siguientes cuatro características comunes en la aplicaciones de datos intensos de tipo científica. Primero, es frecuente que todos los nodos de cómputo realicen el acceso al mismo fichero. Segundo, cada nodo accede al fichero de manera no contigua y con una granularidad muy pequeña. Tercero, existe una gran degradación de la localización de espacio (cuando un nodo accede a algunas regiones del fichero, los otros nodos tienden a acceder a los datos vecinos). Cuarto, el acceso de escritura de datos es mayoritariamente no superpuesto.

Como se comentó anteriormente, la distribución de ROMIO contiene la implementación *two-phase I/O*, donde los buffers colectivos residen en los agregadores (los cuales son una parte de los nodos de cómputo). El diseño de AHPIOS incluye dos métodos colectivos de E/S, donde ambos están basados en vistas: *view-based client-directed* y *view-based server-directed*. A continuación se muestra una comparación entre *two-phase I/O*, *client-directed* y *server-directed*.

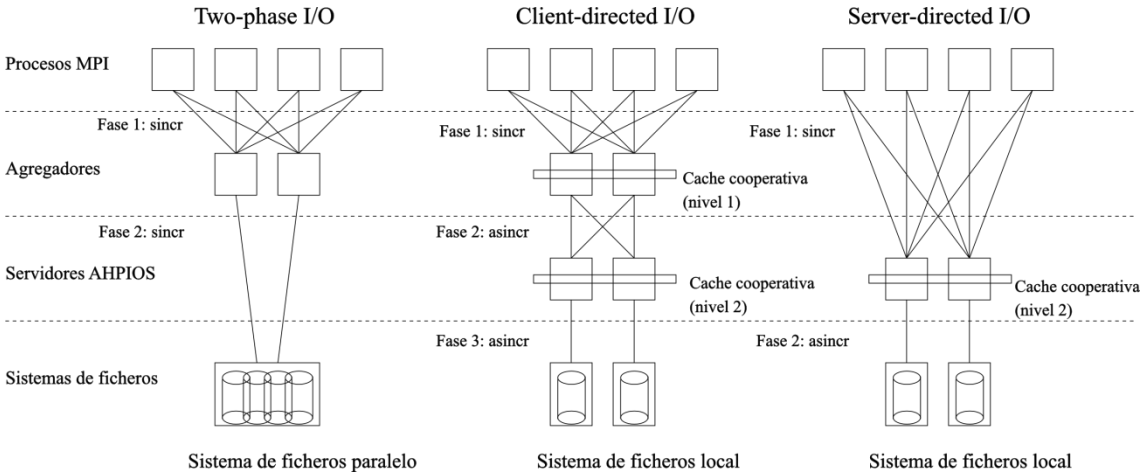

Figura 10: Comparativa funcionamiento métodos de E/S

Asumiendo que la vista ya ha sido declarada, la descripción es almacenada en el proceso cliente para el método *two-phase I/O*. Para client-directed *I/O* la vista es enviada a los agregadores, donde es almacenada para su uso futuro. Para *server-directed I/O* la vista es enviada a los servidores de AHPIOS, donde también será almacenada para su uso futuro.

Tanto para *two-phase I/O* como para *client-directed I/O,* el agregador representa un subconjunto del proceso MPI y son usados para agrupar pequeñas peticiones en una grande. Por defecto, todos los procesos MPI actúan como agregadores. Sin embargo, el usuario puede configurar el número de agregadores a partir de un argumento configurable de MPI.

Client-directed I/O consiste en tres fases. Cuando el client-directed *I/O* empieza, las vistas ya han sido almacenadas en los agregadores como una declaración. La primera fase es síncrona y consiste en repartir los datos entre los procesos MPI y los agregadores (pequeñas regiones del fichero son reunidos en el agregador para la escritura y son separados de los agregadores para la lectura). Las pequeñas regiones del fichero son transferidas de manera contigua entre cada par de procesos MPI y agregadores, y son agrupados o separados según la declaración de la vista almacenada anteriormente. En la segunda fase, los datos son enviados de manera asíncrona desde el primer nivel de cache cooperativa a los servidores AHPIOS. Sólo bloques de fichero completos son transferidos entre estos dos niveles de cache. Finalmente en la tercera fase, los datos son almacenados asíncronamente también desde la cache de los servidores AHPIOS al sistema de almacenamiento final.

Server-directed I/O consiste en dos fases: cuando el *Server-directed I/O* comienza, las vistas han sido también almacenadas en los servidores AHPIOS. La primera fase, es síncrona similar que con el *client-directed I/O,* exceptuando el hecho de que los datos son repartidos entre los procesos MPI y los servidores AHPIOS. La segunda fase es asíncrona y es igual que la tercera fase del *client-directed I/O.*

Se aprecia que existen dos grandes diferencias entre *client-directed I/O* y *server-directed I/O*: el lugar donde las vistas son almacenadas y el nivel intermedio de cache para el *client-directed I/O.* En consecuencia, pequeñas peticiones son unidas en grandes por los agregadores del *client-directed I/O* o en los servidores AHPIOS para *server-directed I/O*.

La Tabla 2 muestra una comparación el método *two-phase I/O* con los métodos *client-directed I/O* y *server-directed I/O.*

Operación	Two-phase I/O	Client-directed	Server-directed
Declaración de vista	Almacena vista en cliente	Envía vista a los agregadores AHPIOS	Envía vista a los servidores AHPIOS
Acceso fichero (metadatos)	Genera la posición del último acceso de las vistas y se lo manda a los agregadores.	n/a	n/a
Acceso fichero (cliente)	No contiguo	Contiguo.	Contiguo.
Acceso fichero (agregador)	No contiguo	No Contiguo.	n/a.
Acceso fichero (servidor)	Contiguo	Contiguo	No contiguo
Cache de fichero	No.	Primer nivel de cache cooperativa.	Segundo nivel de cache cooperativa.
Transferencia datos desde agregadores hacia servidores	Síncrona.	Asíncrona.	n/a.
Transferencia datos desde servidores hacia almacenamiento	n/a.	Asíncrona.	Asíncrona.
Cerrar fichero	n/a.	Tras verificar que todos los agregadores AHPIOS y servidores han puesto todos los datos.	Tras verificar que todos servidores han puesto todos los datos.

Tabla 2: Comparativa métodos de acceso

A diferencia de *two-phase I/O*, para *client-directed* y *server-directed I/O*, las vistas, representadas como tipo de datos, no son almacenadas en la aplicación del cliente, pero son decodificadas por la capa MPI-IO, serializadas y enviadas ya sea para agregadores de AHPIOS o servidores AHPIOS. Tras la recepción de la vista, los

agregadores o los servidores de-serializan y reconstruyen el tipo de datos original de la vista. La ventaja de esta solución es que los metadatos no son necesarios enviarlos por la red en el tiempo de acceso, porque la vista representa el acceso a ficheros almacenados de manera remota. Para *two-phase I/O* el patrón de acceso generado por la vista, tiene que ser enviado como una lista de tuplas (file offset, length). En el caso de *client-directed* y *server-directed I/O* los datos pueden ser enviados contiguos entre el cliente y el agregador/servidor. En *two-phase I/O*, no está cacheado en agregadores, pero solo de manera temporal es agrupado y enviado de manera síncrona al sistema de ficheros. En *client-directed I/O* los datos del fichero son cacheados en dos niveles de jerarquía de caches cooperativas y son enviados de manera asíncrona al sistema de almacenamiento final. En *server-directed I/O,* los datos son cacheados en el segundo nivel de la jerarquía de cache cooperativa y son enviados de manera asíncrona al almacenamiento. Las transferencias asíncronas permiten una superposición de cómputo de manera transparente, E/S relacionadas con la comunicación y el acceso al almacenamiento.

4.3.4 Coherencia de caches y consistencia

La coherencia cache está orientada a que ambos niveles de cache no almacenen más de una copia de los bloques de datos en cada nivel. Esta decisión es tomada debido a la frecuencia de los patrones de acceso en las aplicaciones paralelas: procesos independientes no sobrescriben regiones del fichero y existe una alta localización espacial entre los accesos de los procesos. El grano de acceso más pequeño que un bloque, ocurre solo en el primer nivel de cache, por ejemplo cuando se reparten los datos para operaciones colectivas. Los datos son transferidos entre ambos niveles de cache siempre con la granularidad de un bloque. Para escribir en un bloque del fichero menor que su tamaño en el primer o segundo nivel de cache, se necesita una operación de lectura-modificación-escritura, donde la operación final de escritura es realizada de forma asíncrona. Para *client-directed I/O*, la modificación de un bloque de fichero, siempre se realiza por un agregador del primer nivel de cache. *Server-directed I/O,* no usa el primer nivel de cache, por lo tanto la coherencia está orientada sólo a los

servidores, mediante el alojamiento de una copia del bloque del fichero. Existe un disparador en el *Server-directed I/O* que provoca la negación de acceso a los bloques de fichero de la cache de primer nivel desde el servidor, antes de que este haya terminado de realizar sus operaciones.

De acuerdo al estándar de MPI, MPI proporciona tres niveles de consistencia: consistencia secuencial a través de todos los accesos (usando un descriptor único del fichero), a través de todos los accesos (usando descriptores de ficheros creados tras la operación de apertura de modo atómico), y por último mediante la consistencia definida por el usuario.

AHPIOS proporciona una implementación parcial de la consistencia que define MPI. El acceso colectivo desde una aplicación es consistente, es decir, los datos son llevados al almacenamiento final desde los dos niveles de cache ya sea llamando a la primitiva *MPI_FILE_SYNC* o cerrando el fichero. El modo atómico para la independencia de acceso a los datos no se ha implementado, dado que la superposición de accesos no es frecuente en las aplicaciones paralelas y puede ser aplicada mediante consistencia semántica por el usuario, mediante la función *MPI_FILE_SYNC*, como se describe en el estándar de MPI.

4.3.5 Metadatos

En AHPIOS existe dos niveles de manejo de metadatos: nivel global de manejo de metadatos de las particiones AHPIOS y nivel local de cada partición.

- **NIVEL GLOBAL:** En AHPIOS no existe un servicio que realice la gestión de los metadatos globales. Los metadatos globales son mínimos y contienen solo información particular de las particiones del sistema de ficheros. Esta información es almacenada en un fichero compartido por todas las particiones llamado *registry*. Cada conjunto de servidores AHPIOS, gestiona el acceso a este fichero de manera atómica tanto para lectura como para modificarlo. Este acceso no debe causar un cuello de

botella debido a que el *registry* sólo es accedido cuando se monta o se desmonta una partición, siendo todas estas operaciones poco frecuentes.

El fichero *registry* almacena la estructura y la configuración dinámica de los parámetros de las particiones AHPIOS. Los datos estructurales son el nombre de la partición, los recursos de almacenamiento asignados a los servidores AHPIOS, el número de servidores, el tamaño por defecto de bloque y localización del fichero de metadatos. Los parámetros dinámicos incluyen el tamaño del buffer de red, el planificador de la política de E/S, el tamaño del buffer cache del servidor AHPIOS, etc. Los parámetros dinámicos pueden ser cambiados por el usuario cada vez que se activa la partición.

- **NIVEL LOCAL**: Por cada partición, uno de los servidores AHPIOS, juega el rol de gestor de metadatos. Este servidor maneja un espacio de nombres local y una lista de inodos, almacena y recupera el archivo de metadatos, y finalmente actualiza los metadatos en coordinación con el resto de los servidores. El nombre global de un fichero viene dado por lo tanto, por la suma de la ruta local del fichero y el nombre global único de la partición. El espacio de nombres local puede ser simplemente un directorio del espacio de nombres del sistema de ficheros del nodo de cómputo donde el servidor AHPIOS se está ejecutando. El almacenamiento típico de inodos en el fichero de metadatos viene definido por el valor del tamaño de bloque y el número de sectores. Por defecto, el número de sectores es el mismo que el número de servidores AHPIOS, el usuario puede modificar este valor a partir de un argumento configurable.

En el Capítulo 6 se describirá en detalle como se ha extendido la definición de metadatos, permitiendo gestionar a través de MPI una estructura de ficheros y directorios.

5 ANÁLISIS Y DISEÑO

El objetivo de esta etapa del proyecto es obtener una especificación detallada del sistema propuesto, que sirva de base para los implementadores del proyecto.

5.1 Análisis del sistema

Esta fase del proyecto, consiste en definir el problema entendiendo el dominio de aplicación y su funcionalidad, con el objetivo de que dicho *software* responda a las necesidades a las que va destinado, para seguidamente en el diseño del sistema resolverlo.

A pesar de que producir *software* de calidad implica recoger toda la funcionalidad deseada en forma de requisitos, la libertad que deja este proyecto es muy grande, debido a que no hay ninguna aplicación similar que realice la funcionalidad pedida. Es por ello que dichos requerimientos han ido variando a lo largo del desarrollo del sistema para finalmente conseguir solamente las funcionalidades necesarias para una buena evaluación.

5.1.1 Descripción detallada

Se desea construir un sistema que aproveche las capacidades de un sistema de ficheros paralelos como es MPI, pero que a su vez, ofrezca una interfaz totalmente transparente al usuario, permitiendo a los usuarios poder utilizarlo dentro de su propio espacio de usuario, es decir, como un espacio ajeno al núcleo del sistema.

Este Proyecto Fin de Carrera propone distintas mejoras al sistema de E/S paralela AHPIOS, descrito brevemente en el capítulo anterior. En este trabajo se quieren abordar todas las carencias que se han observado previamente. Las principales carencias encontradas son:

- El sistema sólo ofrece una interfaz basada en MPI, debido a que el prototipo está fuertemente ligado a la librería de paso de mensajes MPI.
- El soporte de metadatos es bastante limitado, ofreciendo soporte únicamente a ficheros, sin representar una estructura jerárquica.
- Las particiones de datos creadas dinámicamente no permiten la interacción entre los usuarios y entre distintas aplicaciones.

En este trabajo se pretende mejorar y extender todos los puntos débiles del prototipo anterior. La dificultad añadida de este trabajo consiste en modificar código previamente implementado, por lo que será necesario un análisis previo del mismo. Además, dicho código está integrado dentro de la librería de MPI, añadiendo si cabe, un nivel más de dificultad.

5.1.2 Especificación de Requisitos de Usuario

En este apartado el objetivo es identificar, clasificar y catalogar los requisitos de usuario obtenidos durante las distintas sesiones de trabajo realizadas con el tutor del proyecto.

Los requisitos deberán ser validados por todos los implicados antes de llevarse a cabo la implementación del sistema. Dichos requisitos mostrarán tanto lo que el sistema debe hacer de acuerdo a lo establecido, como la manera en que lo va a hacer. Es decir, se tendrán dos tipos de requisitos diferenciados:

- **Requisitos de capacidad:** representan aquello que el cliente desea que el sistema haga, son requisitos referidos al problema en cuestión que se quiere resolver.
- **Requisitos de restricción:** indica la manera que tendrá el sistema de resolver los problemas, la forma de interactuar con el sistema y las restricciones que habrá sobre éste.

Para la catalogación de los requisitos se utilizarán unas tablas cuyo contenido se explica a continuación:

- **Identificador→** cada requisito de usuario deberá estar identificado de forma unívoca con el objetivo de facilitar su trazabilidad.
- **Prioridad→**Cada requisito debe tener establecida una prioridad que sirve al desarrollador de referencia a la hora de planificar la creación del sistema.
- **Fuente→** Cada requisito tendrá como fuente al tutor o al alumno.
- **Necesidad→**Todo requisito puede ser esencial, deseable u opcional, los requisitos esenciales se deberán cumplir a rajatabla y no son negociables. Los otros dos pueden verse sujetos a modificaciones.

- **Descripción→** Describe de forma clara, verificable y concisa el requisito.
- **Estabilidad→** Algunos requisitos pueden presentarse fijos a lo largo de toda la vida del proyecto (estables) mientras que otros dependen de otros factores y pueden estar sujetos a cambios (no estables).
- **Claridad→** Este atributo define la cantidad de información proporcionada por el usuario/cliente.
- **Verificabilidad→** Propiedad del atributo que indica el grado de comprobación del requisito.

5.1.2.1 Requisitos de Capacidad

Identificador: RUC-001	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	Para la realización del proyecto será necesario migrar un sistema de E/S basado en MPI, a un nuevo sistema de ficheros.
Estabilidad:	No Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 3: Requisito de Capacidad RUC-001

Identificador: RUC-002	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	El sistema debe proporcionar una interfaz similar a POSIX.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☐ Alta ☒ Media ☐ Baja

Tabla 4: Requisito de Capacidad RUC-002

Identificador: RUC-003	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☐ Esencial ☒ Deseable ☐ Opcional	
Descripción:	Para la correcta integración con los sistemas de ficheros actuales, será necesario ampliar la gestión de metadatos del sistema de la versión previa de AHPIOS.
Estabilidad:	No Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 5: Requisito de Capacidad RUC-003

Identificador: RUC-004	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	La interfaz implementada dará soporte completo tanto a operaciones sobre ficheros como a directorios.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 6: Requisito de Capacidad RUC-004

Identificador: RUC-005	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☐ Esencial ☒ Deseable ☐ Opcional	
Descripción:	El sistema implementado tendrá dos métodos de despliegue de los servidores de E/S: estático y dinámico.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 7: Requisito de Capacidad RUC-005

Identificador: RUC-006	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	Los usuarios del sistema de ficheros pondrán escoger la ruta de montaje del mismo, dentro del espacio de usuario o sistema.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 8: Requisito de Capacidad RUC-006

Identificador: RUC-007	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	Los usuarios podrán configurar el sistema de ficheros, además de las nuevas funcionalidades implementadas en este fichero.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 9: Requisito de Capacidad RUC-007

Identificador: RUC-008	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	Será necesario obtener una evaluación completa de la escalabilidad y rendimiento del sistema implementado.
Estabilidad:	Estable
Claridad: ☐ Alta ☒ Media ☐ Baja	**Verificabilidad:** ☐ Alta ☒ Media ☐ Baja

Tabla 10: Requisito de Capacidad RUC-008

Identificador: RUC-009	
Prioridad: ☐Alta ☒ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☐ Esencial ☒ Deseable ☐ Opcional	
Descripción:	El sistema debe permitir acceder a los dispositivos de E/S a través de distintos tipos de redes: Ethernet e Infiniband.
Estabilidad:	No Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 11: Requisito de Capacidad RUC-009

5.1.2.2 Requisitos de Restricción

Identificador: RUR-001	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	El sistema solo ofrecerá funcionalidad en las plataformas Linux.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 12: Requisito de Capacidad RUR-001

Identificador: RUR-002	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	Para el desarrollo del proyecto se usará la implementación de MPI MPICH2 1.0.5
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 13: Requisito de Capacidad RUR-002

Identificador: RUR-003	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	Será necesario hacer uso de la herramienta FUSE.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 14: Requisito de Capacidad RUR-003

Identificador: RUR-004	
Prioridad: ☒Alta ☐ Media ☐ Baja	**Fuente:** Tutor
Necesidad: ☒ Esencial ☐ Deseable ☐ Opcional	
Descripción:	El sistema será desarrollado para infraestructuras cluster.
Estabilidad:	Estable
Claridad: ☒ Alta ☐ Media ☐ Baja	**Verificabilidad:** ☒ Alta ☐ Media ☐ Baja

Tabla 15: Requisito de Capacidad RUR-004

5.2 Diseño del sistema

En esta sección se detalla cómo se ha abordado la solución al problema planteado inicialmente. La Figura 11 muestra la arquitectura del sistema desarrollado en este Proyecto Fin de Carrera. Para cada uno de los componentes del sistema, se define en que espacio de ejecución se despliega.

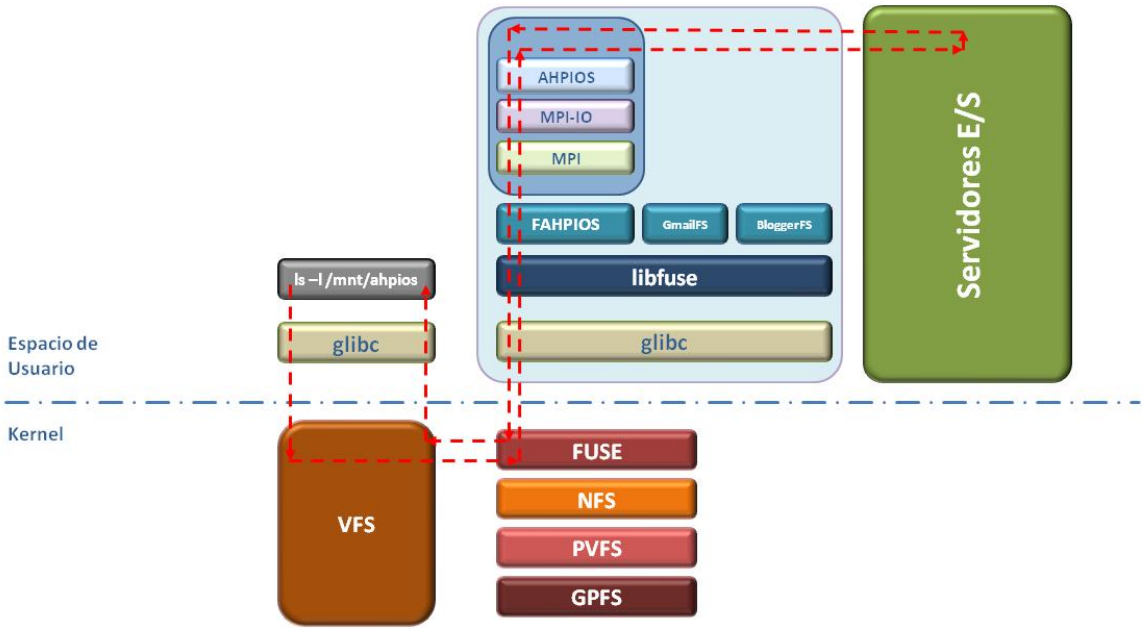

Figura 11: Arquitectura *software* del sistema

Partiendo de un sistema previamente montado, los usuarios podrán acceder al sistema de ficheros paralelo a través de la ruta de montaje. Como se muestra en primer lugar en la figura, los usuarios pueden acceder al sistema de ficheros mediante una interfaz POSIX, por ejemplo "*ls –l /mnt/ahpios/*". Mediante la librería de C *glibc*, las peticiones de E/S son encaminadas como llamadas al sistema al espacio del *kernel*. Una vez en el espacio del *kernel*, el componente VFS, virtualizará la petición de E/S, encaminando las peticiones al módulo del *kernel* asociado a ese sistema de ficheros. En este caso, el sistema de ficheros accedido a nivel del *kernel* será FUSE. FUSE será uno

de los módulos desplegado en el *kernel*. Se puede decir, que el módulo FUSE actúa como un controlador del sistema de ficheros, permitiendo la interacción entre el *kernel* y el espacio de usuario.

Una vez de vuelta al espacio de usuario, las peticiones de E/S serán transmitidas a la librería de usuario de FUSE (*libfuse*) a través de la librería del sistema *glibc*. La librería de usuario de FUSE permitirá desplegar distintos sistemas de ficheros. En concreto, para nuestro caso de estudio, se accederá al sistema de E/S paralela AHPIOS. Para que esto sea posible es necesario definir un nuevo componente que sirva de middleware entre la librería de FUSE y el sistema de ficheros, en este caso, llamado FAHPIOS (*File* AHPIOS). FAHPIOS será utilizado como interfaz entre FUSE y la librería MPI.

FAHPIOS hará uso de la librería MPI para realizar todas las operaciones de E/S sobre el sistema AHPIOS. FAHPIOS gestionará la capa de representación de metadatos del cliente. Es importante comentar que MPI no dispone de gestión de metadatos como tal, y por lo tanto esta funcionalidad deberá ser implementada mediante primitivas de la librería MPI. Todas las peticiones de E/S (*open*, *write*, *read*, *close*, etc.) serán traducidas a primitivas de MPI-IO. Un ejemplo simplificado de esta traducción podría ser la llamada del sistema *open*, que será traducida a la primitiva *MPI_File_open* de MPI-IO.

De aquí en adelante, el funcionamiento corresponderá con el funcionamiento estándar del sistema de E/S AHPIOS, detallado en el Capítulo 4.

6 DETALLES DE IMPLEMENTACIÓN

A continuación se detallan cada una de las aportaciones realizadas durante este proyecto al sistema de E/S paralela AHPIOS. Estas aportaciones son: la integración de AHPIOS con la interfaz ofrecida por FUSE, la extensión de los metadatos y el despliegue de los servidores de E/S de forma estática.

6.1 Integración de AHPIOS en FUSE

En este apartado se detalla cómo se ha integrado el sistema AHPIOS dentro del sistema FUSE. La etapa de integración conlleva diferentes pasos, desde la integración de la librería de MPI, como montar el nuevo sistema de ficheros.

En primer lugar ha sido necesario compilar FUSE usando el compilador proporcionado por la librería MPI, el cual, añadirá todas las primitivas de ROMIO que forman parte de MPI. Una vez terminado este paso, es posible invocar todas las primitivas y tipos de datos incluidos en la librería de MPI. De este modo, es posible, por

ejemplo, invocar la primitiva *MPI_File_open* desde un sistema implementado en FUSE. Esto nos permite de forma muy elegante hacer uso del sistema de E/S paralela AHPIOS, simplemente indicando a ROMIO que el sistema al que se desea acceder comienza por *ahpios:/* en todos los casos (*pvfs2:/* en caso de acceder al sistema de ficheros paralelo PVFS2). El siguiente paso corresponde con asignar a cada primitiva de POSIX, su correspondiente en ROMIO. Es importante señalar que la actual implementación de ROMIO no está destinada a gestionar ningún tipo de metadatos. Por lo tanto, aquellas llamadas que involucran metadatos han sido asignadas a la primitiva MPI_File_open. Como se verá en los siguientes apartados, ha sido necesario implementar toda la gestión de metadatos a través de las primitivas de ROMIO.

El siguiente punto a tener en cuenta es el montaje del nuevo sistema de ficheros implementado. La etapa de montaje se divide en dos tareas: despliegue estático/dinámico de los servidores de E/S, y el despliegue del cliente del sistema de ficheros en los nodos de cómputo del cluster.

Como se comentará en detalle en el Apartado 6.3, existen dos modos de despliegue de los servidores de E/S, estático y dinámico. En el caso del despliegue de los servidores de modo dinámico no es necesario realizar ningún paso adicional, debido a que es la propia librería de MPI la encargada de desplegar los procesos en el cluster (gracias a la primitiva MPI_Comm_spawn). En el caso de los servidores de E/S estáticos es necesario ejecutar el servicio en el nodo escogido, usando la siguiente llamada:

```
$ ./ioserver_connect nombreservicio
```

Donde *nombreservicio* será el nombre escogido por el usuario para identificar al proceso, y por lo tanto facilitar la conexión entre los clientes y los servidores. Es

importante señalar que los clientes deben conocer el nombre de todos los servidores de E/S a los que conectarán.

El siguiente y último paso consistirá en montar cada uno de los clientes. Para ello es totalmente necesario que todos los servidores de E/S hayan sido desplegados en caso de un despliegue estático. Esto se realizará mediante la ejecución del siguiente comando:

```
$ mpiexec –n 1 ./fahpios   -d   -f /mnt/fahpios
```

Al estar integrado FUSE con la librería MPI, para ejecutar el comando de montaje es necesario ejecutarlo como si de una aplicación paralela se tratase, pero sin embargo, es sólo ejecutada en el nodo local. Este requisito viene dado por la librería MPI, y por lo tanto tendrá que estar disponible para los usuarios del sistema.

6.2 Extensión e implementación del sistema de metadatos

La versión previa de AHPIOS soportaba un sistema de metadatos basado únicamente en ficheros, ofreciendo una vista plana de la jerarquía de directorios.

En el presente Proyecto Fin de Carrera se ha considerado necesaria la optimización del almacenamiento del sistema de metadatos, proponiendo una implementación basada en árboles B* (usada comúnmente en otros sistema de ficheros como NTFS[39] y PVFS), ya que actualmente tanto el almacenamiento como las búsquedas se realizan de forma secuencial (como se ha comentado anteriormente). Por tanto se ha optado por el uso de la base de datos DB Berkeley [41].

DB Berkeley es una biblioteca que gestiona una base de datos basada en claves y atributos. Actualmente, existe un gran conjunto de implemetaciones en disintos lengujes de programación y es usada por empresas como Google y HP, entre muchas otras. Esta

librería permite definir distintas estructuras de datos para albergar las claves y datos, permitiendo la optimización de los accesos dependiendo de la aplicación.

Para el prototipo implementado se optó por almacenar los metadatos en una estructura B*. Esta base de datos almacenará todos los metadatos por ficheros, usando como clave la ruta completa del fichero. Otro punto a favor de esta solución es el uso de cache. DB Berkeley soporta un sistema de cache de forma nativa dentro de la solución, permitiéndonos acelerar el acceso de aquellos ficheros que son accedidos con mayor frecuencia. Finalmente, una de las principales ventajas de usar DB Berkeley, es la reducción y re-utilización de código necesario para la programación.

La interfaz de FUSE implementada accede a los metadatos a través de la primitiva MPI_File_open, que usa los hints de ROMIO tanto para establecer los metadatos como para recuperarlos. En la Tabla 16 se describen todos los metadatos soportados en el prototipo implementado.

Nombre campo	Tipo	Descripción
Filename	char con tamaño MAX_NAME_SIZE	Nombre de fichero con la ruta completa.
global_fd	int	Descriptor global de fichero.
nr_ios	int	Número de servidores de entrada/salida.
Offset	off_t	Desplazamiento del puntero dentro del fichero.
Mode	mode	Modo de apertura del fichero.
Nlink	nlink	Número de enlaces simbólicos.
Uid	uid_t	Id del usuario propietario.
Gid	gid_t	Id del grupo propietario.
Size	off_t	Tamaño total del fichero en bytes.
Blksize	unsigned long	Tamaño de bloque para el sistema de ficheros de entrada/salida.
Atime	time_t	Instante en el que se accedió por última vez al fichero.
mtime	time_t	Instante en el que se modificó por última vez el fichero.
Ctime	time_t	Instante en el que se creó el fichero.

Tabla 16: Definición de la estructura de datos que definirán los metadatos

6.2.1 Flujo de metadatos

Es importante describir todas las entidades por las que circula una petición de metadatos del sistema de ficheros. En una petición de metadatos intervienen todas los elementos del sistema de ficheros, desde el cliente implementado en FUSE, pasando por la librería de MPI hasta los servidores de E/S. Por ejemplo, en una llamada normal al comando de consola "stat /mnt/ahpios/ejemplo" se ejecutarían las llamadas y el flujo de datos que describe la Figura 12.

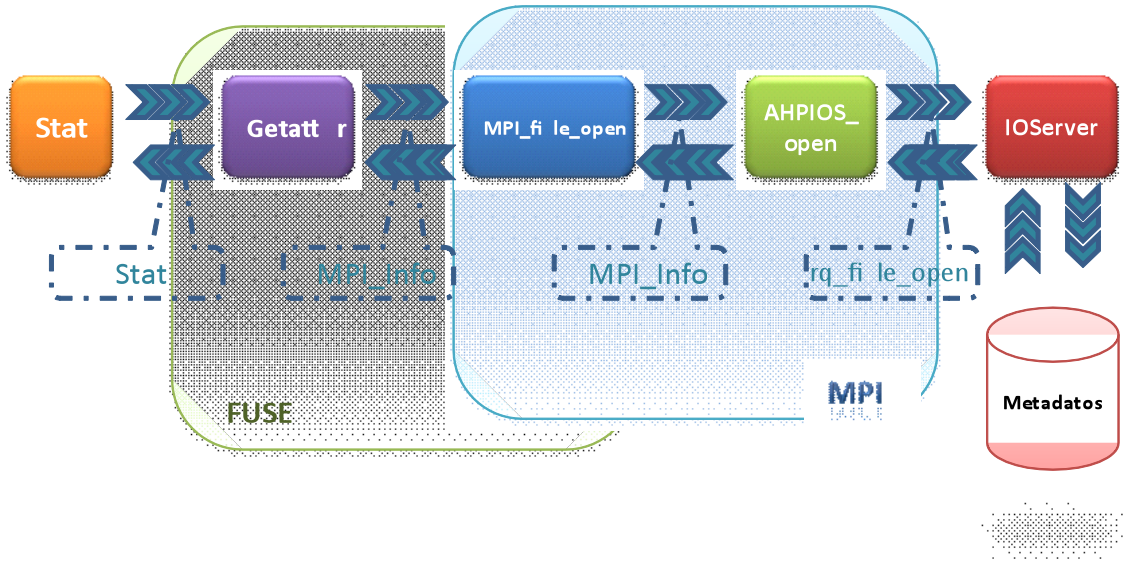

Figura 12: Flujo de una llamada al comando stat

La llamada al comando "stat" y tras su correspondiente mapeo dentro de FUSE provoca la ejecución de la función "Getattr", la cual básicamente traducirá los metadatos necesarios para poder entrar en MPI a través de la llamada a la función "MPI_File_open". Esta llamada de MPI provocará prácticamente de forma instantánea una llamada a la función homónima de AHPIOS "AHPIOS_open". Esta función de AHPIOS será la que mediante la estructura "rq_file_open" pase los metadatos para el acceso al fichero en los servidores de entrada salida.

Como se puede apreciar en la Tabla 17, la estructura de datos pasada hacia los servidores de entrada y salida "rq_file_open" contiene solamente la estructura de datos que guarda la información de los meta datos y el modo de acceso al fichero.

Nombre campo	Tipo	Descripción
access_mode	int	Modo de acceso.
statfs	struct stat	Estructura de datos de los metadatos descrita en Tabla 16.

Tabla 17: Definición de la estructura de datos rq_file_open

6.2.2 Gestión del árbol de directorios

Los directorios son un archivo más en el fichero de metadatos. Los nuevos directorios son almacenados en el fichero de metadatos. Para obtener el contenido de un directorio es necesario recorrer todo el fichero de metadatos. El algoritmo de obtención del contenido de un directorio es el siguiente:

```
if  (r->access_mode & ADIO_DIRECTORY) {
        int pos;
        struct metadata *mdir;
        pos = 0;
        while (mdir = lookup_metadata(pos)) {
            int level_m = num_dirfile(m->filename);
            int level_mdir = num_dirfile(mdir->filename);
            if (!strncmp( mdir->filename,m->filename,strlen(m->filename))) {
                if (level_m == level_mdir - 1) {
                    content = last_dirfile(mdir->filename,level_m+1);
                    if (content) {
                         strcat(content_str,content);
                         strcat(content_str,"#");
                     }
                }
             }
```

```
            pos++;
        }
        strcpy((char*)(rt+1), (char *)content_str);
}
```

6.2.2.1 Creación de directorios

La creación de directorios es similar a la creación de un fichero convencional. La única diferencia consiste en que a la llamada MPI_File_open se obtiene como argumento adicional la nueva etiqueta MPI_MODE_DIRECTORY, la cual indica que a la hora de crear el fichero, se procesa como un directorio en el conjunto de metadatos. El directorio se creará como una entrada en el fichero de metadatos con un tamaño de 4KBytes (4096 bytes), similar a la interfaz POSIX. El código necesario para crear un directorio a través de FUSE sobre MPI es el mostrado a continuación:

```
static int ahpios_mkdir(const char *path, mode_t mode)
{
        int res;
        MPI_File cFile;
        int rc;
        char *new_path;
        pthread_mutex_lock( &io_mutex );
        new_path = malloc(strlen(path) + strlen("ahpfs:") +1);
        strcpy(new_path,"ahpfs:");
        strcat(new_path,path);
        rc = MPI_File_open( MPI_COMM_WORLD, new_path, MPI_MODE_RDWR |
MPI_MODE_CREATE | MPI_MODE_DIRECTORY, MPI_INFO_NULL, &cFile );
        if (rc == MPI_SUCCESS){
            MPI_File_close (&cFile);
        }
        pthread_mutex_unlock( &io_mutex );
        free(new_path);
        return 0;
}
```

Como se puede ver en el código anterior, la primera acción que se realiza es el establecimiento de cerrojo de exclusión. A continuación se antepone la cadena “ahpfs:” al path que se ha pasado como argumento a esta función creando así la nueva cadena y formará el path. Este nuevo path será uno de los parámetros que se pase a la llamada ”MPI_File_open”, la cual será realmente la que cree el directorio como si fuese un nuevo fichero pero añadiendo la constante “MPI_MODE_DIRECTORY” que denotará el carácter de directorio. Finalmente, se comprueba el éxito de la llamada a MPI antes de cerrar el fichero y proceder al desbloqueo del cerrojo.

Parámetro FUSE	Parámetro MPI-IO	Descripción
buf	buf	El mapeo es directo.
fi	fi->fh	
count	int	Se usa para conocer los bytes accedidos en las llamadas MPI.

Tabla 18: mapeo de datos entre FUSE y MPI-IO

6.3 Despliegue de servidores de E/S estáticos

El prototipo implementado en este Proyecto Fin de Carrera soporta, además de un despliegue dinámico de los servidores de E/S, un despliegue puramente estático. Con estático nos referimos a que los servidores de E/S son desplegados de forma independiente de las aplicaciones, y será labor de los usuarios/administradores de sistemas ejecutar y configurar los servidores. Las aplicaciones tendrán que conectar con los servidores estáticos al inicio de su ejecución, y será necesario indicar a estas los parámetros de conexión necesarios para establecer la conexión entre los extremos.

La implementación del despliegue de los servidores de E/S estáticos viene motivada por los siguientes puntos. Primero, este tipo de despliegue permite indicar de forma sencilla en qué nodos del cluster se va a ejecutar dicho servicio. Este sistema permite por lo tanto, adaptar el despliegue a distintos tipos de arquitectura de E/S. Segundo, el despliegue estático permite por un lado, que distintas aplicaciones paralelas

accedan a una misma partición de FAHPIOS, y por otro, persistencia de los servidores de E/S.

A continuación se detallará la nueva implementación que permite compatibilizar ambos métodos de despliegue. Ante todo, es importante señalar que se pretendió minimizar los cambios en el código.

La modificación fundamental ha sido añadir el nuevo sistema de comunicación entre las aplicaciones y los servidores de E/S. La versión de despliegue dinámica hace uso de la primitiva MPI_Comm_spawn, añadida en la versión 2 del estándar de MPI. Esta primitiva despliega dentro del anillo de procesos, nuevos procesos distintos a los dados. De tal modo, que haciendo uso de esta primitiva es posible crear nuevos procesos. A la hora de comunicarse con los nuevos procesos se procede a usar intercomunicadores de MPI como se comentó en el punto 4.3. A la hora de implementar el despliegue estático se decantó por usar las primitivas MPI_Comm_accept y MPI_Comm_connect (incluidas también en el estándar MPI 2). Gracias a estas dos primitivas es posible conectar dos procesos independientes de MPI que se ejecuten en el mismo anillo de procesos. El resultado de ambas primitivas es un nuevo intercomunicador que será usado para enviar y recibir mensajes entre ambos extremos.

Es importante comentar, que la solución detallada anteriormente permite definir únicamente un canal de comunicación punto a punto. Sin embargo, dado que el sistema debe contar con más de un servidor de E/S, es necesario ofrecer un mecanismo de comunicación “uno a muchos”. Por lo que, tanto los clientes como los servidores dispondrán de un “poll” de comunicadores. Por un lado, los clientes disponen de un conjunto de intercomunicadores, cada uno identificado con el número de servidor de E/S. Por otro lado, los servidores de E/S dispondrán tanto de comunicadores como clientes estén conectados a dicho servidor de E/S. En un futuro se propone estudiar la escalabilidad de esta solución, debido a que un número significativo de conexiones puede afectar negativamente sobre el tiempo de comunicación.

7 VERIFICACIÓN DEL SISTEMA

A continuación se describe el conjunto de operaciones realizadas al sistema que han permitiendo verificar que las funcionalidades requeridas funcionan satisfactoriamente.

Finalmente, se adjunta la matriz de trazabilidad que relaciona cada una de las pruebas descritas con los requisitos presentados en la Sección 5.1.2.

7.1 Especificación del plan de pruebas

Identificador	PR-01
Descripción	Verificar el correcto funcionamiento de la creación de un directorio dentro del nuevo sistema de ficheros.
Metodología	Utilizar el comando "mkdir" para la creación de un directorio, acceder a dicho directorio mediante el comando "cd" y revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 19: PR-01

Identificador	PR-02
Descripción	Verificar que el comando "cp" realiza la copia de un fichero correctamente dentro del nuevo sistema de ficheros. El fichero origen debe proceder de fuera del nuevo sistema de ficheros.
Metodología	Para una correcta comprobación de esta prueba es recomendable utilizar un fichero de texto. Una vez copiado dicho fichero al nuevo sistema de ficheros se podrá comprobar el éxito de la prueba utilizando el comando "cat" o "tail". Además, mediante el comando "ls –lah nomfichero", se puede comprobar de un vistazo rápido que sus atributos se han copiado correctamente.
Resultado	Satisfactorio

Tabla 20: PR-02

Identificador	PR-03
Descripción	Verificar que el comando de consola “cp” realiza la copia de un fichero correctamente dentro del nuevo sistema de ficheros. El fichero origen debe proceder del nuevo sistema de ficheros.
Metodología	Para una correcta comprobación de esta prueba es recomendable utilizar un fichero de texto. Una vez copiado dicho fichero al nuevo sistema de ficheros se podrá comprobar el éxito de la prueba utilizando el comando “cat” o “tail”. Además, mediante el comando “ls –lah nomfichero”, se puede comprobar que sus atributos se han copiado correctamente.
Resultado	Satisfactorio

Tabla 21: PR-03

Identificador	PR-04
Descripción	Verificar que la creación de un fichero de texto en el nuevo sistema de ficheros se realiza de forma correcta, en cuanto a su contenido.
Metodología	Para una correcta comprobación es recomendable utilizar un fichero de texto. Una vez creado dicho fichero utilizando por ejemplo el editor “vi” las en el nuevo sistema de ficheros, se podrá comprobar el éxito de la prueba utilizando el comando “cat” o “tail”. Además, mediante el comando “ls –lah nomfichero”, se puede comprobar de un vistazo rápido que sus atributos se han copiado correctamente.
Resultado	Satisfactorio

Tabla 22: PR-04

Identificador	PR-05
Descripción	Verificar que el atributo de fecha de creación se establece correctamente.
Metodología	Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 23: PR-05

Identificador	PR-06
Descripción	Verificar que el atributo de fecha de modificación se establece correctamente.
Metodología	Para esta verificación se recomienda el uso del comando "stat nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 24: PR-06

Identificador	PR-07
Descripción	Verificar que el atributo “uid” se establece correctamente.
Metodología	Para esta verificación se recomienda el uso del comando “ls –lah nombfichero”. Para una correcta comprobación revisar el fichero de meta datos utilizando el comando “hexdump –C /tmp/.metadata_lios_xxxxx”
Resultado	Satisfactorio

Tabla 25: PR-07

Identificador	PR-08
Descripción	Verificar que el atributo “gid” se establece correctamente.
Metodología	Para esta verificación se recomienda el uso del comando “ls –lah nombfichero”. Para una correcta comprobación revisar el fichero de meta datos utilizando el comando “hexdump –C /tmp/.metadata_lios_xxxxx”
Resultado	Satisfactorio

Tabla 26: PR-08

Identificador	PR-09
Descripción	Verificar el correcto funcionamiento de los atributos de lectura, escritura y ejecución para un archivo que se encuentre dentro del nuevo sistema de ficheros.
Metodología	Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 27: PR-09

Identificador	PR-10
Descripción	Verificar el correcto funcionamiento del atributo que indica el tamaño de un fichero tanto a la hora de creación del mismo, como en la modificación de su contenido.
Metodología	Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 28: PR-10

Identificador	PR-11
Descripción	Verificar que el borrado de un fichero del nuevo sistema de ficheros funciona correctamente. Para ellos se usará el comando “rm”.
Metodología	Para esta verificación se recomienda el uso del comando “ls –lah nombfichero”. Para una correcta comprobación revisar el fichero de meta datos utilizando el comando “hexdump –C /tmp/.metadata_lios_xxxxx”
Resultado	Satisfactorio

Tabla 29: PR-11

Identificador	PR-12
Descripción	Verificar que el renombrado de un fichero del nuevo sistema de ficheros funciona correctamente. Para llevar a cabo esta labor se usará el comando “mv”.
Metodología	Para esta verificación se recomienda el uso del comando “ls –lah nombfichero”. Para una correcta comprobación revisar el fichero de meta datos utilizando el comando “hexdump –C /tmp/.metadata_lios_xxxxx”
Resultado	Satisfactorio

Tabla 30: PR-12

Identificador	PR-13
Descripción	Verificar que el borrado de un directorio del nuevo sistema de ficheros funciona correctamente. Se usará el comando "rmdir" sobre un fichero vacio para borrarlo del sistema de ficheros.
Metodología	Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 31: PR-13

Identificador	PR-14
Descripción	Verificar que el renombrado de un directorio del nuevo sistema de ficheros funciona correctamente. Al igual que la prueba PR-12, se usará el comando "mv" para modificar el nombre del directorio.
Metodología	Para el renombrado un directorio utilizar el comando "mv". Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 32: PR-14

Identificador	PR-15
Descripción	Comprobar que el tamaño en bytes de un directorio que se encuentre en el nuevo sistema de ficheros es 4096 bytes.
Metodología	Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 33: PR-15

Identificador	PR-16
Descripción	Comprobar el correcto funcionamiento del comando "chmod".
Metodología	Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 34: PR-16

Identificador	PR-17
Descripción	Comprobar el correcto funcionamiento del comando "chown".
Metodología	Para esta verificación se recomienda el uso del comando "ls –lah nombfichero". Para una correcta comprobación revisar el fichero de meta datos utilizando el comando "hexdump –C /tmp/.metadata_lios_xxxxx"
Resultado	Satisfactorio

Tabla 35: PR-17

Identificador	PR-18
Descripción	Esta prueba consiste en verificar el funcionamiento de los siguientes escenarios, variando el número de clientes y servidores de E/S: • Un cliente accede a una partición creada por un único servidor de E/S. • 6 clientes acceden a una partición creada por un único servidor de E/S. • Un cliente accede a una partición gestionada por 4 servidores de E/S. • 6 clientes acceden a una partición gestionada por 4 servidores de E/S.
Metodología	Se comprueba que el primer servidor de E/S es el que contiene correctamente el fichero de metadatos. También se verifica que el fichero de configuración y los parámetros almacenados en él, se transfieren correctamente a todos los servidores de E/S. Finalmente se verifica que el fichero ha sido distribuido entre los distintos servidores de E/S.
Resultado	Satisfactorio

Tabla 36: PR-18

Identificador	PR-19
Descripción	Una vez inicializado y puesto en funcionamiento el sistema, se procede a interrumpir el funcionamiento del gestor de procesos de MPI (mpd). El resultado esperado es que el sistema falle, y que el cliente notifique que no es posible la conexión con los servidores de E/S.
Metodología	Simplemente es necesario eliminar el proceso mediante "killall -9 mpd".
Resultado	Satisfactorio

Tabla 37: PR-19

Identificador	PR-20
Descripción	Una vez inicializado y puesto en funcionamiento el sistema, se procede a interrumpir el funcionamiento de un servidor de E/S cualquiera. El resultado esperado consiste en notificar, tanto al conjunto de servidores de E/S restante como a todos los clientes, que el sistema ha fallado debido a una desconexión.
Metodología	Simplemente es necesario eliminar el proceso mediante "killall -9 ioserver".
Resultado	Satisfactorio

Tabla 38: PR-20

7.2 Matriz de trazabilidad: Requisitos *software* y plan de pruebas

	PR-01	PR-02	PR-03	PR-04	PR-05	PR-06	PR-07	PR-08	PR-09	PR-10	PR-11	PR-12	PR-13	PR-14	PR-15	PR-16	PR-17	PR-18	PR-19	PR-20
RUC-001	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
RUC-002	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
RUC-003	X				X	X	X	X	X	X	X	X	X	X	X	X	X			
RUC-004	X	X	X								X	X	X	X	X					

Tabla 39: Matriz de trazabilidad

8 EVALUACIÓN

Durante el siguiente capítulo se describe como se ha llevado a cabo la evaluación del sistema. Para este fin, se ha realizado un conjunto determinado de pruebas de rendimiento para evaluar el sistema migrado sobre distintos escenarios.

Para la evaluación del sistema, se ha diseñado un plan de pruebas, consistente en la ejecución de un conjunto de herramientas de evaluación. Estas herramientas arrojan distintos medidores de rendimiento de la ejecución del sistema. La especificación del plan de pruebas viene detalla en el siguiente apartado, en donde se explicará cuales van a ser los mecanismos de evaluación y que resultados y medidas arrojan estos.

La ejecución de las pruebas se ha realizado en el cluster de investigación del grupo ARCOS, perteneciente a la Universidad Carlos III. El cluster está formado por 22 equipos. Cada nodo del cluster tiene las siguientes características *hardware* y *software*:

- Sistema operativo Debian Lenny.
- Quad core Intel(R) Xeon(R) CPU E5405 @ 2.00GHz.
- Memoria RAM 4 Gbytes.
- Todos los nodos se encuentran comunicados, a través de una red Ethernet de 1 GBytes/segundo.

Para todos los casos se han establecido los siguientes parámetros de configuración:

- Tamaño del buffer colectivo a 16 Mbytes
- Tamaño de bloque del sistema de ficheros: 128 KBytes.
- Tamaño de buffer de red 1Mbyte.
- Número de servidores de E/S entre 1 y 4.

También es importante mencionar, que debido a limitaciones *hardware*, las aplicaciones y los servidores de E/S comparten nodos, por lo que es posible que haya nodos en los que se ejecuten ambas entidades. En todas las ejecuciones se estableció un proceso por nodo.

8.1 Metodología de evaluación

A continuación, se mostrarán los resultados obtenidos en la etapa de evaluación del sistema propuesto. En todos se comparará nuestro sistema con otras soluciones actuales y conocidas.

8.1.1 Collperf

Collperf es un *benchmark* de prueba integrado dentro de la distribución de ROMIO, utilizado para estimar el funcionamiento de las rutinas de E/S colectivas de MPI-IO. La prueba mide la tasa de transferencia de E/S tanto para la escritura como para la lectura de un archivo, con un patrón acceso definido como una serie tridimensional distribuida por bloques. La división de datos se realiza por la asignación de un número de procesos sobre cada dimensión cartesiana.

Para la E/S colectiva de ROMIO, todos los nodos intervienen en las operaciones de acceso al fichero. El tamaño del buffer colectivo se estableció a 16 MBytes (valor definido por defecto en la versión 1.0.8 de MPICH2). Tanto para PVFS como para AHPIOS y FAHPIOS, el número de servidores de E/S se fijó a 4.

En la Figura 13, Figura 14 y Figura 15 se muestran los resultados obtenidos con el *benchmark* Coll_perf, únicamente para operaciones de escritura sobre una matriz tridimensional de enteros de 128, 256 y 512 enteros respectivamente.

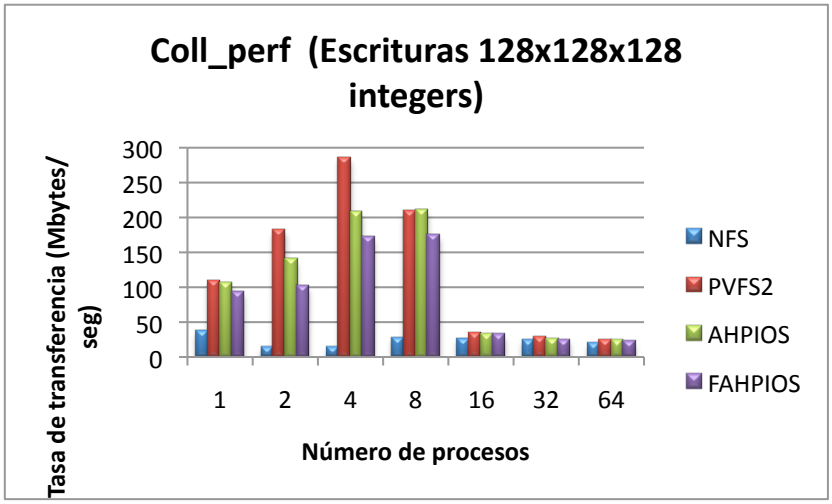

Figura 13: Coll-Perf con operaciones de escritura sobre una matriz de 128x128x128 enteros

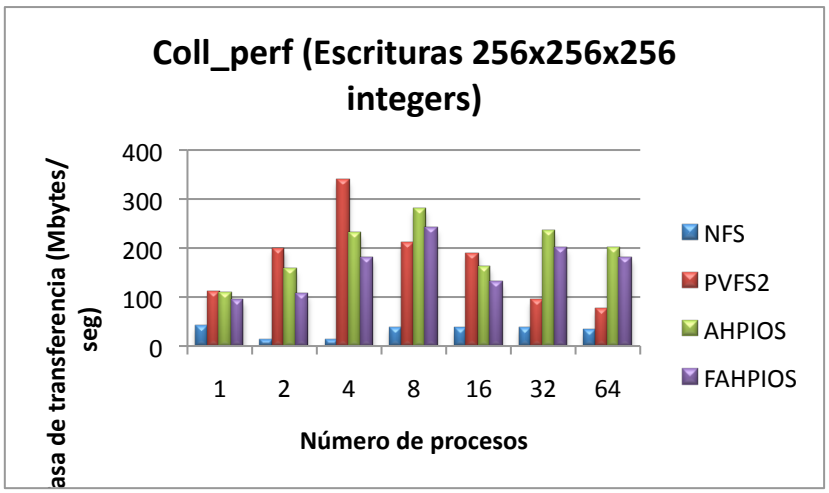

Figura 14: Coll-Perf con operaciones de escritura sobre una matriz de 256x256x256 enteros

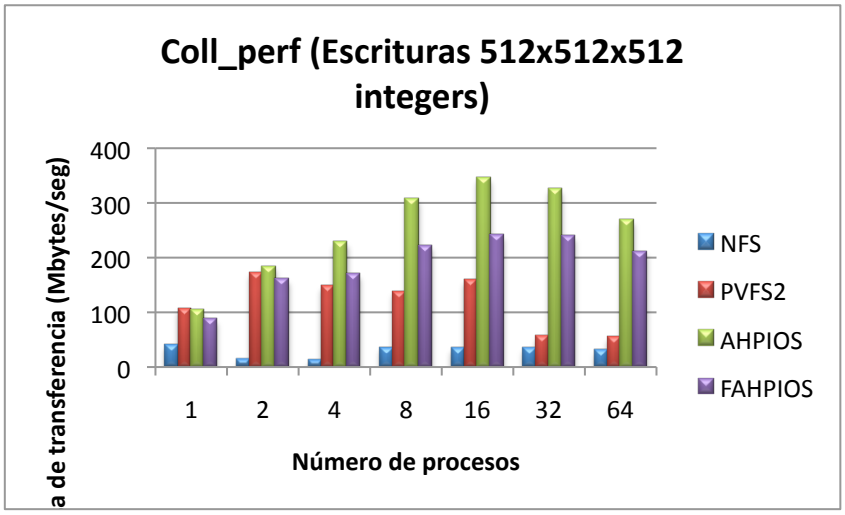

Figura 15: Coll-Perf con operaciones de escritura sobre una matriz de 512x512x512 enteros

Los resultados arrojan que la solución que obtiene un mejor rendimiento en la mayoría de los casos es AHPIOS. Esto es debido a que en AHPIOS las vistas que proporciona MPI están integradas fuertemente con esta solución de E/S paralela. Sin embargo, esta característica no está presente en FAHPIOS, el cual realiza accesos secuenciales sobre el sistema de ficheros. Es importante comentar que a pesar que el tamaño del buffer colectivo se estableció a 16 MBytes a nivel de ROMIO, la interfaz de ficheros proporcionada por FUSE particiona dichos accesos en bloques de 128 Kbytes (mayor tamaño permitido por la última versión de FUSE). Por último, se observa que el

rendimiento en todos los sistemas evaluados decrece drásticamente cuando los servidores de E/S se despliegan en los nodos de aplicación, sin embargo, se observa que los sistemas AHPIOS y FAHPIOS obtienen mejores resultados en este escenario.

En la Figura 16, Figura 17 y la Figura 18 se muestran los resultados obtenidos con este *benchmark* para operaciones de lectura sobre una matriz tridimensional de enteros de 128, 256 y 512 enteros respectivamente.

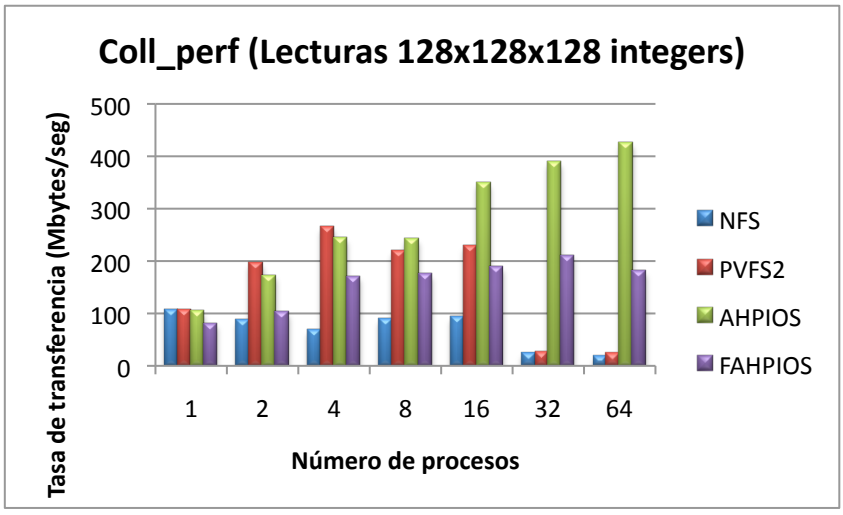

Figura 16: Coll-Perf con operaciones de lectura sobre una matriz de 128x128x128 enteros

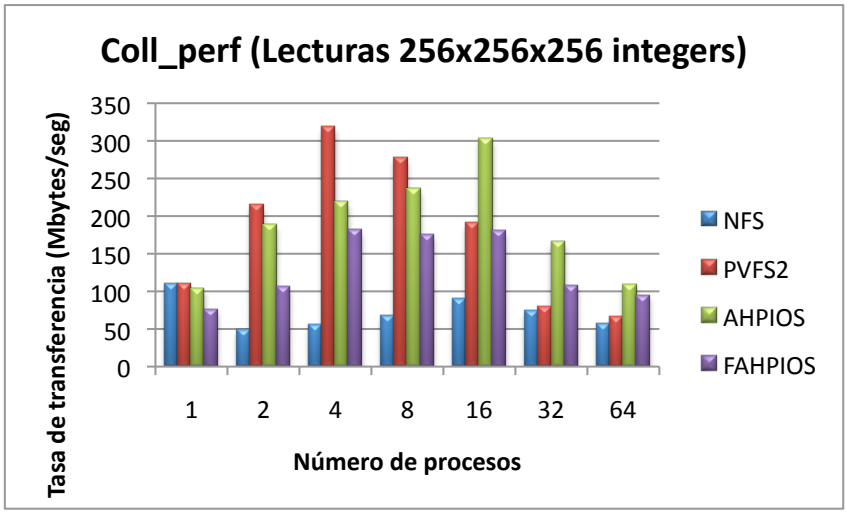

Figura 17: Coll-Perf con operaciones de lectura sobre matriz de 256x256x256 enteros

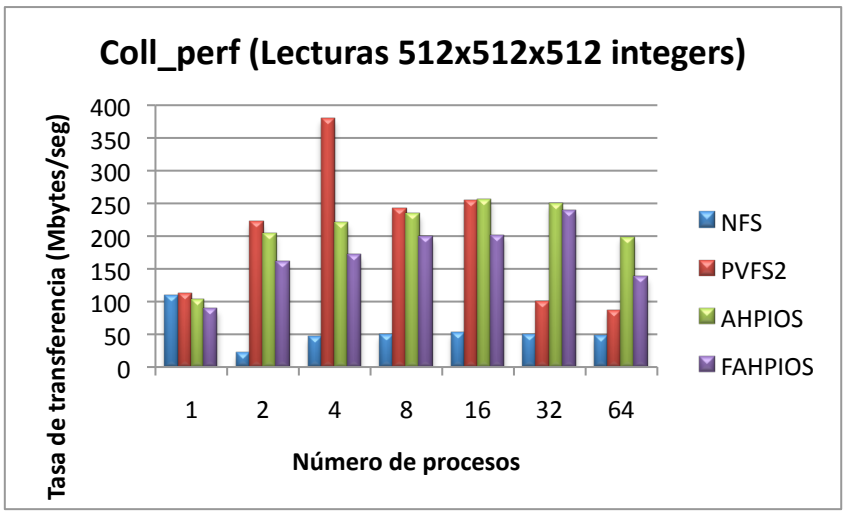

Figura 18: Coll-Perf con operaciones de lectura sobre una matriz de 512x512x512 enteros

Como se observa en las figuras, el sistema de ficheros NFS alcanza el máximo porcentaje de utilización de la red que dispone el cluster. Adicionalmente, se observa como el sistema de cerrojos afecta al rendimiento cuando tenemos más de un proceso accediendo al fichero. Otra conclusión que se obtiene de las figuras es que los sistema de E/S paralela AHPIOS y FAHPIOS superan en rendimiento al resto de soluciones cuando se aumenta el tamaño de los datos accedidos y cuando aumenta el grado de paralelismo en la aplicación. Finalmente apuntamos que en la Figura 16 se obtienen resultados difíciles de explicar. Por ello, se propone como trabajo futuro investigar detalladamente cuales son las causas del bajo rendimiento para los sistemas NFS y PVFS2.

8.1.2 MPI-TILE-IO

MPI-Tile-IO en un *benchmark* destinado a E/S paralela, utilizado para evaluar el rendimiento de los accesos a datos no contiguos. En este sistema de evaluación, el acceso de E/S a los datos no contiguos es realizado en un único paso, mediante el uso de operaciones colectivas de E/S. Este *benchmark* es muy conocido dentro de la comunidad científica, referenciado en multitud de publicaciones científicas.

El *benchmark* evalúa el rendimiento del acceso simultáneo a una matriz bidimensional de datos, mediante la simulación del tipo de trabajo existente en algunas aplicaciones de visualización (representación gráfica de partículas, animaciones, etc). El *benchmark* dispone de una gran cantidad de opciones de configuración, como son el tamaño de datos a tratar, la distribución de los datos, solapamiento de los mismos, etc.

La Figura 19 y Figura 20 muestran los resultados obtenidos con este *benchmark* para operaciones de escritura y lectura respectivamente. Tanto para PVFS como para AHPIOS y FAHPIOS, el número de servidores de E/S se fijó a 4. El *benchmark* se ha configurado de tal forma que se acceda a un único fichero de 1 Gbyte, con tamaño de registro de 8 números de alta precisión.

Los resultados obtenidos arrojan que la mejor solución en todos los escenarios es AHPIOS. Como se puede observar FAHPIOS obtiene peores resultados que AHPIOS y PVFS2 en las escrituras. Esto es debido principalmente al pequeño tamaño de acceso al fichero proporcionado por FUSE. El tamaño de acceso al fichero proporcionado por FUSE no llega a ser el adecuado para el ancho de banda ofrecido por la red de comunicación. En las lecturas, tanto AHPIOS como FAHPIOS obtienen los mejores resultados en todas las situaciones. Esto es debido porque ambas soluciones usan un tamaño de buffer de red de 1 Mbyte, obteniendo ventajas del ancho de banda proporcionado por la red.

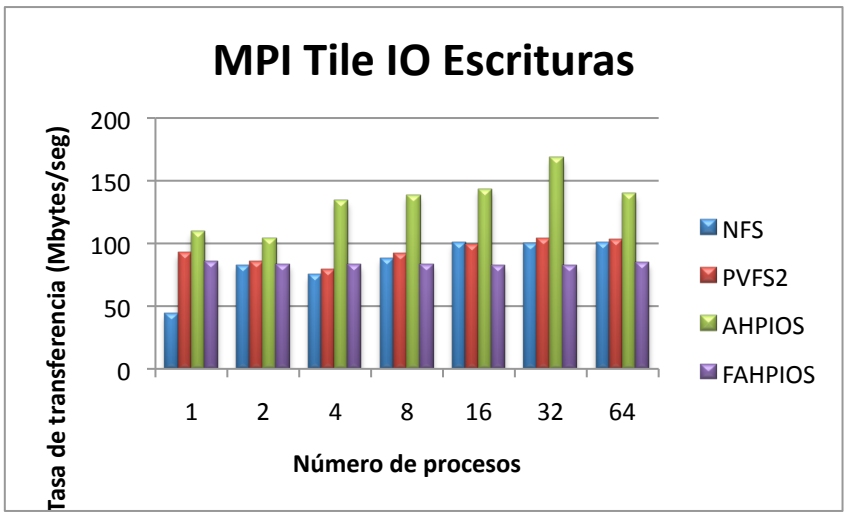

Figura 19: MPI Tile IO para operaciones de escritura

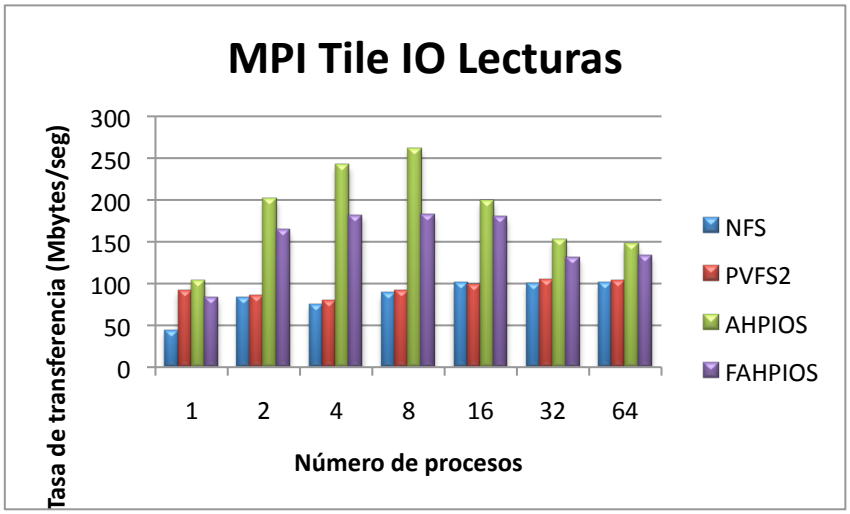

Figura 20: MPI Tile IO para operaciones de lectura

8.1.3 NFSSTONES

NFSSTONES [33] es un *benchmark* propuesto en los sistemas de ficheros distribuidos. Gracias a este *benchmark* se pueden realizar pruebas independizando tanto los fundamentos de la configuración de una red como de los protocolos utilizados para

la comunicación. Esto permite una gran variedad de combinaciones de sistemas de ficheros y redes, que pueden ser comparados con una sencilla aplicación.

Los resultados que reporta *NFSSTONES* están basados en clientes que solicitan un número de operaciones sobre distintos ficheros y devuelve los tiempos empleados en completar cada una de las operaciones sobre los dichos ficheros. Los resultados se representan con una puntuación conocida como *nfsstones*.

Hay que tener en cuenta las siguientes asunciones antes de ejecutar este *benchmark*:

- Los clientes son ejecutados en el sistema de ficheros de red.
- Tratar al sistema de ficheros como una caja negra ya que el *benchmark* es independiente de los fundamentos del sistema de ficheros, protocolos y arquitectura de red, así como el sistema operativo. Obviamente, si se estuviese utilizando la actual versión *NHFSSTONES[35]* sería necesario tener *NFS* corriendo, ya que esta versión del *benchmark* inspecciona las estadísticas de *NFS*.

NFSSTONE es un *benchmark* sencillo que toma algunas asunciones como por ejemplo qué es ejecutado en un sistema de ficheros de red, lo cual permite utilizar a las llamadas estándar de un sistema de ficheros (*read, write, lookup* y estado de un fichero). No se realiza ninguna asunción en cuanto a los fundamentos de los sistemas de ficheros (con soporte de red o no), protocolos y arquitectura de red ni en cuanto al sistema operativo utilizado.

Como inconveniente, NFSSTONE necesita la sincronización manual de los clientes distribuidos. Si la carga de trabajo requerida por el servidor excede del ratio máximo de peticiones de un cliente sencillo, entonces muchos clientes podrían ser requeridos para generar la suficiente carga; este sería el caso de un servidor potente que posee un cliente mucho más potente.

Para la ejecución de este *benchmark*, se han definido cinco escenarios o casos de evaluación distintos, con los que se medirá el rendimiento de distintos sistemas de ficheros (NFS, PVFS2 con 1 y con 4 servidores de E/S y FAHPIOS, desarrollado en este Proyecto Fin de Carrera, con 1 y con 4 servidores de E/S). Estos casos de prueba se describen en la Tabla 40.

	Configuración del test		Descripción
1	TOP_DIRS 1 BOT_DIRS 1 FILES_PER_DIR 100	FILE_CREATES 100 BLOCKS_PER_FILE 1 FILE_LOOKUPS 1	Todo el test se realizará una vez, donde cada operación R/W se realizará una vez, con 100 ficheros por cada directorio que serán creados 100 veces y cada uno de esos ficheros estará formado por un bloque. Se realizará una operación de búsqueda sobre cada fichero. Este test es un sencillo test de referencia para el resto, ya que se ha configurado con parámetros que no cargan en exceso al sistema.
2	TOP_DIRS 1 BOT_DIRS 1 FILES_PER_DIR 10	FILE_CREATES 10 BLOCKS_PER_FILE 1 FILE_LOOKUPS 5000	Todo el test se realizará una vez, donde cada operación R/W se realizará una vez, con 10 ficheros por cada directorio que serán creados 10 veces y cada uno de esos ficheros estará formado por un bloque. Se realizarán 5,000 operaciones de búsqueda sobre cada fichero. Con ello se busca un test con más peso en las búsquedas.
3	TOP_DIRS 1 BOT_DIRS 1 FILES_PER_DIR 10	FILE_CREATES 10 BLOCKS_PER_FILE 1K FILE_LOOKUPS 10	Todo el test se realizará una vez, donde cada operación R/W se realizará una vez, con 10 ficheros por cada directorio que serán creados 10 veces y cada uno de esos ficheros estará formado por 1,024 bloques (1k). Se realizarán 10 operaciones de búsqueda sobre cada fichero. Este test busca el comportamiento del sistema ante tamaños de bloque de potencias de 2 (1k).
4	TOP_DIRS 10 BOT_DIRS 1 FILES_PER_DIR 100	FILE_CREATES 10 BLOCKS_PER_FILE 8 FILE_LOOKUPS 10	Todo el test se realizará completamente 10 veces, donde cada operación R/W se realizará una vez, con 100 ficheros por cada directorio que serán creados 10 veces y cada uno de esos ficheros estará formado por 8 bloques. Se realizarán 10 operaciones de búsqueda sobre cada fichero. Con este test se busca observar el comportamiento del sistema ante la carga del mismo mediante la repetición del mismo test (10 veces).

	Configuración del test		Descripción
5	TOP_DIRS 10 BOT_DIRS 1 FILES_PER_DIR 10K	FILE_CREATES 10K BLOCKS_PER_FILE 1 FILE_LOOKUPS 10	Todo el test se realizará completamente 10 veces, donde cada operación R/W se realizará una vez, con 10,000 ficheros por cada directorio que serán creados 10,000 veces y cada uno de esos ficheros estará formado por un bloque. Se realizarán diez operaciones de búsqueda sobre el mismo. Este test busca cargar el sistema para ver su comportamiento.

Tabla 40: Descripción de los escenarios de evaluación de NFSSTONES

Todos los *test case* de la tabla anterior se han configurado con el tamaño de bloque de AHPIOS (128 Kbytes) facilitando la tarea a FUSE. Además, no se han tenido en cuenta medidas de lecturas para enlaces simbólicos. En todos los casos un solo nodo accede al sistema de ficheros. Sin embargo, el *benchmark* crea diferentes procesos pesados (*forks*) para acceder a distintos ficheros.

La Figura 21 muestra los resultados obtenidos con este *benchmark* en cada uno de los escenarios propuestos. En la evaluación no se ha contemplado la comparación con AHPIOS debido a que este sistema no implementa un sistema de metadatos completo. Los resultados se muestran normalizados en relación con el sistema de ficheros distribuido NFS. En la mayoría de los escenarios evaluados, FAHPIOS es hasta un 300% más lento que NFS. Esto es debido a la sobrecarga añadida por la librería MPI en caso de AHPIOS, y en caso de FUSE y MPI en caso de FAHPIOS. Es importante mencionar que una implementación previa del sistema de metadatos basada en fichero proporcionaba resultados por encima del 800%. En este caso hemos observado una gran mejoría al usar DB Berkeley [41] como base de datos ligera para los metadatos del sistema. Finalmente, observamos en el caso 3 que FAHPIOS obtiene mejores resultados que PVFS debido a que en este escenario se trabaja con ficheros de gran tamaño y el número de operaciones sobre los metadatos es reducida.

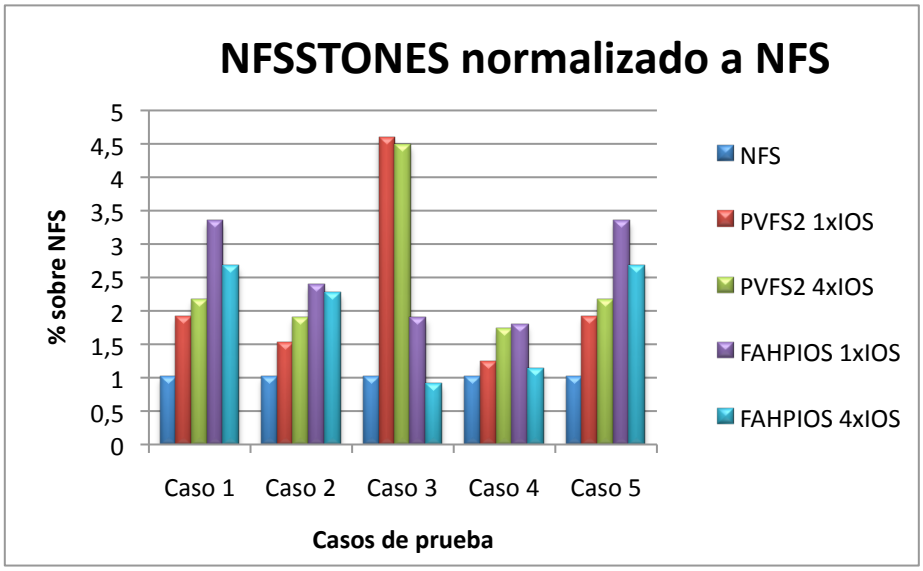

Figura 21: NFSSTONES normalizado a NFS

8.1.4 BTIO

Desarrollado por NASA Advance Supercomputing Division [37], el conjunto de *benchmark* paralelos NPB-MPI BT versión 3.2 E/S [38] es formalmente conocido como el *benchmark* BTIO [36]. BTIO presenta un modelo de particionado en bloques tridiagonal sobre una matriz tridimensional a través de un número cuadrado de procesos. Cada proceso es responsable de múltiples subconjuntos cartesianos del conjunto completo de datos, cuyo número incrementa con la raíz cuadrada del número de procesos participantes en la computación. La Figura 22 ilustra el modelo de particionado de este *benchmark* con un ejemplo de nueve procesos.

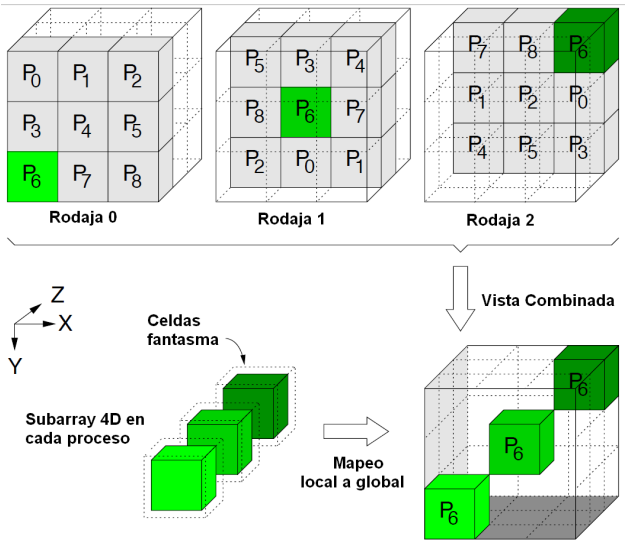

Figura 22: Modelo de particionado de BTIO

BTIO utiliza distintas opciones para usarse con MPI colectivo o con E/S independiente. En BTIO, 40 matrices se escriben consecutivamente a un fichero compartido abriendo una tras otra. Cada matriz tiene que ser escrita en formato canónico de la principal fila en el fichero. Inicialmente, todos los nodos de computación abren colectivamente un fichero y declaran vistas sobre las regiones relevantes del fichero. Después de cada cinco pasos de computación el nodo escribe la solución a un fichero a través de una operación colectiva. Al final el fichero resultante es leído colectivamente y la solución verificada ante posibles correcciones. El *benchmark* notifica el tiempo total incluyendo el tiempo empleado para escribir la solución al fichero. Sin embargo, la fase de verificación del tiempo contenido en la lectura de los ficheros de datos no está incluida en el tiempo total notificado. El modelo de acceso de BTIO es de paso anidado con una profundidad de anidamiento de 2 con la granularidad de acceso a fichero dada en la Tabla 41.

Número de procesos	Clase B	Clase C
9	1360	2160
16	1000	1640
25	800	1280
36	680	1080
49	600	920
64	480	800

Tabla 41: Granularidad del acceso a fichero de BTIO (en bytes)

En la Figura 23 y la Figura 24 se muestran los resultados obtenidos para este *benchmark*. Las figuras muestran el tiempo invertido en las operaciones de escritura y lectura respectivamente. En el caso de las escrituras el tiempo necesario para cerrar el fichero está incluido en los resultados. El *benchmark* genera y lee un fichero de 1,6 Gbytes de forma colectiva. En la evaluación se han desplegado hasta 4 procesos por nodo (recordar que los nodos cuentan con 4 núcleos cada uno). Por lo tanto, se han usado hasta 16 nodos de cómputo. Además es importante mencionar que en este caso, los servidores de E/S (4 en este caso), se despliegan en nodos distintos a los de cómputo (hasta completar los 20 nodos que se disponían durante la etapa de evaluación de este Proyecto Fin de Carrera). Sin embargo, tanto en AHPIOS como en FAHPIOS la localización de los servidores de E/S no está controlada, y por lo tanto, en posible sobrecarga cuando superamos los 16 nodos de cómputo. Está carencia viene dada por la implementación de MPI utilizada en este trabajo.

En los resultados podemos observar que las soluciones propuestas en este Proyecto Fin de Carrera son tan competitivas como la solución PVFS2.

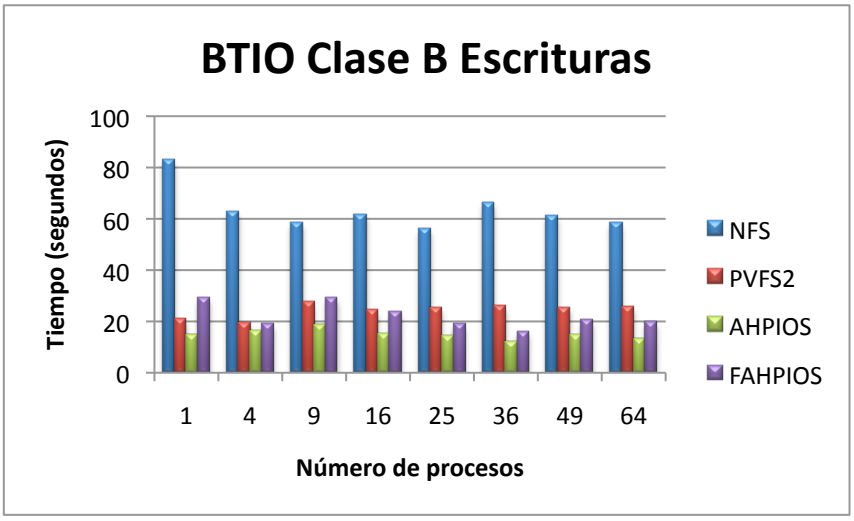

Figura 23: BTIO Clase B escrituras

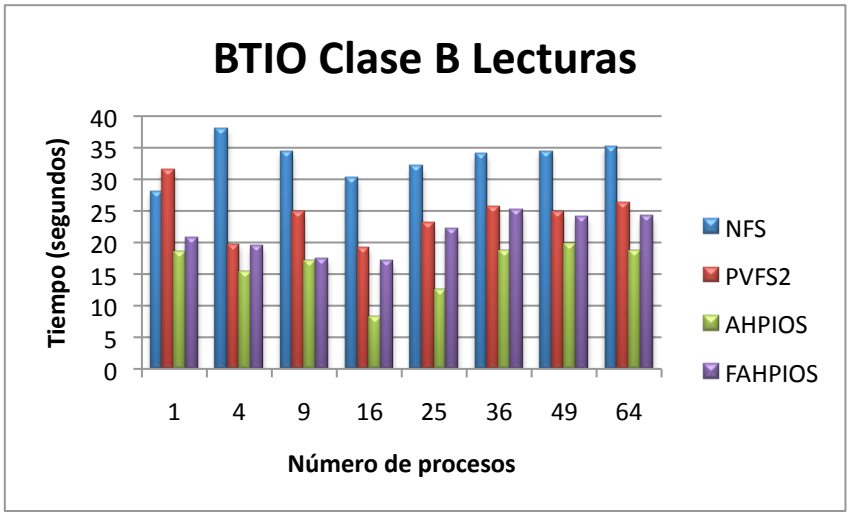

Figura 24: BTIO Clase B lecturas

8.2 Conclusiones de la evaluación

Tras la realización del análisis ofrecido durante los puntos anteriores, a continuación se realizan unas conclusiones obtenidas de la evaluación del sistema propuesto en este Proyecto Fin de Carrera.

En primer lugar se ha observado que el rendimiento del sistema FAHPIOS es víctima de la sobrecarga generada por las librerías MPI y FUSE. A pesar de ello, el sistema ofrece adaptabilidad y portabilidad. Por otro lado es posible desplegar tanto AHPIOS como FAHPIOS en un mismo sistema, el primero para ofrecer un sistema de alto rendimiento de E/S paralela, y el segundo para ofrecer una interfaz conocida a los usuarios de las aplicaciones paralelas.

En segundo lugar la evaluación realizada con el *benchmark NFSSTONE* demuestra que nuestra solución es tan solo un 50% más lenta en comparación con otros sistemas de E/S paralela como PVFS2. Sin embargo, en escenarios en los que el número de operaciones de metadatos es reducido, nuestra solución es capaz de mejorar a PVFS2.

Por último, como se ha comentado en anteriores ocasiones, nuestro sistema se ve afectado por la limitación impuesta por FUSE, que no permite a nuestra solución adaptarse a las condiciones de la red de intercomunicación, especialmente con las operaciones de escritura.

9 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se revisan los objetivos planteados al inicio del proyecto, así como del grado de cumplimiento de los mismos. Por último se proponen futuras mejoras y líneas de trabajo.

Además se comentará la metodología seguida durante la realización del proyecto, así como la planificación y el presupuesto final para la realización del mismo.

9.1 Conclusiones

Como objetivo principal del proyecto se ha conseguido migrar el sistema de E/S paralela AHPIOS hacia el sistema operativo Windows®, permitiendo el ahorro de tiempo en la ejecución de aplicaciones paralelas.

El proyecto no ha quedado finalizado hasta que se ha comprobado que todos los objetos marcados han sido completados:

- Se ha adaptado un sistema de E/S paralela sobre MPI a la interfaz POSIX mediante la herramienta FUSE.
- Se ha estudio la herramienta FUSE y de las distintas posibilidades de implementación que posee.
- Se ha estudiar y analizado el sistema de E/S paralela AHPIOS y se han conocido todas sus posibilidades de funcionamiento y posibles métodos de adaptación.
- Se han ampliado el conjunto de metadatos existentes, manteniendo en la medida de lo posible, la compatibilidad con la interfaz de POSIX.
- Se ha diseñado e implementado el despliegue de los servidores de E/S de forma estática.
- Se ha cuantificado la sobrecarga del sistema basado en FUSE sobre el sistema original de E/S AHPIOS.

Además de los objetivos afines a la realización de este proyecto, se derivan objetivos meramente académicos. Gracias a la realización de este proyecto se han profundizado en los siguientes aspectos:

- Se ha aprendido a gestionar un proyecto grande que ya ha sido iniciado, además de las dificultades que entraña una migración y las diferencias existentes entre proyectos en Linux.

- Se ha profundizado en las características del lenguaje C.
- Estudio sobre funcionamiento, características, limitaciones, etc. de los sistemas de ficheros paralelos.
- Profundizado en las posibilidades que ofrece MPI.
- Aprendido una metodología de evaluación de sistemas (*benchmarking*).

9.2 Planificación

En esta sección se describe la metodología utilizada en el proyecto, así como la organización y planificación de las tareas que se van a llevar a cabo; además, se especifican los recursos humanos y materiales necesarios para la realización del proyecto. Para finalizar, se ofrece un presupuesto que refleja el coste total y de mercado que supondría el desarrollo del proyecto.

9.2.1 Metodología de trabajo

Para el desarrollo del proyecto se ha utilizado el ciclo de vida incremental o de versiones sucesivas. La elección de este ciclo de vida se debe además de porque el proyecto es de gran tamaño, porque algunas de las funcionalidades del proyecto deben estar operativas antes que otras y además, este método permite validar el sistema a medida que se construye.

Con este método, cada versión o fase es un sistema funcional capaz de realizar progresivamente la función deseada.

Como se puede observar en la Figura 25, el ciclo de vida incremental posee una fase de análisis inicial y varias fases de diseño, implementación y pruebas de forma incremental. Estas tres fases incrementales convergerán en una fase final de evaluación del sistema. Además, durante todo el proyecto y de forma transversal al resto de las fases, será necesaria una fase o tarea de documentación.

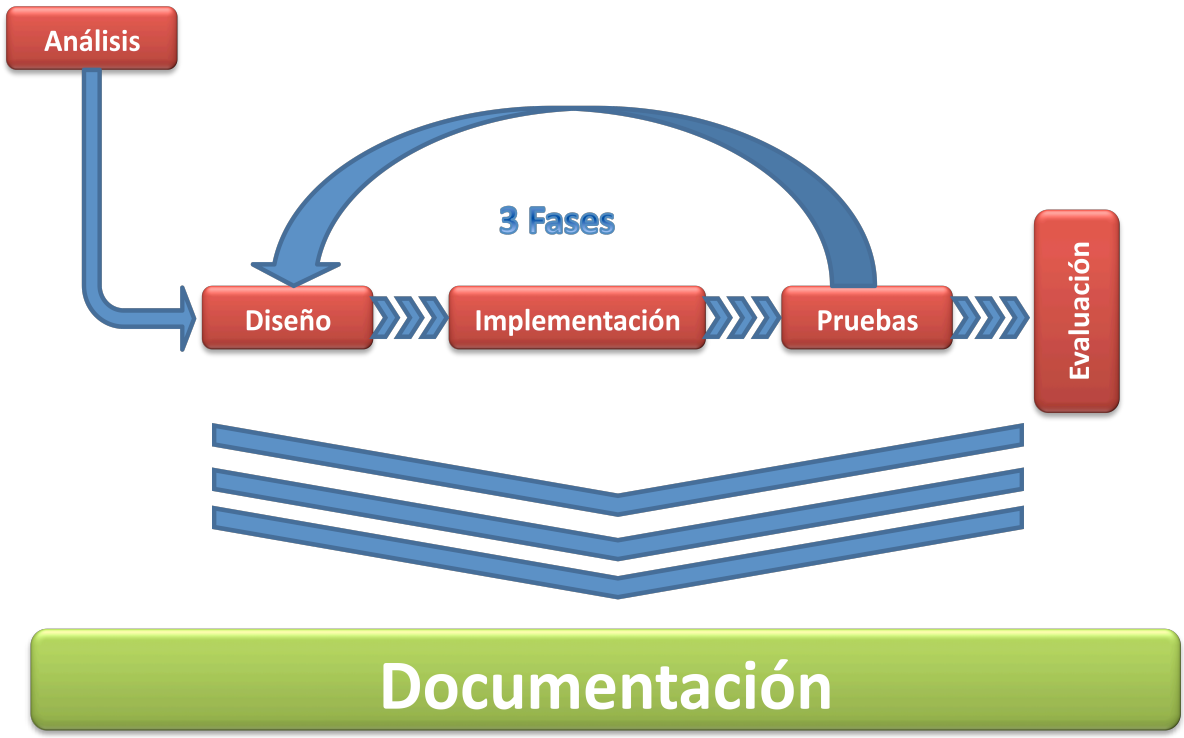

Figura 25: Ciclo de vida incremental

A continuación se resumen los objetivos de cada una de las fases del diseño incremental.

Análisis:

Se realiza un estudio de las posibilidades que ofrece el sistema desarrollado dentro del ámbito actual, y se comprueba la disponibilidad tecnológica que existe en la actualidad.

Fase 1:

Una vez realizado un estudio previo, se lleva a cabo el proceso de despliegue del sistema. En esta fase se determinan los componentes necesarios para la creación del sistema final: recursos *hardware*, sistemas operativos, recursos *software*, etc.

Fase 2:

Construcción básica del esqueleto del sistema que dará soporte al desarrollo de todos los requisitos planteados en este proyecto.

Fase 3:

Diseño, implementación y pruebas de todos los requisitos restantes planteados en este proyecto.

Evaluación:

Una vez que finalice el desarrollo del nuevo sistema, será necesaria esta fase de evaluación que englobe todas las fases incrementales anteriormente descritas, proporcionando una visión global del rendimiento del sistema.

Documentación:

Esta fase de documentación estará presente desde el comienzo hasta la finalización del proyecto. Como parece lógico pensar, ya que existen tres fases incrementales, la documentación de las mismas será también incremental completámdose finalmente al término del proyecto.

9.2.2 Presupuesto

En este apartado se realiza el desglose del coste asociado que tiene la realización del proyecto.

Gastos de personal imputables al proyecto

Para realizar el cálculo del presupuesto se han tenido en cuenta las siguientes consideraciones:

Jornada laboral de 5 horas.

5 días laborables a la semana (excluidos fines de semana y festivos).

Las vacaciones y días festivos considerados en la planificación son:

8 de Diciembre 2009: Inmaculada Concepción.

25 de Diciembre 2009: Navidad.

1 de Enero 2010: Año Nuevo.

6 de Enero 2010: Reyes.

19 de Marzo 2010: San José.

1 y 2 de Abril jueves y viernes Santo.

1 de Mayo: Día del trabajador.

Periodo de realización:

Inicio del proyecto: 20 de Octubre de 2009.

Finalización del proyecto: 7 de Abril de 2010

El proyecto requiere un analista y un programador.

Coste de personal por hora de trabajo:

Analista: 52,00 €/hora.

Analista-programador: 33,00 €/hora.

Por lo tanto, el cómputo total de horas dedicadas al proyecto es 865 horas, por lo que el proyecto ha tenido una duración total de 173 días reales.

Actividades	Duración (horas)	Recursos	Total (Euros)
Análisis (Estudio preliminar)	95	Analista	4.940,00 €
Fase 1 (Despliegue del sistema)	20	Analista-Programador	660,00 €
Fase 2 (Estructura básica del sistema)	250	Analista-Programador	8.250,00 €
Fase 3 (Completitud del sistema)	210	Analista-Programador	6.930,00 €
Evaluación del sistema	200	Analista	10.400,00 €
Documentación	90	Analista	4.680,00 €
Total	865 horas / 173 días		35.860,00 €

Tabla 42: Especificación de actividades y coste

Coste del *hardware* para el desarrollo del sistema:

El despliegue del sistema ha sido realizado con el siguiente *hardware*:

Equipo servidor:

Pentium IV de 1,8 Ghz de velocidad y 3 GBytes de memoria. 2 discos duros de 1 TBytes.

El coste de este servidor es nulo ya que se prestó para la realización de este proyecto.

Equipo portátil Sony VAIO VGN-FE31Z. El coste de este servidor es nulo ya que se prestó para la realización de este proyecto.

Cluster Linux cedido por la Universidad Carlos III (ARCOS), el cual está compuesto por sus:

- Sistema operativo Debian Lenny.
- Quad core Intel(R) Xeon(R) CPU E5405 @ 2.00GHz.
- Memoria RAM 4 Gbytes.
- Red Ethernet de 1 GBytes/segundo.

Coste de licencias para el desarrollo del sistema:

Puesto que uno de los requisitos de usuario era la utiliz3ación de Linux como sistema operativo, esto conlleva un coste nulo en cuanto licencias de los sistemas operativos utilizados en cada uno de los servidores o computadores. Además, todas las herramientas utilizadas para el desarrollo de este proyecto fueron también de *software* libre eliminando los costes que pudieran derivarse de las mismas.

Coste de consumibles:

Se estima un coste de 200 € para el material fungible necesario en el desarrollo de este proyecto, como puede ser: papel, DVDs en blanco, acceso a Internet en momentos puntuales y cableado Ethernet.

Costes indirectos:

Se estima en un 20% los costes indirectos atribuidos a los costes de instalación del SW necesario en el cluster de la Universidad y en el sistema de desarrollo, desplazamientos a la Universidad para la investigación en los laboratorios y el resto de tareas generales necesarias para la consecución de este proyecto.

Coste total del proyecto:

Atendiendo a los supuestos anteriores, el presupuesto inicial del proyecto será:

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor:
Javier Escobar Guisado
2.- Departamento:
Informáti ca
3.- Descripción del Proyecto:

- Titulo	**Diseño e Implementación de un Sistema de Ficheros en MPI sobre FUSE**
- Duración (meses)	**8,65**
Tasa de costes Indirectos:	**20%**

4.- Presupuesto total del Proyecto (valores en Euros):
Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a ti tulo informati vo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Escobar Guisado, Javier	DNI_1	Analista	4,8	3.300,00	15.840,00	
Escobar Guisado, Javier	DNI_1	Analista-programad	3,85	5.200,00	20.020,00	
					0,00	
					0,00	
					0,00	
		Hombres mes	**8,65**	**Total**	35.860,00	

[a] 1 Hombre mes = (100)131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciació n	Coste imputable [d]
		100		60	0,00
		100		60	0,00
		100		60	0,00
		100		60	0,00
		100		60	0,00
					0,00
				Total	0,00

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Consumibles (DVDs, internet,etc	Varias	200,00
	Total	200,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas,

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	35.860
Amorti zación	0
Subcontratación de tareas	0
Costes de funcionamiento	200
Costes Indirectos	7.212
Total	43.272

Figura 26: Presupuesto inicial atendiendo a la plantilla de Rúbrica

La Tabla 43detalla el coste total del proyecto:

Concepto	Importe (€)
Recursos Humanos	**35.860,00 €**
HW y Licencias SW	**0,00 €**
Consumibles	**200,00 €**
20 % Costes indirectos	**7.212,00 €**
Total presupuesto inicial	**43.272,00 €**
Riesgo (10%)	4.327,20 €
Total antes de beneficio	**47.599,20 €**
Beneficio (15%)	7.139,88 €
Total sin IVA	**54.739,08 €**
16%IVA	8.758,25 €
Total con IVA	**63.497,33 €**

Tabla 43: Coste total del proyecto

El presupuesto total de este proyecto se estima en sesenta y tres mil cuatrocientos noventa y siete cin treinta y tres céntimos **(63.497,33 €)**[1]**.**

Madrid a 5 de octubre de 2009

El jefe de proyecto

Fdo: Javier Escobar Guisado.

[1] Este presupuesto está orientado a un documento de coste interno y no se debe mostrar al cliente final.

9.2.3 Planificación

En la Figura 27 se muestra la planificación asignada a este proyecto con los requisitos de calendario expuestos en la Sección 9.2.2.

Al comienzo de cada una de las fases o tareas principales, se ha dispuesto un pequeño número de días que servirán tanto de holgura, como para distribuir los tiempos necesarios para la tarea de documentación del proyecto.

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**PFC_JEG**	**195,2 días**	**mar 20/10/09**	**lun 19/04/10**
2	Análisis	19 días	mar 20/10/09	jue 05/11/09
3	**Fase 1: Despliegue del sistema**	**4 días**	**jue 12/11/09**	**mar 17/11/09**
4	Diseño	1 día	jue 12/11/09	vie 13/11/09
5	Implementación	2 días	vie 13/11/09	lun 16/11/09
6	Pruebas	1 día	lun 16/11/09	mar 17/11/09
7	**Fase 2: Estructura básica del sistema**	**50 días**	**mar 24/11/09**	**mié 13/01/10**
8	Diseño	6 días	mar 24/11/09	lun 30/11/09
9	Implementación	40 días	lun 30/11/09	vie 08/01/10
10	Pruebas	4 días	vie 08/01/10	mié 13/01/10
11	**Fase 3: Completitud del sistema**	**42 días**	**mar 19/01/10**	**jue 25/02/10**
12	Diseño	5 días	mar 19/01/10	vie 22/01/10
13	Implementación	33 días	lun 25/01/10	lun 22/02/10
14	Pruebas	4 días	lun 22/02/10	jue 25/02/10
15	Evaluación Final	40 días	vie 05/03/10	mié 14/04/10
16	Documentación	18 días	mar 20/10/09	jue 05/11/09
17	Defensa del proyecto	0 días	lun 19/04/10	lun 19/04/10

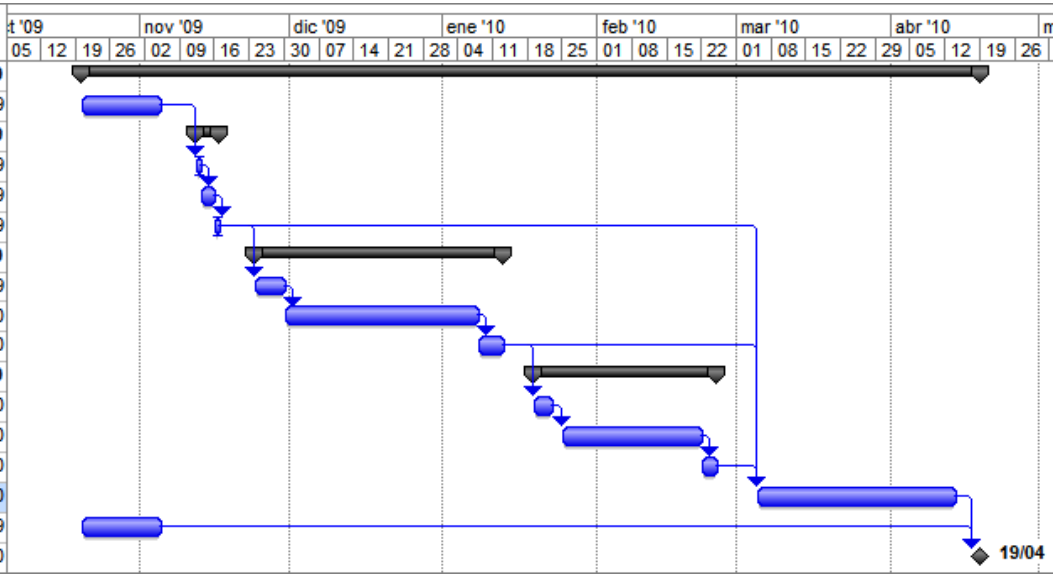

Figura 27: Planificación del proyecto

9.3 Trabajos futuros

Para ampliar las posibilidades que puede ofrecer este Proyecto Fin de Carrera, se proponen las siguientes tareas:

Se propone modificar el sistema propuesto para que de soporte a tolerancia a fallos. Actualmente los supercomputadores cuentan con cientos de miles de nodos, por lo tanto, la probabilidad de fallo aumenta progresivamente. La mejora consistiría en dotar a AHPIOS de un sistema de tolerancia a fallos, el cual garantice la integridad de los datos, incluso cuando las aplicaciones o los servidores de E/S fallen.

Integrar servicios web dentro de la aplicación para permitir trabajar en entornos como Grid Computing o Cloud Computing. Este trabajo podría permitir realizar servicios de cómputo distribuido entre recursos de *hardware* heterogéneos comunicados entre sí mediantes redes de área extensa como Internet. La gestión de los metadatos podría mantenerse tanto local como remotamente.

Un trabajo anterior a este Proyecto Fin de Carrera, tuvo como objetivo integrar AHPIOS en plataformas Windows. El resultado de este proyecto consistió en la migración del sistema de E/S paralela AHPIOS a este tipo de plataformas. La carencia que presenta el sistema portado es que, al igual que la versión preliminar de AHPIOS, ofrece únicamente acceso a los datos mediante la librería MPI-IO. Se propone como línea de trabajo futura, el diseño e implementación de un controlador para las plataformas Windows.

Realizar un mayor número de pruebas y evaluaciones. Entre ellas, ejecutar aplicaciones implementadas en distintos lenguajes de programación, como puede ser Fortran. Fortran es un lenguaje de programación muy extendido dentro de la comunidad científica. Finalmente, se propone evaluar el sistema con una carga de trabajo real, en un entorno real.

Implementación de enlaces simbólicos que permita usar este tipo de enlaces dentro de FUSE.

Durante la evaluación del sistema creado se ha utilizado PVFS2 (2.8). Esta versión de PVF2 permite utilizar más de un servidor de datos y más de un servidor de metadatos. Sin embargo, todas las pruebas se realizaron siempre con un solo servidor de metadatos. Por tanto, se plantea como mejora futura para este trabajo el soporte para la utilización en PVFS2 de más de un servidor de metadatos.

Bibliografía

A continuación se exponen los recursos y la documentación que se han consultado y estudiado para el análisis y desarrollo del proyecto. Estas referencias aparecen a lo largo de todo el documento con su correspondiente número entre corchetes.

[1] Proyecto FUSE: http://fuse.sourceforge.net/	16 enero 2010
[2] Ley de Amdahl: http://cs.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html	05 febrero 2010
[3] Proyecto ARPANET: http://www.iso.port.ac.uk/~mike/docs/internet/internet/node4.html	05 febrero 2010
[4] Protocolo TCP/IP: http://www.yale.edu/pclt/COMM/TCPIP.HTM	05 febrero 2010

[5] BSD (Berkeley) Unix: http://www.freebsd.org/doc/es_ES.ISO8859-1/articles/explaining-bsd/article.html	05 febrero 2010
[6] ARCNET: http://www.arcnet.com/abtarc.htm	05 febrero 2010
[7] Xerox PARC: http://www.parc.com/about/milestones.html	05 febrero 2010
[8] Kronenberg, N. P., Levy, H. M., and Strecker, W. D. 1985. VAXclusters (extended abstract): a closely-coupled distributed system. SIGOPS Oper. Syst. Rev. 19, 5 (Dec. 1985), 1. DOI= http://doi.acm.org/10.1145/323627.323628	06 febrero 2010
[9] Tandem Himalaya: http://www.redbarnhpc.com/v2/index.php/cluster-computing/the-history-of-cluster-computing	06 febrero 2010
[10] IBM S/390 Parallel Sysplex: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=508100	06 febrero 2010
[11] Parallel Virtual Machine (PVM): http://www.csm.ornl.gov/pvm/	07 febrero 2010
[12] Arquitectura Beowulf: http://www.beowulf.org/overview/index.html	07 febrero 2010
[13] RAID: http://www.technick.net/public/code/cp_dpage.php?aiocp_dp=guide_raid	07 febrero 2010
[14] GPFS: http://publib.boulder.ibm.com/epubs/pdf/a7604132.pdf	07 febrero 2010
[15] LUSTRE: http://www.oracle.com/us/products/servers-storage/storage/storage-software/031855.htm	09 febrero 2010
[16] PVFS: http://www.pvfs.org/	09 febrero 2010
[17] *MPI Message Passing Interface Forum: http://www.mpi-forum.org/*	09 febrero 2010
[18] MPI-IO: http://www.nersc.gov/nusers/resources/software/libs/io/mpiio.php	13 febrero 2010
[19] ROMIO: http://www.mcs.anl.gov/research/projects/romio/	13 febrero 2010
[20] An abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces (6th Symposium on the Frontiers of Massively Parallel Computation, October 1996, pp 180-187, 1996): http://www.mcs.anl.gov/~thakur/papers/adio.pdf	13 febrero 2010

[21] MPICH: http://www.mcs.anl.gov/research/projects/mpich2/	13 febrero 2010
[22] Performance Evaluation of Remote Memory Access (RMA) Programming on Share Memory Parallel Computers: http://www.nas.nasa.gov/News/Techreports/2003/PDF/nas-03-001.pdf	14 abril 2009
[23] A performance Study of Two-Phase-I/O: http://www.umcs.maine.edu/~dickens/pubs/292.21.ps	21 febrero 2010
[24] *Benchmark*: http://es.tldp.org/COMO-INSFLUG/es/pdf/Benchmarking-COMO.pdf	27 febrero 2010
[25] Tutorial FUSE: http://www.buanzo.com.ar/lin/FUSE.html	01 marzo 2010
[26] VFS: http://es.tldp.org/Articulos-periodisticos/jantonio/vfs/vfs_1.html	01 marzo 2010
[27] EXT3: http://kataix.umag.cl/~ruribe/Utilidades/ext3.pdf	02 marzo 2010
[28] AVFS: http://avf.sourceforge.net/	02 marzo 2010
[29] FreeBSD: http://www.freebsd.org/	02 marzo 2010
[30] NetBSD Home page: http://www.netbsd.org	02 marzo 2010
[31] OpenSolaris: http://hub.opensolaris.org/bin/view/Main	02 marzo 2010
[32] MAC OS X Home page: http://www.apple.com/es/macosx/	02 marzo 2010
[33] *Benchmark* NFSSTONES: http://www.filewatcher.com/m/nfsstone.tar.gz.3525.0.0.html http://www.gelato.unsw.edu.au/IA64wiki/NFSSTONE	28 marzo 2010
[34] *Benchmark* IPBench: http://sourceforge.net/projects/ipbench/	28 marzo 2010
[35] *Benchmark* NHFSSTONES: http://linuxcommand.org/man_pages/nhfsstone8.html	28 marzo 2010
[36] *Benchmark* BTIO: http://www.nas.nasa.gov/News/Techreports/2003/PDF/nas-03-002.pdf http://www.springerlink.com/content/x67210045210/?p=7910762b6b62460c82c6bc7594b5860b&pi=0	29 marzo 2010

[37] NASA Advance Supercomputing Division: http://www.nas.nasa.gov	29 marzo 2010
[38] NPB-MPI versión 2.4 E/S: http://www.nas.nasa.gov/Resources/Software/npb.html	29 marzo 2010
[39] NTFS: http://www.ntfs.com/	05 mayo 2010
[40] POSIX: http://standards.ieee.org/regauth/posix/index.html	10 mayo 2010
[41] DB Berkeley: http://www.oracle.com/global/lad/database/berkeley-db/index.html	06 junio 2010
[42] Top 500: http://www.top500.org/	06 junio 2010

Apéndice I: Instalación y configuración de FUSE

Instalación

Algunos proyectos incluyen el paquete de FUSE al completo (para simplificar la instalación). En otros casos, o incluso sólo para intentar ejecutar el ejemplo de fuse, tiene que ser instalado previamente.

Para desarrollar un sistema de ficheros utilizando FUSE, lo primero que debemos hacer es obtener el código fuente de FUSE y descomprimir lo mediante la utilización del comando ”tar –zxvf fuse-2.8.3.tar.gz”. Esto creará un directorio FUSE con el código fuente. El contenido de dicho directorio será:

- **./doc**: contiene la documentación relativa a FUSE.
- **./kernel**: contiene el código fuente del modulo del *kernel* de FUSE y que no suele ser necesario modificar para la creación de un nuevo sistema de ficheros.

- **./include**: contiene las cabeceras de la API de FUSE, necesarias para crear un sistema de ficheros. La única que siempre será necesaria está localizada en el fichero "*fuse.h*".
- **./lib:** en este directorio se encuentra el código fuente necesario para la creación de las librerías de FUSE, las cuales serán enlazadas con los ficheros binarios para crear el nuevo sistema de ficheros.
- **./util**: guarda el código fuente de la librería de utilidades de FUSE.
- **./example**: obviamente esta carpeta contendrá algunos ejemplos que podrán servir de referencia. En especial cabe resaltar los sistemas de ficheros implementados en "fusexmp" y "hello".

La instalación, después de descomprimir el paquete, es simple:

> *./configure*

> *make*

> *make install*

Cuando se ejecute el último comando "*make install*", hay que asegurarse que se poseen permisos de *root*, ya que el programa "*fusermount*" debe contener su "*set-user-id*" para *root* y de esta forma permitir a los usuarios normales montar sus propias implementaciones de sus sistemas de ficheros. Aún de esta forma, existen algunas limitaciones para prevenir acciones maliciosas de los usuarios:

1. un usuario sólo puede montar su sistema de ficheros sobre un punto de montaje sobre el cual tenga permisos de escritura.
2. El punto de montaje debe ser propiedad del usuario y no contener activado el byte *sticky* como normalmente ocurre con el directorio "/tmp".
3. Ningún otro usuario incluido *root,* puede acceder al contenido el sistema de ficheros montado.

Si los comandos anteriores produjesen algún error, será necesario investigar la procedencia del mismo, recomendando como punto de partida las FAQ existentes para

este proyecto en sourceforge [3] o incluso alguno de los excelentes tutoriales que circulan por la red [25].

El script "*configure*" intentará averiguar la localización de la fuente del *kernel*. Caso que esto falle, debería ser especificado usando el parámetro –with-kernel. Construir el módulo del *kernel* necesita un árbol del fuente del *kernel* configurado que coincida con el *kernel* en ejecución. Si construye su propio *kernel* esto no es un problema. De todas formas si se usa un *kernel* precompilado, las cabeceras del *kernel* usadas por los procesos de construcción de FUSE deben ser preparadas con antelación. Existen dos posibilidades:

1. Un paquete que contenga las cabeceras del *kernel* para los binarios del *kernel* se encuentra disponible en la distribución (por ejemplo, en Debian son las cabeceras del *kernel* X.Y.Z paquete para la imagen del *kernel* X.Y.Z)

2. Los fuentes del *kernel* tienen que ser preparados:

   - Extrae los fuentes del *kernel* en algún directorio o carpeta
   - Copia la configuración del *kernel* en ejecución (normalmente se puede encontrar en /boot/config-X.Y.Z) en el fichero .config en lo alto del árbol de directorios del fuente
   - Ejecutar "make prepare"

Configuración

Existen algunas opciones referentes a la política de montaje que pueden ser configuradas dentro del fichero "/etc/fuse.conf":

- **mount_max=NNN:** establece el número máximo de montajes permitidos de FUSE para usuarios que no sean *root*. Por defecto contendrá el valor 1000.
- **User_allow_other:** permite a los usuarios distintos de *root* especificar las opciones de montaje "*allow_other*" y "*allow_root*".

Opciones de montaje:

La mayoría de las opciones de montaje generales descritas en "*man mount*" están soportadas (*ro, rw, suid, nosuid, dev, nodev, exec, noexec, atime, noatime, sync async, dirsync*). Los sistemas de ficheros son montados por defecto con las opciones *"-onodev, nosuid*", las cuales solamente pueden ser anuladas por un usuario con privilegios adecuados.

Hay algunas opciones de montaje específicas de FUSE que pueden ser establecidas para todos los sistemas de ficheros:

- **default_permissions:** por defecto FUSE no chequea los permisos de acceso, los sistemas de ficheros son libres de implementar sus políticas de acceso o dejárselo al mecanismo subyacente de acceso de ficheros (como podría ser en el caso de un sistema de ficheros de red). Esta opción habilita el permiso de chequeo, restringiendo el acceso basado en el modo de ficheros. Esta opción es normalmente de mucha ayuda junto con la opción de montaje "*allow_other*".
- **allow_other:** esta opción anula las medidas de seguridad restrictivas de acceso de ficheros para el usuario montando el sistema de ficheros. Así, todos los usuarios (incluyendo *root*) pueden acceder a los ficheros. Esta opción es por defecto y permitirá solamente a *root*, pero esta restricción puede ser eliminada con una opción de configuración descrita en la sección anterior.
- **allow_root:** esta opción es similar a "*allow_other*" pero el acceso de ficheros está limitado al usuario que montó el sistema de ficheros y a *root*. Esta opción y "*allow_other*" son mutuamente excluyentes.
- **kernel_cache:** esta opción deshabita la limpieza de cache de los contenidos del fichero en cada operación de apertura (*open ()*). Esto debería ser habilitado solamente en sistemas de ficheros, donde los datos de fichero no sean cambiados externamente nunca (no a través de los sistemas de ficheros FUSE montados. Esto no es recomendable para sistemas de ficheros entre y para otros sistemas de ficheros

"intermediarios". Si esta opción no se específica (ni la opción "direct_io"), los datos permanecerán en la cache tras la operación "*open()*" y una llamada de sistema "*write ()"* no provocará siempre una operación de lectura.

- **auto_cache:** esta opción habilita automáticamente la limpieza de los datos en la cache con una operación "*open ()*". La cache será solamente limpiada si el tiempo de modificación o el tamaño del fichero ha cambiado.
- **large_read:** esto puede mejorar el rendimiento para algunos sistemas de ficheros, aunque también podría llegar a degradarlo. Esta opción es solamente útil sobre *kernels* 2.4.x, ya que para *kernels* 2.6 estas peticiones extensas de lectura de datos son automáticamente optimizadas para un mejor rendimiento.
- **direct_io:** esta opción deshabita el uso de páginas de cache (cache de contenido de ficheros) en el *kernel* para este sistema de ficheros. Esto tiene bastantes afecciones:

    - cada llamada al sistema de "*read()*" "*write()*" iniciará una o más operaciones de lectura o escritura, los datos no serán cacheados en el *kernel*.
    - Los valores de retorno de las llamadas al sistema de "*read()*" y "*write()*" corresponderán a los valores de retorno de las operaciones de lectura y escritura. Esto es útil por ejemplo si el tamaño del fichero no se conoce por adelantado (antes de leerlo).

- **max_read=N:** con esta opción se puede establecer el máximo tamaño de las operaciones de lectura. Por defecto es infinito. Note que el tamaño de las peticiones de lectura está limitado de cualquier forma a 32 páginas (lo cual es 128 kbytes sobre i386).
- **max_readahead=N:** establece el número máximo de bytes de lectura adelantada. Por defecto está determinado por el kernel. En Linux 2.6.22 o anteriores es 131.072 (128 Kbytes).

- **max_write=N:** establece el número máximo de bytes en una operación sencilla de escritura. Por defecto son 128 kbytes. Note, que debido a varias limitaciones, el tamaño de las peticiones de escritura puede ser mucho menor (4 Kbytes). Esta limitación será eliminada del futuro.
- **async_read:** realiza lecturas a cinco. Esta opción es la de por defecto.
- **sync_read:** realiza todas las lecturas (incluidas las lecturas adelantadas) de forma síncrona.
- **hard_remove:** el comportamiento por defecto es que si una operación de lectura de fichero es borrada, el fichero es renombrado a un fichero oculto y solamente eliminado cuando el fichero es finalmente liberado. Esto alivia a la implementación del sistema de ficheros de tener que lidiar con este tipo de problemas. Esta opción deshabita el comportamiento de ocultación, y los ficheros son eliminados inmediatamente en una operación *unlink* (o con una operación de renombrado que sobre-escribe un fichero existente). Esta opción no es muy recomendable. Cuando se activa esta opción, las siguientes funciones de "libc" fallan sobre los ficheros no enlazados (devolviendo "*errno*" de ENOENT): *read(), write(), fsunc(), close(), f*attr(), ftruncate(), fstat(), fchmod() y fchown()*.
- **debug:** activa la impresión de la información de depuración por la librería.
- **fsname=NAME:** establece los fuentes del sistema de ficheros (primer campo en "*/etc/mtab*"). Por defecto es el nombre del programa.
- **subtype=TYPE:** establece el tipo del sistema de ficheros (tercer campo en"*/etc/mtab*"). Por defecto es el nombre del programa. Si el *kernel* lo soporta, "*/etc/mtab*" y "*/proc/mounts*" mostrarán el tipo del sistema de ficheros como "*fuse.TYPE*". Si el *kernel* no soporta subtipos, el campo fuente será "*TYPE#NAME*", o si la opción *fsname* no está especificada, será simplemente "*TYPE*".
- **use_ino:** respeta el campo "*st_ino*" en *getattr()* y *fill_dir()*. Este valor es utilizado para rellenar el campo "*st_in*" en las llamadas a las funciones *stat(), lstat()* y *fstat()* y el campo "*d_ino*" en la función *readdir()*. El

sistema de ficheros no tiene que garantizar la unicidad sin embargo, algunas aplicaciones confían en este valor siendo único para todo el sistema de ficheros.

- **readdir_ino:** si no se proporciona la opción "*use_ino*", intentar rellenar el campo "*d_ino*" en *readdir()*. Si el nombre fue previamente buscado y todavía está en cache, el número de *inode* encontrado será el usado. De lo contrario será establecido a -1. Si la opción "*use_ino*" es utilizada, entonces será ignorada.
- **nonempty:** permite montar el sistema de ficheros sobre un fichero o directorio no vacío. Por defecto estos montajes son rechazados (desde la versión 2.3.1) para prevenir re-escrituras accidentales, las cuales pueden prevenir copias de seguridad automáticas.
- **umask=M:** anula los byte de permisos establecidos por sistema de ficheros. Los bits de permisos resultantes sólo desaparecidos del valor dado de *unmask*. El valor es proporcionado en formato octal.
- **uid=N:** anula el campo "*st_uid*" establecido por el sistema de ficheros.
- **gid=N:** anula el campo "*st_gid*" establecido por el sistema de ficheros.
- **blkdev:** monta un sistema de ficheros respaldado por un dispositivo bloqueado. Esto es una opción privilegiada. El dispositivo debe estar especificado con la opción "*fsname=NAME*".
- **entry_timeout=T:** tiempo de espera en segundos que se cachearán las búsquedas de nombres. Por defecto es 1.0 segundos. Para todas las opciones de tiempo de espera, es posible proporcionarlas con fracciones de segundos (por ejemplo entry_timeout=2.8).
- **negative_timeout=T:** tiempo de espera en segundos para el que se guardará una búsqueda negativa en la cache. Esto significa, que si el fichero no existía (la búsqueda devolvió ENOENT), la búsqueda sólo se realizará después del tiempo de espera y se entenderá que el fichero o directorio no existirá hasta entonces. Por defecto es 0.0 segundos, lo que significa que la búsqueda negativa está deshabitada.

- **attr_timeout=T:** tiempo de espera en segundos para el que los atributos de un fichero o directorio se guardarán en la cache. Por defecto es 1.0 segundo.
- **ac_attr_timeout=T:** tiempo de espera segundos con el que los atributos serán guardados en la cache para el propósito de chequeo si "*auto_cache*" se deberían limpiar los datos del fichero con la apertura del mismo. Por defecto es el valor de "*attr_timeout*".
- **intr:** permite las peticiones sean interrumpidas. Activar esta opción se podría experimentar un comportamiento no adecuado caso que el sistema de ficheros no soporte la interrupción de peticiones.
- **Intr._signal=NUM:** específica el número de señal a enviar al sistema de ficheros cuando una petición es interrumpida. Por defecto es 10 (USR1).
- **modules=M1[:M2…]:** añade módulos a la pila del sistema de ficheros. Los módulos son apilados en el orden especificado, con el sistema de ficheros original colocado al fondo de la pila.

Apéndice II: Configuración de NFSSTONE

Debido a que a diferencia del resto de *benchmark* utilizados en este Proyecto Fin de Carrera, NFSTONES proporciona un número de parámetros de configuración bastante extenso. Por ello, se ha considerado necesario incluir en este apéndice, una descripción detalla de todos los parámetros necesarios para configurar la ejecución de esta aplicación.

Este *benchmark* es algo antiguo y con el paso del tiempo ha sufrido múltiples modificaciones y adaptaciones como en el caso de *IPBench* [34]. En esta versión, se portó *NFSSTONES* para permitir a múltiples clientes la sincronización dirigiendo todas las peticiones de un *benchmark* a uno o a más objetivos, produciendo los resultados en un nodo controlado. Esto proporciona varias ventajas como:

- Sincronización del *benchmark* al comienzo y a su finalización.
- Un nodo controlado que inicializa el *benchmark* y recolecta los resultados, esto elimina la necesidad de un *login* individual para un cliente y varias máquinas destino donde se ejecutarían los test.

- Reduce el tráfico de red y la ejecución de procesos en los clientes y en las máquinas destino, debido a la habilidad de ejecutar los test desde una máquina separada y controlada.

Los principales cambios realizados a la hora de portar *NFSSTONES* a *IPBench* fueron:

- Entrega a la salida una media y una desviación estándar (n-1) de todos los clientes del *benchmark*.
- Permite acumular el trabajo para ser realizado (*TOPDIRS, BOTDIRS*), especificándolo en la línea de comandos, tomando como valor por defecto la configuración original.
- Distribuye el *benchmark* en varias rutinas en línea con el interfaz IPBench.

Actualmente, *NFSSTONES* ha sido reemplazado por *NHFSSTONE* (utilizado como comando Linux para generar una carga artificial con una mezcla particular de operaciones NFS).

Se considera ***NFSSTONEs*** el número total de operaciones de fichero que son realizadas por este *benchmark*. Estas operaciones son ajustables en tiempo de compilación a través de valores '*#define'* en el código fuente. Los principales factores para *NFSSTONE* son el número total de directorios para una ejecución sencilla (*TOT_DIRS*) y el número de ficheros por directorio (*FILES_PER_DIR*).

- *TOT_DIRS*: es la suma de los valores *TOP_DIRS* y *BOT_DIRS*.
  - *TOP_DIRS*: define el número de veces que se tienen que realizar todos los test.
  - *BOT_DIRS*: define el número de veces que se tienen que realizar las operaciones de lectura y escritura para cada *TOP_DIRS*.

- Otras opciones configurables en tiempo de compilación son:
  - *FILE_CREATES*: el número de veces que un fichero es creado por cada test *BOT_DIRS*.

- *FILE_PER_DIR*: el número de ficheros por cada directorio.
- *BLOCKS_PER_FILE*: número de bloques por fichero.
- *FILE_LOOKUPS*: número de operaciones de búsqueda (*LOOKUPS*) de ficheros.
- *BYTES_PER_BLOCK*: número de bytes por bloque.
- *NREAD_LINK*: número de enlaces simbólicos creados.

Con la configuración de los valores por defecto, cada cliente utilizará alrededor de 12,2 MB de espacio en disco del servidor. Es muy importante mantener el espacio de disco para los clientes por encima de esta cantidad de memoria. Puesto que se intenta normalmente comprobar la velocidad del servidor y no la velocidad de la cache del cliente, los bloques de ficheros deberán ajustarse para mantener la cache rebosante.

La compilación del código fuente se realizará mediante el siguiente comando:

gcc nfsstone.c -o nfsstone

Tras compilar el código fuente, podrá ser ejecutado mediante el siguiente comando:

./nfsstone <path el sistema de ficheros de test> <fichero cerrojo>

Resultados obtenidos

Existen muchas combinaciones de diferentes posibilidades de configuración de este *benchmark.* El sistema de ficheros que retorna tras exportar a un sistema de ficheros en red, como por ejemplo exportando un NFS devuelto por un sistema de ficheros Ext3, introduciendo de esta forma un gasto de tiempo extra en el sistema de ficheros y produciendo un impacto directo los resultados.