UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

FPFS: a parallel file system
based on OSD+ devices

FPFS: un sistema de ficheros paralelo
basado en dispositivos OSD+

Dña. Ana Avilés González

2014

UNIVERSIDAD DE MURCIA
Facultad de Informática
Departamento de Ingeniería y Tecnología de Computadores

FPFS: a parallel file system based on OSD+ devices

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy

By
Ana Avilés González

Advised by
Juan Piernas Cánovas
María Pilar González Férez

Murcia, September 2014

Abstract

The growth in the number of nodes and computational power of supercomputers allow us to process not only larger and more compute-intensive workloads, but also data-intensive workloads that access and store vast amounts of data. Traditionally, to provide fast and reliable performance, parallel file systems have focused on scalable data distribution to share out the storage and management of large data volumes between several nodes. However, increasing workload is now pushing metadata servers to their limits, with not only greater volumes of data but also critically ever-greater numbers of metadata requests. Metadata operations are very CPU consuming, and a single metadata server is often no longer sufficient.

In this thesis we define a new parallel file system architecture that provides a completely distributed metadata service. We believe this is key as we approach the exabyte scale-era in parallel file systems. The main contributions to reach this aim are the following.

First, we introduce the Fusion Parallel File System (FPFS) that presents a novel architecture that merges the metadata cluster and the data cluster into one large cluster through the use of a single type of server. This enlarges the capacity of the metadata cluster because it becomes as large as the data cluster, also augmenting the overall system capacity.

Second, we propose OSD+ devices to support the FPFS architecture. Object-based Storage Devices (OSDs) have proved to be a perfect tool for parallel file systems, since their inherent intelligence allows file systems to delegate tasks to the devices themselves, such as the management of the device space. OSD+s are improved object-based storage devices that, in addition to handle data objects, as traditional OSDs do, can also manage *directory objects*, a new type of object able to store and handle metadata.

Third, we develop a scalable metadata cluster made up of hundreds to thousands of OSD+ devices. The design takes into account essential aspects, like a balanced namespace distribution, able to adapt to cluster changes, and the atomicity of operations that involve serveral OSD+s.

Fourth, we enlarge the metadata service capabilities by including support for huge directories. Directories storing thousand to millions of files accessed by thousands of clients at the same time are becoming common in file systems, and present performance downgrades if not handled properly. We modify OSD+ devices to dynamically distribute these directories among several objects in different OSD+s in the cluster.

Finally, we propose the use of batch operations, which embed several operations of the same type into a single network packet. This way, we optimize network resources, and significantly reduce network traffic on the system, reducing network delays and round-trips.

Experiments show that FPFS can create, stat and unlink hundreds of thousands of empty files per second with a few OSD+s. The rates obtained by FPFS are an order of magnitude better than what Lustre achieves when a single metadata server is used, and what OrangeFS and Ceph achieve when a metadata cluster is deployed.

«*Solo no puedes, con amigos sí*»
La bola de cristal

Agradecimientos

A lo largo del tiempo se van teniendo, según cada cual, ideas, intuiciones o teorías sobre multitud de cosas. Dentro y fuera del ámbito científico el procedimiento a seguir suele ser similar: experimentos o experiencias que con el tiempo terminan por confirmar si tenían fuste, o si las teorías eran más o menos acertadas. A día de hoy, termino este ciclo con más dudas de las que comencé, y por ahora no sé si eso es bueno o malo. Lo que sí parece haberse confirmado es aquello de *solo no puedes, con amigos sí*.

Primero, agradezco a mis padres apoyo, comprensión, y sobretodo, a diferencia de lo que suele suceder, no ser cansinos con determinadas preguntas. En el otro lado, mis compañeros de laboratorio y de rutina diaria. Agradecer especialmente los descansos y charlas con: Juanma, Alberto, Chema, Toni, Epi, y Antonio. También, a Jesús y nuestro deporte el tissue-tennis (pendiente de ser nueva categoría olímpica), y a Dani y nuestro intento frustrado de conseguir la fama con PyHK, pero a falta de eso, buenos ratos.

A mis amigas, agradecer el esfuerzo de escuchar y aconsejar, a pesar de no entender muy bien de lo que hablaba, o no parecerles un tema tan crucial. Sin embargo, he tenido la suerte de compartir este periodo con mi amigo Antonio Gomariz, con el que he tenido conversaciones simbióticas que han sido sin duda los momentos más reconfortantes. También agradecer a Jeremy su ayuda conteniendo mi creatividad con el inglés, e "inspirarme" cuando frases y títulos se me atascaban.

Pero quien tiene el cielo ganao es Miguel Ángel Aguilar Avilés. Es la persona que más me ha aguantado. Hasta la saciedad. De forma regular y periódica. La misma retahíla una y otra vez. Espero que mis monólogos hagan las veces de sermones y cojas vía directa al paraíso, porque no hay manera de compensar eso.

A pesar de todo lo agradecido, lo cierto es que quien merece el grueso de estos agradecimientos son mis directores de tesis, porque han trabajado mano a mano conmigo, además, de una forma cómplice y cercana. En concreto, agradecerle a Pilar su ingeniería inversa, pedagogía y empatía, y a Juan su lógica, precisión, paciencia y generosidad.

Ana Avilés González.
Septiembre 2014.

Índice

Contents

List of Figures

List of Tables

Chapter 0

Resumen en español

0.1. Introducción

A medida que los supercomputadores aumentan su potencia, es posible ejecutar aplicaciones con mayor carga computacional y aplicaciones que procesan volúmenes de datos cada vez más grandes. Sin embargo, el rendimiento de los sistemas de almacenamiento no ha evolucionado al mismo ritmo que el ancho de banda de memoria o la potencia de las CPUs, que han crecido de forma exponencial. Es por esto que, generalmente, los sistemas de E/S se identifican como el principal cuello de botella en muchos sistemas informáticos.

En esta primera sección describimos la evolución de los supercomputadores en términos de potencia de cálculo, y también la aparición de diferentes sistemas de ficheros paralelos capaces de manejar cantidades de datos cada vez más grandes. Debido al tamaño de los datos y a un creciente número de ficheros en los sistemas modernos de altas prestaciones, sostenemos que, además de los datos, los metadatos también necesitan ser procesados de forma paralela y distribuida. A continuación describimos la motivación y, después resumimos de las distintas contribuciones de esta tesis.

0.1.1. Antecedentes

En supercomputación, generalmente se considera el CDC 6600 como el primer supercomputador de la historia. El CDC 6600 presentaba un rendimiento de 1 MFLOP, que era 3 veces superior al de la máquina más rápida de su época. Sin embargo, el supercomputador más famoso y de mayor éxito en la historia fue el Cray-1 (1976), con un rendimiento de 80 MFLOPS. Los supercomputadores en ese momento no alcanzaban los diez procesadores; de hecho, a mediados y finales de la década de los 80, Cray-2 estableció las fronteras de la supercomputación a tan sólo 8 procesadores. No fue hasta los años noventa cuando aparecieron los primeros supercomputadores con miles de procesadores. En el momento de su lanzamiento, el Intel Paragon tenía 2048 procesadores Intel i860 a $43,40$ GFLOPS de velocidad máxima. Su diseño trató de resolver la limitación de ancho de banda de E/S que presentaban las máquinas anteriores y que impedían resolver problemas a gran escala de manera eficiente.

La arquitectura del supercomputador Blue Gene [5] de IBM encontró un uso generalizado en los primeros años del siglo XXI. De hecho, 27 de los equipos de la lista TOP500 utilizaron esa arquitectura, llegando a alcanzar $478,2$ TFLOPS en 2007. Sus principales objetivos eran un bajo consumo en lugar de la velocidad de los procesadores, así como el diseño de los *sistemas-en-chip*. Blue Gene revolucionó la economía de la supercomputación debido a su tamaño pequeño y consumo eficiente.

Más recientemente, en 2010, Tianhe-I [4] se convirtió en el primer supercomputador chino en entrar en el TOP500. Asimismo, fue el primer supercomputador con una arquitectura híbrida

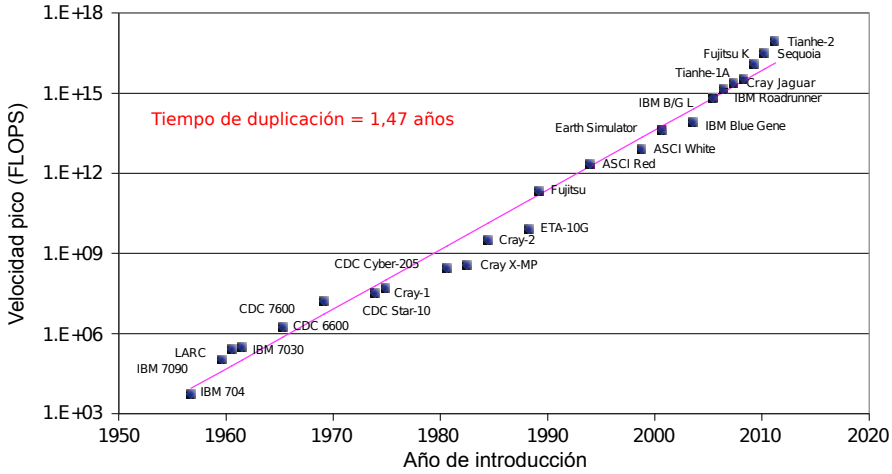


Figure 0.1: Crecimiento de la supercomputación. Fuente: «High Performance Computing - History of supercomputer» [75].

que combinaba unidades de procesamiento gráfico (GPUs) y unidades de procesamiento convencionales (CPUs).

La figura 1.1 muestra la evolución de la supercomputación en términos de velocidad máxima a lo largo de los últimos 60 años. Como se puede ver, la velocidad máxima se dobla cada 1,47 años. En 2008 alcanzamos la petaescala y, de continuar esta tendencia, la exaescala se estima para 2019.

Una mayor potencia de cálculo permite realizar tareas complejas e intensivas de datos como predicción meteorológica, investigación climática, exploración de gas y petróleo, o simulaciones biológicas o genómicas. De la misma manera, más potencia de cálculo permite aumentar el nivel de detalle de las simulaciones. Todo esto provoca que las aplicaciones procesen y/o generen enormes cantidades de datos. Sin embargo, el rendimiento de los sistemas de almacenamiento no ha mejorado al mismo ritmo que el de la memoria y la CPU, por lo que el almacenamiento suele ser un cuello de botella en este tipo de cargas de trabajo. De ahí que, con la intención de mejorar el rendimiento de E/S, surjan tanto nuevos dispositivos de almacenamiento como nuevos sistemas de ficheros capaces de manejar de forma eficiente, escalable y tolerante a fallos esas enormes cantidades de datos.

Inicialmente, los sistemas de ficheros paralelos se desarrollaron para máquinas específicas. En 1998, Dibbers *et al.* [30] implementaron un sistema de archivos intercalado llamado Bridge File System. Este sistema se implementó para el supercomputador BBN Butterfly Parallel Processor, el cual era una máquina de memoria compartida. La idea de Dibber *et al.* era diseñar un sistema de ficheros que funcionara en paralelo y que mantuviera la estructura lógica de ficheros, a la vez que distribuyera físicamente los datos. Ellos usaban el servidor Bridge (equivalente hoy en día a un servidor de metadatos) para mapear un fichero y número de bloque al fichero y número de bloque correspondiente localmente. Los bloques se distribuían en orden round-robin a lo largo de los nodos de E/S. Asimismo, contemplaron el acceso concurrente a ficheros por parte de varios clientes.

Unos años más tarde, Freedman *et al.*, en un intento de mejorar el rendimiento del Intel Paragon XP/S (una máquina con arquitectura de paso de mensajes), crearon un sistema de ficheros paralelo escalable llamado SPIFFI [38]. SPIFFI fue diseñado para ser utilizado por

aplicaciones intensivas de E/S. Para ello, proponían 3 tipos diferentes de punteros compartidos para los ficheros, con la intención de simplificar el diseño de las aplicaciones paralelas. SPIFFI dividía horizontalmente los ficheros en orden round-robin a lo largo de un conjunto de discos seleccionados por los usuarios en el momento de su creación. No había servidor central de metadatos. En su lugar, cada disco tenía un hilo de control que se encargaba de manejar las peticiones de apertura y cierre de ficheros. No obstante, debido a que en los metadatos no se incluía el tamaño del fichero, tenían que mantener en un servidor central los tamaños de los diferentes ficheros, que actualizaban cada vez que un fichero se cerraba.

Vesta [24] fue un proyecto desarrollado por IBM en la misma época que SPIFFI. Vesta formaba parte de las bases del AIX Parallel I/O File System en el IBM SP2. Vesta proporcionaba un control explícito sobre la manera en que los datos se distribuían a lo largo de los nodos de E/S. También permitía que esa distribución fuera hecha a medida en función de los patrones de acceso esperados (p.e., partición de matrices en filas o columnas para acelerar aplicaciones de multiplicación de matrices). Los metadatos de los ficheros se distribuían sobre todos los nodos de E/S aplicando funciones de dispersión (*hash*) sobre los nombres completos de las rutas. Vesta repartía los bloques alrededor de varios discos en cada nodo de E/S de forma transparente al cliente. Su implementación no necesitaba mantener los directorios para encontrar los ficheros. A pesar de esto, emulaba la jerarquía de directorios usando Xrefs (referencias a directorios) para permitir a los usuarios organizar sus ficheros y listar subconjuntos de ficheros. Sin embargo, sus estrategias de hash presentaban varios problemas: no permitían el uso de enlaces físicos, el renombramiento de directorios suponía una gran carga de trabajo, y, ante cambios de configuración del sistema, todos los objetos tenían que moverse para volver a ser colocados. Todo esto llevó a que, a pesar de no usar servidores de metadatos, consideraran la necesidad de usarlos en futuras implementaciones con el fin de solucionar los problemas asociados al uso de técnicas hash. Al igual que el anterior caso, abordaron la compartición de ficheros a través de punteros. Cada proceso podía tener un puntero independiente en el fichero compartido, o podían compartir un único puntero con otros procesos. Vesta usaba un mecanismo de paso de tokens para garantizar atomicidad en las peticiones que abarcaban múltiples nodos, y para proporcionar consistencia secuencial y linealización entre peticiones.

Dentro de los sistemas de ficheros paralelos orientados a bloques, el más conocido es el General Parallel File System (GPFS) desarrollado por IBM. Aunque Vesta es uno de los antecesores de GPFS, éste último ha evolucionado principalmente del Tiger Shark multimedia file system [43]. Tiger Shark fue diseñado para soportar aplicaciones multimedia interactivas, pero su escalabilidad, alta disponibilidad, y gestión de sistemas en línea, le hacían un candidato perfecto para ejecutar aplicaciones no multimedia, tales como computación científica, minería de datos, etc. GPFS ha heredado de Tiger Shark características como interfaces de programación POSIX, replicación a nivel de bloque, partición de ficheros, *extensible hashing* para directorios, y manejo de sistemas en línea en caso de fallos de hardware. A pesar de estas características, GPFS pertenece a los sistemas de ficheros orientados a bloques, los cuales, especialmente hoy en día, presentan limitaciones que restringen el nivel de paralelismo que el sistema puede conseguir.

En 2003, Mesnier *et al.* [65] introdujeron el concepto de almacenamiento orientado a objetos, el cual supuso un antes y un después en el diseño de los sistemas de ficheros paralelos. Con este cambio de enfoque, en lugar de ver los ficheros como un array de bloques, estos pasaban a

ser objetos. Este diseño remplazó la interfaz de bloques de los discos convencionales con una interfaz de objetos, y con dispositivos más inteligentes orientados a objetos que soportaban esta interfaz. Este cambio simplificó parte del trabajo de los sistemas de ficheros, ya que la tarea de asignación de espacio a bajo nivel correspondía a los propios dispositivos de almacenamiento.

A partir de ese momento, han surgido diversos sistemas de ficheros paralelos orientados a objetos. Uno de los primeros ha sido PVFS2 [58]. Aunque inicialmente no es descrito como sistema orientado a objetos, PVFS2 se abstrae de la interfaz de bloques para ver los datos y metadatos como ficheros. Asimismo, asigna un descriptor único a cada uno. Los servidores PVFS2 pueden funcionar como servidores de metadatos y/o datos. Los ficheros se distribuyen de forma estática alrededor de los servidores, a los cuales se les asignan distintos rangos de descriptores [22] al inicio. PVFS2 no es POSIX y no implementa ningún tipo de redundancia de datos.

Dentro de los sistemas de ficheros paralelos orientados a objetos, Lustre [20] ha sido uno de los que más éxito ha conseguido, siendo el sistema usado por alrededor de un 60% de los supercomputadores [48] del TOP500. Lustre comenzó como un proyecto de Peter Braam en la Carnegie Mellon University. A pesar de que la arquitectura de Lustre ha tenido un único servidor de metadatos durante mucho tiempo, recientemente han introducido cierto soporte para varios servidores de metadatos, que permite crear manualmente un subdirectorio en un servidor de metadatos distinto al padre. Como ya hemos dicho anteriormente, un único servidor de metadatos termina por ser un cuello de botella en el sistema. La mayoría de los metadatos se almacenan en los servidores de metadatos, y los clientes acceden a ellos para localizar los datos. Lustre soporta las semánticas POSIX, pero no provee ningún tipo de redundancia.

PanFS [101] es una solución comercial desarrollada por Panasas para su aplicación Active Stor. El manejo de metadatos se distribuye entre un conjunto de nodos de gestión (*managers*), los cuales manejan todo el clúster, implementan las semánticas distribuidas del sistema, y manejan el proceso de recuperación de los nodos que fallan. Los nodos de almacenamiento almacenan tanto datos como metadatos, y proveen al sistema de un acceso paralelo y redundante. Los ficheros se dividen a lo largo de objetos usando un RAID a nivel de fichero. De esta forma, se permite a los clientes calcular y escribir la paridad de sus ficheros, moviendo así este cálculo de los nodos de almacenamiento a los clientes. Asimismo, gracias a la distribución RAID a nivel de fichero, los fallos se acotan a ficheros individuales, y el proceso de recuperación se lleva a cabo en paralelo por los *managers* de metadatos. PanFS proporciona una interfaz POSIX, donde los clientes tienen un único punto de montaje a través del cual acceden al sistema.

Por último, Ceph [97] es el más joven de todos los sistemas orientados a objetos que hemos mencionado. Todavía no se encuentra en estado de producción. Ceph es un sistema POSIX, formado por un clúster de metadatos, un clúster de datos y un grupo de monitores. De forma similar a PanFS, los metadatos y los datos se almacenan en los servidores de datos, y se replican a lo largo de agrupaciones de los servidores de datos. Dentro de cada grupo, los nodos replican los datos y se recuperan en caso de fallo de algún nodo del grupo. Los clientes acceden de forma independiente a cualquier objeto a través de una función de distribución y un mapa del clúster que los monitores proporcionan en el momento en que acceden al sistema.

0.1.2. Motivación

En la computación de alto rendimiento, los sistemas de ficheros paralelos son un elemento fundamental para reducir el cuello de botella de la E/S y proporcionar escalabilidad, coherencia y tolerancia a fallos. Las primeras propuestas [24, 30, 38] se centraban en almacenar los datos a lo largo de varios servidores, compartiendo la carga de trabajo y proporcionando acceso concurrente a ficheros. Sin embargo, se prestaba poca atención al manejo de los metadatos. Tradicionalmente, los metadatos han permanecido en un segundo plano, ya que sólo eran una pequeña parte del total de la carga de trabajo. Los clústeres tenían un tamaño pequeño y un solo servidor de metadatos era suficiente [20]. A medida que los supercomputadores han aumentado su número de nodos y capacidad de cómputo, han permitido ejecutar aplicaciones que necesitan acceder a una mayor cantidad de ficheros y datos, y, por lo tanto, a una mayor cantidad de metadatos.

Aunque el almacenamiento de metadatos es pequeño en comparación con el de datos, las operaciones de metadatos suponen el mismo porcentaje, o incluso mayor, que las operaciones de datos [78]. Conforme los sistemas crecen, las cargas de trabajo también crecen y cambian. Actualmente, la media del tamaño de los ficheros en muchos sistemas de almacenamiento que almacenan petabytes de datos está disminuyendo, y esto, a su vez, incrementa el número de ficheros que los sistemas distribuidos de almacenamiento modernos tienen que manejar [39, 64, 72, 92]. Por ejemplo, un sistema de ficheros modesto de 100 TB con ficheros de 10 kB, debe ser capaz de manejar 10,000 millones de ficheros [103]. Esta tendencia en el tamaño de los ficheros se mantiene a medida que los clústeres crecen y se adaptan a una mayor variedad de aplicaciones que generan distintos tipos de cargas de trabajo y tamaños de ficheros [18, 26, 83]. Algunas de estas aplicaciones, por ejemplo, crean un fichero pequeño por cada hilo que lanzan, en lugar de crear un único fichero para todos los hilos porque de esta manera se simplifica el diseño e implementación de la misma [35, 71].

Aplicaciones más tradicionales también se ven afectadas por un mayor número de ficheros, aumentando de forma considerable el acceso a los servicios de metadatos y presionando los servicios de metadatos todavía más. Hay muchas tareas que se ejecutan regularmente (incluso diariamente) para mantener un repositorio de datos genérico como: copias de seguridad, migración de seguridad, procesos de indexación o de etiquetar ficheros, etc. Estas tareas necesitan enumerar o identificar conjuntos de ficheros en los repositorios de datos para hacer un procesamiento posterior. Sin embargo, ejecutar estas tareas puede afectar la ejecución normal del sistema, de modo que éste retrasa las tareas a momentos en que no se ejecuta ningún trabajo de producción [39]. Aunque hay algunas soluciones que pueden acelerar la búsqueda de ficheros que cumplen determinados criterios [51, 52, 59], un enfoque más general mejoraría en gran medida el rendimiento de los metadatos en los sistemas de almacenamiento actuales.

Por todo esto, nosotros proponemos el diseño e implementación de un servicio de metadatos de alto rendimiento, distribuido y escalable. Para ello creamos una nueva arquitectura de sistema de ficheros paralelo, la cual permite tener un clúster de metadatos tan grande como el clúster de datos. Con éste objetivo, el nuevo sistema de ficheros une el manejo y almacenamiento de los datos y metadatos en un nuevo tipo de servidor: el dispositivo OSD+. Usando dispositivos de almacenamiento orientados a objetos como los *object-based storage devices* (OSD), aprovechamos su inteligencia inherente para llevar a cabo el manejo de los

metadatos, y para distribuir los metadatos a lo largo de varios servidores. Con este diseño, también incrementamos la capacidad total del sistema y su escalabilidad.

0.2. Contribuciones

El objetivo de esta tesis es construir un clúster de metadatos distribuido a través de dispositivos basados en objetos. Hasta la fecha, se han propuesto muchas soluciones para el manejo de datos en clústeres de gran tamaño que funcionan bien [99], y que se pueden exportar a otros sistemas, por lo que hemos preferido centrar nuestro trabajo en proporcionar un servicio de metadatos eficiente y escalable, el cual sea capaz de responder a la creciente demanda de metadatos de la era de la petaescala. A continuación se presentan las mayores contribuciones de nuestra investigación.

- Nuestra primera propuesta es el *Fusion Parallel File System* (FPFS). FPFS es un sistema de ficheros paralelo donde todos los dispositivos trabajan como servidores tanto de datos como de metadatos. Esta es una arquitectura sencilla con la que incrementamos la capacidad del clúster de metadatos, que pasa a ser tan grande como el clúster de datos. Asimismo, incrementamos el rendimiento y escalabilidad de todo el sistema.
- Nuestra segunda contribución es el dispositivo OSD+. Proponemos el uso de dispositivos OSD mejorados para construir la arquitectura de FPFS. Los dispositivos OSD regulares sólo pueden manejar objetos de datos y sus operaciones asociadas; nuestra propuesta amplía los dispositivos OSD, de manera que también puedan manejar metadatos a través de lo que hemos llamado *objetos de directorio*.
- Nuestra tercera aportación es el diseño e implementación de un clúster de metadatos basado en dispositivos OSD+ para FPFS. En el clúster distribuimos de forma uniforme los metadatos a través de todos los nodos del clúster por medio de los objetos de directorio. La distribución que hacemos del espacio de nombres minimiza las migraciones en caso de renombramientos de directorios o de cambios dentro del clúster. La atomicidad se garantiza por medio de un protocolo de compromiso en tres fases y de los sistemas de ficheros locales en cada OSD+.
- Nuestra cuarta contribución es el diseño e implementación del manejo de directorios gigantes en FPFS. Estos directorios almacenan desde miles a millones de entradas que son accedidas por miles de clientes a la vez. Utilizamos los objetos de directorio en los dispositivos OSD+ para permitir a FPFS distribuir de forma dinámica esos directorios a través de varios servidores. A su vez, este diseño mejora el manejo que hacemos de los renombramientos, evitando gran parte de las migraciones de datos.
- Nuestra última contribución es el diseño e implementación de las *operaciones por lotes*. Estas operaciones juntan cientos o miles de entradas del mismo tipo de operación en un sólo paquete. Con estas operaciones conseguimos hacer un mejor uso de los recursos existentes, desplazando, en muchos casos, el cuello de botella de la red a los servidores.

0.3. The Fusion Parallel File System

La arquitectura de un sistema de ficheros paralelo suele estar formada por tres componentes principales: los clientes, los servidores de metadatos y los servidores de datos. Estos últimos

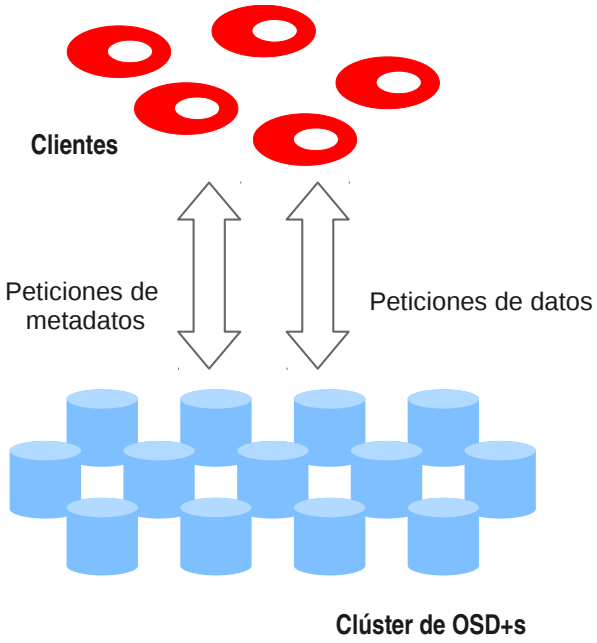


Figure 0.2: Vista general de FPFS

suelen ser dispositivos OSD [65] que exportan una interfaz de objetos y que, gracias a su inteligencia inherente, son capaces de realizar tareas de asignación de espacio en el propio dispositivo. Por su parte, los servidores de metadatos implementan interfaces personalizadas y almacenan de forma permanente información en los dispositivos de almacenamiento propios, o incluso en objetos en los servidores de datos [97, 101].

A diferencia de estos sistemas de ficheros, en FPFS [15] proponemos combinar los servidores de datos y de metadatos en un único tipo de servidor, a través de un nuevo OSD mejorado que llamamos OSD+. Estos dispositivos son capaces, no sólo de manejar datos como los OSDs tradicionales, sino también metadatos. Para esto extendemos la interfaz de los dispositivos para incluir un nuevo tipo de objeto que almacene y maneje los metadatos. Junto con el nuevo tipo de objeto, incluimos las operaciones de metadatos asociadas como crear o eliminar ficheros o directorios, o pedir atributos de un fichero. El nuevo dispositivo también simplifica la complejidad del sistema de almacenamiento, ya que no se hace distinción entre dos tipos de servidores. Ofrecemos una explicación más detallada de este dispositivo en la Sección 0.4.

Tal y como se muestra en la figura 0.2, los principales componentes de FPFS son los clientes y los dispositivos OSD+. Sin embargo, también hay un pequeño conjunto de monitores (que no aparece en la figura) que se encargan de mantener la copia original del mapa del clúster.

El mapa del clúster es el que permite que todos los miembros del clúster conozcan el resto de nodos que conforman el clúster y su estado actual. Principalmente, el mapa describe el estado del clúster y la distribución de objetos en función de *grupos de ubicación* o PGs (*placement groups*). En cada uno de estos grupos los nodos actúan como réplicas guardando el mismo conjunto de objetos. El número de miembros del grupo se establece según el nivel de replicación seleccionado, y los objetos que se almacenan en el grupo se asignan mediante la función de distribución (ver Sección 0.5).

El mapa también se utiliza como mecanismo para recuperar a los nodos caídos o desactualizados dentro de un PG. Los nodos de un mismo PG intercambian periódicamente *heartbeat messages* para detectar fallos dentro del grupo. Cada servidor etiqueta estos mensajes con el número de la última versión del mapa que tiene en ese momento. Los mensajes entre los servidores y los clientes también son etiquetados. De esta manera, los propios miembros del clúster pueden detectar nodos con mapas desactualizados y avisarles sobre las cambios en el mapa, y ellos mismos enviar las actualizaciones del mapa hasta la versión más reciente.

La comunicación de los clientes y los servidores con los monitores es mínima: se reduce a cambios en el estado de un dispositivo (p.e., falla, se une al clúster, etc.). Cada vez que un cliente se une al clúster, un monitor le proporciona la última versión del mapa. Cada vez que un OSD+ se une, los monitores actualizan el mapa e informan de la nueva versión. El clúster de monitores se encarga de generar y actualizar el mapa del clúster, evitando que se produzcan incoherencias en él. Mientras que los servidores son autónomos (pueden localizar, replicar, detectar y recuperarse de fallos sin la ayuda de un servicio central), la coherencia del sistema se mantiene a través del mapa del clúster que todos los nodos comparten.

Además de esto, los monitores juegan un rol clave en la recuperación del sistema. Son ellos los encargados de comenzar el proceso de recuperación cuando un cliente o servidor les avisa del fallo de un OSD+.

Por último están los clientes. Estos contactan con cualquier objeto a través del mapa y de la función de distribución. De forma parecida a lo que ocurre en otros sistemas de ficheros paralelos, para acceder a un fichero, primero acceden al objeto del directorio que lo contiene. Este objeto les informa de la distribución de los objetos de datos del fichero. A continuación, el cliente puede contactar directamente con el OSD+ que almacene el objeto de datos deseado para enviarle operaciones de lectura/escritura.

0.4. El dispositivo OSD+

Los sistemas basados en objetos reemplazan el concepto de fichero como una secuencia de bloques en disco por el de un fichero como un objeto. En los sistemas tradicionales, un fichero estaba compuesto por bloques de datos y bloques de metadatos (nodos-i y bloques indirectos). Con el enfoque orientado a objetos, un fichero es un objeto formado por un identificador, datos, metadatos gestionados por el dispositivo (posición de bloques físicos, tamaño, fecha de creación/modificación, etc.), y atributos accesibles a nivel de usuario que describen características del objeto (p.e., patrones de acceso, calidad de servicio, etc.) [65]. Un fichero se convierte así en un objeto y sus atributos, o, en otras palabras, un fichero se convierte en sus datos y metadatos asociados.

Los sistemas de ficheros paralelos actuales [20, 49, 58, 97, 101] suelen presentar una arquitectura basada en objetos donde los discos convencionales se sustituyen por dispositivos orientados a objetos (OSDs). Los OSDs son dispositivos «inteligentes» capaces de almacenar y devolver *objetos* (u *objetos de datos* en la terminología de FPFS) en lugar de simplemente guardar datos en sectores de disco. De esta manera, las funciones de almacenamiento de bajo nivel se transfieren a los propios dispositivos.

FPFS mejora los OSDs existentes añadiéndoles la capacidad de manejar metadatos. Para esto ampliamos la interfaz de objetos para que incluya el almacenamiento y manejo del objeto que hemos llamado *objeto de directorio*. Por lo tanto, los nuevos OSDs, que llamamos OSD+,

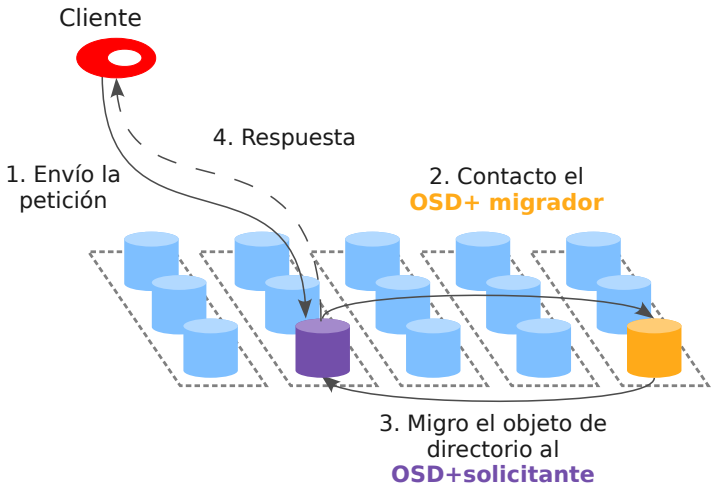


Figure 0.3: Migración de un objeto que había sido renombrado.

son capaces de almacenar entradas de directorio y realizar operaciones sobre ellas. Soportan dos tipos de objetos: los objetos de datos de la misma manera que los OSD convencionales, y los objetos de directorio.

Internamente, implementamos los objetos de directorio como directorios regulares cuya ruta es la ruta completa del directorio en FPFS. De esta manera, replicamos parte del espacio de nombres global dentro de cada OSD+. Asimismo, permitimos que los clientes puedan acceder a los objetos directamente sin necesidad de realizar ninguna resolución de nombres. Evitamos así sobrecargar a los directorios más altos en la jerarquía con peticiones de resolución de rutas.

Sin embargo, tal y como se explica en la sección 0.5, FPFS también usa la ruta completa de los directorios para distribuir sus objetos con una función hash. Esto provoca que al renombrar un directorio, cambie la ubicación de su objeto. Para minimizar el efecto de las migraciones en el sistema, FPFS retrasa la migración de objetos hasta que algún cliente vuelva a solicitar ese objeto. Los OSD+s migran el objeto de forma independiente y transparente al cliente, como se muestra en la figura 0.3. En el ejemplo un cliente solicita un directorio que ha sido renombrado, pero sobre el que queda pendiente la migración. El servidor al que se dirige es el OSD+ que debe almacenar el objeto de directorio después del renombramiento. Este OSD+ detecta que tiene la migración de ese objeto pendiente y comienza el proceso. Para ello contacta con el OSD+ migrador (que almacenaba el objeto de directorio antes de ser renombrado), y ambos llevan a cabo de forma coordinada el intercambio.

Cada objeto de directorio mantiene una entrada por cada fichero y subdirectorio con el fin de preservar la jerarquía del sistema de ficheros y proporcionar semánticas de directorio estándar. Así, los objetos de directorio crean un fichero vacío o directorio por cada cada fichero y subdirectorio que contienen. Estos ficheros vacíos almacenan atributos de fichero, y por lo tanto, sus nodos-i se usan para almacenar los atributos. De esta forma, contenemos todos los metadatos en el objeto de directorio, haciendo más eficientes patrones de acceso frecuentes como `ls -ls` que solicitan información de las entradas del directorio.

Es importante aclarar que, actualmente, no existen dispositivos OSD, de forma que se utilizan máquinas convencionales que exportan una interfaz basada en OSDs a través de

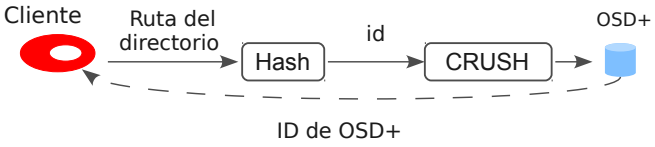


Figure 0.4: Cálculo de la ubicación de un fichero en FPFS.

emuladores [12], u otro tipo de aplicaciones[1]. Internamente, un sistema de ficheros local ordinario almacena objetos. Nosotros aprovechamos esto para mapear directamente las operaciones de directorio en FPFS a operaciones de directorio en el sistema de ficheros local. De esta manera, exportamos muchas de las características del sistema local al sistema de ficheros paralelo, tales como concurrencia y atomicidad cuando las operaciones involucran un solo directorio (y por lo tanto un solo OSD+). Cuando las operaciones de metadatos involucran más de un OSD+ (p.e., un renombramiento de directorio), los OSD+s participantes resuelven la concurrencia y atomicidad por su cuenta, sin necesidad de intervención de los clientes, a través de un protocolo de compromiso en tres fases [86].

0.5. El clúster de metadatos

Los sistemas modernos de almacenamiento distribuidos tratan, además de con grandes volúmenes de datos, con un creciente número de ficheros. A pesar de que los metadatos suponen menos del 10% del total del almacenamiento, las operaciones de metadatos representan entre un 50% y un 80% del total de operaciones del sistema [78]. Estas operaciones tienen un gran consumo de CPU, lo que implica que un único servidor de metadatos se puede sobrecargar fácilmente con unos cuantos clientes accediendo concurrentemente. En el diseño de estos sistemas distribuidos es, por lo tanto, fundamental hacer un manejo distribuido de los metadatos para evitar cuellos de botella y conseguir un buen rendimiento y escalabilidad [72]. PVFS2 [58] y Ceph [96], por ejemplo, usan un conjunto pequeño de servidores como clúster de metadatos, y Lustre acaba de incluir a partir de su versión 2.4 [54] cierta infraestructura para poder construir un clúster de metadatos similar en un futuro cercano.

En nuestra propuesta, FPFS usa dispositivos OSD+ para manejar datos y metadatos dentro del mismo dispositivo. Todos los servidores del clúster son parte del clúster de metadatos, de forma que pasa a ser tan grande como el clúster de datos. En clústeres de metadatos tan grandes son clave cuestiones como el balanceo de carga y la atomicidad de las operaciones.

FPFS distribuye de forma uniforme los objetos de directorio al lo largo de todos los servidores a través de una función determinista pseudo-aleatoria de distribución llamada CRUSH [97], la cual garantiza una distribución equilibrada. Tal como se muestra en la figura 0.4, inicialmente se aplica una función hash a los nombres completos de las rutas del los directorios, y el valor resultante se usa como parámetro de entrada para CRUSH. Así, cualquier componente del sistema es capaz de calcular de forma independiente la ubicación de

[1] Recientemente, Seagate anunció Kinetic [81], un dispositivo que es un servidor con una interfaz clave/valor y conexión Ethernet. Tiene una interfaz limitada orientada a objetos, que soporta varias operaciones sobre objetos identificados por claves. Kinetic se podría ver como una primera implementación de algo similar a la propuesta de Gibson [41], pero, debido a su diseño limitado, todavía necesita de una capa de abstracción mayor como Swift [88] para llevar a cabo operaciones básicas, tales como mapeo de objetos grandes, coordinación de condiciones de carrera en operaciones de escritura, etc.

cualquier objeto de directorio. Sin embargo, las distribuciones hash presentan problemas con los renombramientos, enlaces simbólicos, cambios de permiso y cambios en el clúster. FPFS aborda estas cuestiones a través de CRUSH y técnicas perezosas de actualización.

En el caso de los renombramientos, al cambiar la ruta de un objeto, se está cambiando también su ubicación, ya que el valor que devuelve la función hash cambia. Esto provoca la migración de un directorio y toda la jerarquía que le subyace. FPFS maneja de forma perezosa estas migraciones, aplazándolas hasta que se soliciten los objetos o la carga de trabajo sea baja, para no inundar el sistema con migraciones. Igualmente, ante cambios de permisos, las actualizaciones de estos se retrasan, de forma parecida a como se hace en Lazy Hybrid [21] (LH). Sin embargo, nuestro diseño usa el hash a nivel de directorio, por lo que los renombramientos y cambios de permisos sólo afectan a directorios, y esto nos permite reducir en gran medida la cantidad de migraciones. Varias trazas de E/S y estudios muestran que estas operaciones no son frecuentes para los directorios [21, 47, 84]

Además, utilizamos el mismo mecanismo de los renombramientos para solucionar los enlaces simbólicos. Cuando intentemos acceder a un objeto de directorio cuya ruta contenga un enlace simbólico, se producirá un error, pues la ruta real del objeto de directorio será otra y vendrá determinada por el directorio al que apunte el enlace simbólico. Para resolver este problema, una posible solución sería hacer un recorrido de la ruta desde el directorio raíz, buscando el posible enlace simbólico y la ubicación correcta del directorio al que apunta dicho enlace. FPFS, en cambio, trata estos casos como renombramientos, pero, en lugar de provocar la migración del objeto, lo que hace es devolver la ruta real del objeto de directorio, en la que se ha sustituido el enlace simbólico por el directorio al que apunta.

Frente a los cambios que se pueden producir en el clúster (caídas, incorporaciones, etc.) FPFS minimiza las migraciones a través de la propia función CRUSH, que reduce el número de migraciones en estos escenarios.

Por otra parte, garantizamos la atomicidad a través del propio sistema de ficheros local, o a través de un protocolo de compromiso en tres fases [86]. Cuando la operación sólo involucra a un OSD+ (p.e., `creat`, `unlink`, etc.), es el propio sistema de ficheros local el que se encarga de asegurar la atomicidad y semánticas POSIX. Sin embargo, hay operaciones que pueden afectar a dos OSD+s, tales como `mkdir`, `rmdir` and `rename`. Por ejemplo, en un renombramiento, el directorio fuente y destino suelen ser diferentes y, probablemente, estarán ubicados en OSD+s distintos. Para esas operaciones se usa, como hemos dicho, un protocolo de compromiso en tres fases [86], donde un nodo actúa como *coordinador* dirigiendo al resto de nodos que actúan como *participantes*.

Hemos evaluado el rendimiento y escalabilidad del clúster de metadatos a través de un prototipo, y lo hemos comparado con Lustre. Los resultados muestran que nuestro prototipo mejora el rendimiento de Lustre entre un 60%–80%, y que escala con el número de OSD+s.

En la figura 0.5 se muestra el porcentaje de mejora de FPFS frente a Lustre para una traza HP [47]. Aunque Lustre es un sistema de ficheros paralelo completo y FPFS sólo implementa un servicio de metadatos incompleto, ambos realizan aproximadamente las mismas operaciones. El hecho de que los resultados alcancen un 82% para 16/32 hilos con el sistema de ficheros Ext4, asegura que FPFS supone una mejora significativa respecto de Lustre en entornos de tiempo compartido.

FPFS supera a Lustre independientemente del sistema de ficheros que se use por debajo. Esto se debe principalmente a la delgada capa que FPFS añade sobre el sistema de ficheros

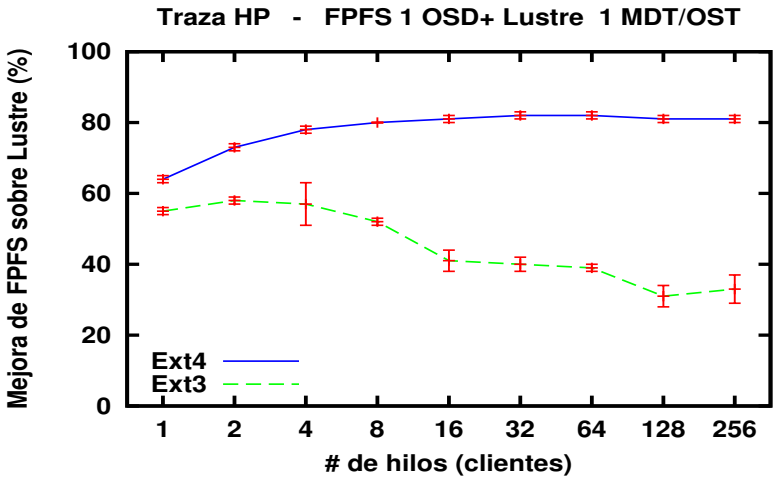


Figure 0.5: Mejora de rendimiento de FPFS con 1 OSD+ sobre Lustre.

local, la cual traduce directamente las peticiones de FPFS en peticiones al sistema de ficheros, introduciendo una sobrecarga pequeña. Por contra, Lustre añade varias capas de abstracción.

0.6. Los directorios gigantes

El aumento de metadatos no es el único problema que los sistemas de ficheros necesitan afrontar. Otro problema relacionado es el creciente uso de directorios gigantes (a partir de ahora *hugedirs*) con millones o billones de entradas que son accedidas por miles de clientes de forma concurrente [14, 19, 39, 72]. Este escenario aparece, por ejemplo, en las aplicaciones paralelas que hacen un uso intensivo de datos, y que crean un fichero por hilo/proceso [35, 71], o aplicaciones que usan un directorio como una bases de datos ligera (p.e., para *check pointing*) [76].

Para manejar un mayor número de ficheros, algunos sistemas de ficheros paralelos usan un pequeño clúster de metadatos [85, 97]. Otros esperan proporcionar un servicio similar a corto plazo [29, 82]. Pero, sin embargo, sólo unos pocos proporcionan (o planean proporcionar) algún tipo de soporte para hugedirs [29, 97, 105].

Nuestra propuesta para hugedirs utiliza los dispositivos OSD+s y los objetos de directorios. Aprovechamos los objetos de directorios para distribuir de forma dinámica los hugedirs en varios objetos que son capaces de funcionar de forma independiente. De esta manera, mejoramos el rendimiento y la escalabilidad del sistema.

Hasta ahora hemos asumido que cada objeto de directorio es manejado por un OSD+. Esta es probablemente la manera más eficiente cuando los directorios son pequeños, especialmente cuando se usan discos duros, ya que repartirlos en varios servidores produciría latencias de disco en todos los servidores para leer únicamente trozos minúsculos [72]. Sin embargo, para directorios gigantes, el tratamiento debe ser otro.

Consideramos que un directorio es gigante cuando almacena más de un determinado número de ficheros; una vez este umbral se supera, el directorio se reparte entre un conjunto de nodos

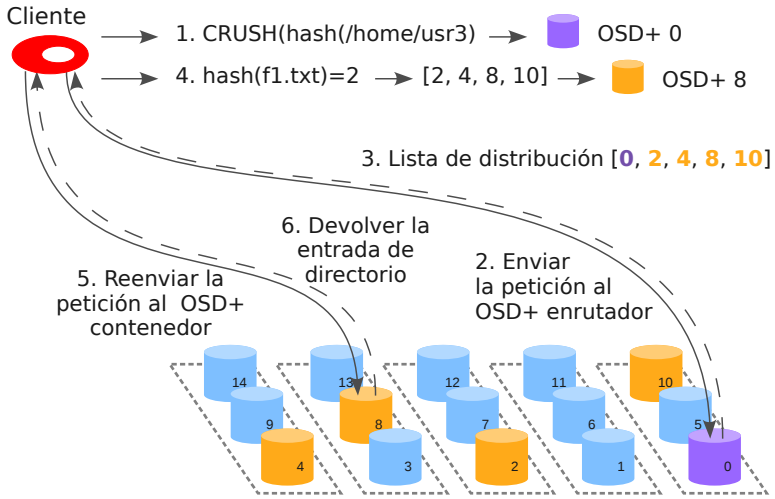


Figure 0.6: Ejemplo de un cliente enviando una petición a un hugedir.

del clúster. Un directorio también se puede distribuir desde el principio si se establece como umbral 0.

Los OSD+s que contienen un hugedir están formados por un *OSD+ enrutador* y un grupo de *OSD+s contenedores*. El primero contiene el *objeto de directorio enrutador*, que se encarga de proporcionar a los clientes la información de distribución del hugedir. Los *OSD+s contenedores* tienen los *objetos de directorio contenedores*, que almacenan el contenido del directorio, y son los OSD+s con los que contactan los clientes que están al tanto de la distribución. El OSD+ enrutador puede formar parte del grupo de OSD+s contenedores en el caso en que también almacene contenido de directorio. Obsérvese que en los directorios pequeños o sin distribuir, los objetos enrutador y contenedor son el mismo.

El conjunto de OSD+s enrutadores y contenedores se describe a través de lo que llamamos *lista de distribución*. La lista almacena los IDs de los OSD+s que conforman el grupo. Comienza con el ID del OSD+ enrutador, seguido de los IDs de los OSD+s contenedores.

Hasta ahora usábamos una función hash a nivel de directorio para distribuir el espacio de nombres. Para distribuir los hugedirs entre varios nodos tenemos que cambiar la distribución a una función hash a nivel de ficheros. La función se muestra en la ecuación 0.1, donde *osd_set* es la lista de OSD+s contenedores. Cuando queremos localizar un fichero que se encuentra dentro de un hugedir, necesitamos la lista de distribución y el hash a nivel de fichero. En la figura 0.6 se muestra un ejemplo de un cliente, que originalmente desconoce que el directorio está distribuido, solicitando el fichero `/home/usr3/f1.txt`, donde `/home/usr3/` es un hugedir.

$$oid = osd\_set[hash(filename)]. \tag{0.1}$$

Nuestro diseño de los hugedirs también elimina las migraciones en el caso de renombramientos, ya que sólo es necesario mover el OSD+ enrutador. Los OSD+ contenedores se mantienen, ya que su posición no depende de la ruta del directorio, como sucede con el enrutador.

Los resultados experimentales muestran que FPFS supera los requisitos de las aplicaciones HPC en lo que se refiere a hugedirs (millones de ficheros por directorio, más de 40.000 ficheros creados por segundo, etc.), mejorando OrangeFS en un orden de magnitud. U-

sando Gigabit, sólo 8 OSD+s sobre discos duros, y Ext4 como sistema de ficheros, nuestra propuesta es capaz de crear más de 70.000 ficheros/segundo, obtener información de más de 120.000 ficheros/segundo y borrar más de 37.000 ficheros/segundo, para un directorio con 3.200.000 ficheros. Cuando el sistema es ReiserFS, estos números son 118.000, 97.000 y 67.000 ficheros/segundo, respectivamente. Estos ratios son incluso mejores cuando usamos discos SSD en lugar de HDD: obtiene información y borra más de 118.000, 218.000 y 135.000 ficheros/segundo, respectivamente para Ext4, y 132.000, 257.000 y 104.000 ficheros/segundo con ReiserFS. La escalabilidad es generalmente lineal, siendo super-lineal en algunos casos. Esto permite a FPFS alcanzar fácilmente requisitos más exigentes, simplemente incrementando el número de OSD+s en el clúster.

Para aplicaciones de checkpointing, mdtest muestra que la distribución también es beneficiosa. Con Ext4, FFPS logra 110.000 ficheros/segundo con 8 SSD-OSD+s, mientras que con ReiserFS consigue alrededor de 70.000 ficheros/segundo.

Sin embargo, los experimentos también muestran resultados inesperados. Mientras que la distribución es beneficiosa cuando hay muchos clientes accediendo a un hugedir, puede empeorar el rendimiento cuando unos pocos clientes acceden concurrentemente a varios hugedirs. Los discos SSD mitigan en gran medida este problema, al eliminar la sobrecarga de los movimientos de cabezas que limitan el número de IOPS en HDD. FPFS también ayuda a evitar este problema, distribuyendo los hugedirs alrededor del clúster, previniendo así que los OSD+s compartan varios hugedirs.

Los resultados cuestionan la distribución basada solamente en el tamaño del directorio, por lo que son necesarios nuevos métodos de distribución que tengan en cuenta el número de procesos y los recursos disponibles en los servidores.

0.7. Las operaciones por lotes (batchops)

En los sistemas de ficheros paralelos hay varios subsistemas involucrados, tales como las CPUs, los dispositivos de E/S o la red. Dependiendo de las cargas de trabajo y los recursos disponibles, cada uno de esos subsistemas sufrirán una mayor o menor sobrecarga, pudiendo llegar a convertirse en un cuello de botella y limitar el rendimiento de todo el sistema.

Concretamente, aplicaciones donde los clientes y servidores intercambian una gran cantidad de paquetes, pueden terminar por saturar la red. Con el objetivo de salvar este cuello de botella, hemos diseñado las operaciones por lotes, o *batchops*. Estas operaciones permiten agrupar cientos o miles de entradas de un mismo tipo de operación en un único paquete. A través de batchops, hacemos un uso más eficiente de la red, desplazando, en muchos casos, el cuello de la red a los servidores. Con las batchops, reducimos el número de paquetes (y, por lo tanto, el número de cabeceras), ahorrando latencias de red y *round-trips*.

Las batchops son una herramienta particularmente útil, no sólo para aplicaciones en general, sino también para sistemas de ficheros paralelos que migran datos o, tal y como hace FPFS, distribuyen o migran directorios completos.

Implementamos el manejo de las batchops en la librería de FPFS, incluyendo operaciones específicas para la creación, consulta de entradas y borrado de ficheros. Para ello modificamos el formato de los paquetes, e incluimos una lista de entradas del mismo directorio. Nuestras operaciones por lotes soportan semánticas para indicar el comportamiento en caso de fallo de una batchop. La implementación también soporta hugedirs de forma transparente; los

clientes no necesitan hacer diferencias entre los directorios distribuidos y los no distribuidos al usar batchops.

El uso de batchops nos ayuda a reducir la sobrecarga de red, mejorando el rendimiento de FPFS en los experimentos realizados en los capítulos anteriores. En concreto, en los tests donde se hace un uso más intenso de la red, se consigue aumentar el rendimiento en un 50%, doblando el número de operaciones por segundo en algunas configuraciones. En el caso de *stat*, la mejora es de un 25%. Por último, en el caso de *unlink* donde el sistema de fichero influye en gran medida, se consigue entre un 23% y un 60% de mejora dependiendo del sistema de ficheros utilizado.

0.8. Conclusiones

Conforme el rendimiento, la escalabilidad y los requisitos para los sistemas paralelos aumentan, necesitamos arquitecturas más sencillas y descentralizadas. Tal y como hemos visto, un manejo eficiente de los metadatos es fundamental para evitar cuellos de botella en las arquitectura de los sistemas de almacenamiento, y para conseguir las características de rendimiento y escalabilidad necesarias en sistemas de gran potencia. Con el fin de proporcionar un servicio de metadatos eficiente, hemos propuesto un nuevo sistema de ficheros y su servicio de metadatos asociado basado en dispositivos OSD+. A continuación, describimos las principales contribuciones de esta tesis.

FPFS

Primero hemos diseñado el Fusion Parallel File System, un sistema de ficheros que mezcla el manejo y almacenamiento de datos y metadatos en un mismo tipo de servidor. De esta manera, los clientes pueden enviar operaciones de metadatos y datos de forma paralela a cualquier nodo en el clúster.

Este diseño simplifica la complejidad de la arquitectura al unir el clúster de datos con el clúster de metadatos en los mismos nodos. Asimismo, hace un mejor uso de los recursos existentes, ya que no necesita tener dos tipos de servidores. Generalmente, los clústeres de metadatos no alcanzan más de una docena de nodos. Con nuestra arquitectura, aumentamos la capacidad del clúster de metadatos, el cual se hace tan grande como el clúster de datos, aumentando la capacidad total del sistema y su escalabilidad.

OSD+

En segundo lugar, hemos descrito un nuevo tipo de dispositivo OSD, llamado OSD+, que soporta la arquitectura de FPFS. Un OSD+ es un OSD mejorado que, además de manejar objetos de datos, es capaz de almacenar y manejar objetos de metadatos a través de lo que llamamos objetos de directorio. Hemos extendido la interfaz de los OSDs para manejar los objetos de directorio y sus operaciones. Además, nos hemos asegurado de que las operaciones de directorio sean atómicas.

Los dispositivos OSD+ aprovechan la existencia de un sistema de ficheros local en cada nodo, y mapean directamente las operaciones de los objetos de directorio a operaciones en el sistema de ficheros subyacente. De ahí que exportemos muchas de las características del

sistema de ficheros local al sistema de ficheros del clúster, consiguiendo una gran flexibilidad, simplicidad, y una pequeña sobrecarga.

Al usar OSD+s, los datos y metadatos del sistema de ficheros paralelo pueden ser manejados por todos los OSD+s del clúster, mejorando así su rendimiento y escalabilidad.

Clúster de metadatos

La tercera contribución ha sido el diseño e implementación del clúster de metadatos de FPFS usando dispositivos OSD+s. Tener un clúster de metadatos tan grande como el clúster de datos incrementa la capacidad total del sistema y su escalabilidad, pero también conlleva retos como la distribución del espacio de nombres, un balanceo de carga equilibrado y posibles migraciones de objetos.

Usamos hash sobre las rutas de los directorios y una función de distribución pseudo-aleatoria para tener una distribución de objetos de directorio balanceada, y para minimizar la migración de los objetos cuando ocurren cambios en el clúster. Las estrategias hash normalmente sufren migraciones masivas en caso de renombramientos. Sin embargo, nosotros reducimos en gran medida el número de migraciones usando una estrategia hash sólo a nivel de directorio. Asimismo, manejamos las migraciones de forma perezosa, lo que nos permite retrasar el movimiento de los metadatos y evitar inundar el sistema con migraciones. También aprovechamos el manejo de los renombramientos para tratar los enlaces simbólicos, de manera que no se necesita ningún mecanismo extra.

La atomicidad de las operaciones de metadatos que involucran a varios OSD+s se garantiza a través de un protocolo de compromiso en tres fases, mientras que el sistema de ficheros local en cada OSD+ garantiza la atomicidad para operaciones que afectan a un único directorio.

El prototipo implementado mejora el rendimiento de Lustre en un 60–80%, y muestra que el rendimiento de FPFS escala con el número de OSD+s, siendo super-lineal en algunos casos. Los experimentos también muestran que el sistema de ficheros subyacente y las opciones de formateo pueden afectar al rendimiento del sistema. Sin embargo, ya que FPFS puede usar como sistema de ficheros cualquiera que soporte atributos extendidos, de forma sencilla se puede configurar el sistema para obtener un mejor rendimiento.

Directorios gigantes

La cuarta contribución ha sido el manejo de directorio gigantes (*hugedirs*) que almacenan millones de entradas y que son accedidos por miles de clientes al mismo tiempo. Aprovechamos los objetos de directorio para almacenar cada uno de estos directorios gigantes en varios servidores, a la vez que mantenemos las semántica POSIX.

Los objetos de directorio que soportan un hugedir funcionan de forma independiente, consiguiendo un buen rendimiento y escalabilidad. Distribuimos de forma dinámica el hugedir a lo largo de varios OSD+s en función de un umbral de número de ficheros. La versión mejorada de los objetos de directorio permite optimizar la redistribución de las entradas de directorio. También permite evitar migraciones masivas de metadatos cuando se renombra un hugedir, ya que el renombramiento sólo supone un cambio de roles entre dos nodos.

Los resultados muestran que FPFS supera los requisitos de las aplicaciones HPC en lo que se refiere a hugedirs: un billón de ficheros por directorio, más de 40.000 ficheros creados por segundo, etc. FPFS consigue un alto rendimiento de más de 80.000 ficheros por segundo,

100.000 stats por segundo, y 70.000 unlinks por segundo para un hugedir en un clúster con 8 OSD+s y discos duros, y una escalabilidad super-lineal conforme el número de OSD+s crece. Además, estos ratios mejoran cuando usamos discos SSD, ya que estos dispositivos no sufren la limitación de un pequeño número de operaciones de E/S por segundo. FPFS aumenta su rendimiento a más de 110.000 creates por segundo, más de 200.000 *stats* por segundo, y más de 100.000 *unlinks* por segundo con 8 OSD+s y SSDs.

Sin embargo, los experimentos han revelado que realizar la distribución de los directorios gigantes basándose únicamente en el tamaño de directorio puede empeorar el rendimiento del sistema. Es necesario, por tanto, tener en cuenta distintos factores o variables, o crear heurísticas que den una aproximación más fiable sobre cuál es el momento más idóneo para dividir los hugedirs.

Por último, comparamos el rendimiento de FPFS con el de OrangeFS, el cual desarrolló una versión experimental que daba soporte a la distribución estática de directorios gigantes a lo largo de varios servidores. OrangeFS es capaz de crear y eliminar alrededor de 9.000 ficheros por segundo, y de hacer stat en 10.000 ficheros por segundo con discos SSD. Los resultados de FPFS superan estos tasas de operaciones en un orden de magnitud.

Operaciones batch

Nuestra última contribución presenta el diseño e implementación de operaciones por lotes, las cuales agrupan varias entradas del mismo tipo de operación en un único paquete. Operaciones como el borrado del contenido de un directorio, la creación de un conjunto de ficheros, o los procesos de migración que mueven datos de un servidor a otro, son cargas de trabajo que se pueden realizar por lotes. Este tipo de procesamiento reduce significativamente el número de paquetes de red, ahorrando sobrecargas y latencias. Asimismo, hacemos un mejor uso de los recursos existentes, ya que se aumenta el número de operaciones por segundo que reciben los servidores.

Las batchops se incluyen en la librería de FPFS, y permiten operaciones de creación, consulta de entradas y borrado de ficheros. Nuestra implementación también permite operaciones por lotes en directorios gigantes de una forma completamente transparente a los clientes, de forma que cualquier operación funciona para cualquier tipo de directorio. Sin embargo, las batchops incluyen un campo extra para indicar la semántica de las operaciones en caso de fallo: o bien parar la operación, o continuar con las entradas restantes.

Los resultados de nuestros experimentos muestran que las batchops reducen la cantidad de paquetes de red y aumentan el número de operaciones por segundo en los servidores, mejorando especialmente para aplicaciones con un alto tráfico de red. En el caso del test de creación de ficheros se mejora el rendimiento en un 50%, y para el resto de cargas de trabajo las mejoras van desde el 23% al 60% dependiendo del tipo de carga y sistema de fichero usado.

Chapter 1

Introduction

As computational power of supercomputers increases, we can process more complex, computational-intensive and data-intensive workloads. However, storage systems have not kept up with the exponential increase of performance in memory bandwidth and CPU performance, and I/O is usually identified as the mayor bottleneck for many computer systems. In this chapter, we show the evolution of supercomputers, in terms of computing power, and the emergence of different parallel file systems to handle amounts of data increasingly large. Due to the size of the data and an increasing number of files in modern HPC systems, we state that metadata also needs to be processed in a parallel and distributed way. Finally, we summarize the main contributions of this thesis.

1.1. Background

Usually, in the history of supercomputing, CDC 6600 is considered the first supercomputer. It surpassed the fastest machine in the mid-sixties, achieving a performance of 1 MFLOP, which was 3 times faster than the previous one. However, the best known and most successful supercomputer in history was Cray-1 (1976), which was able to perform 80 MFLOPS. Supercomputers at that time did not reach the ten processors; in fact, in the mid to late 1980s, Cray-2 set the frontiers of supercomputing to only 8 processors. It was not until the nineties when supercomputers with thousands of processors appeared. At the time of release, Intel Paragon had 2048 Intel i860 processors and 143.40 GFLOPS peak speed. Its design tried to solve the I/O bandwidth limit of previous machines that prevented to solve large-scale problems efficiently.

The IBM Blue Gene supercomputer architecture [5] found widespread use in the early part of the 21st century, and 27 of the computers on the TOP500 list used that architecture, reaching 478.2 TFLOPS in 2007. Some of its major features were trading the speed of processors for lower power consumption, and a system-on-a-chip design. It revolutionized the economics of supercomputing due to its small size and power efficiency.

More recently, in 2010, Tianhe-I [4] became the first Chinese supercomputer to TOP500. It was also the first supercomputer that used graphics processing units (GPUs) in addition to conventional central processing units (CPUs), having a hybrid architecture.

Figure 1.1 shows the evolution of supercomputing in terms of peak speed over the last 60 years. As we can see, the peak speed doubles every 1.47 years. We reached petascale in 2008 and, with this trend, exascale is expected for 2019.

More computational power allows to make very complex data-intensive tasks such as weather forecasting, climate research, oil and gas exploration, or biology and genomics simulations. Also, more computational power allows to simulate with a finer level of detail. In

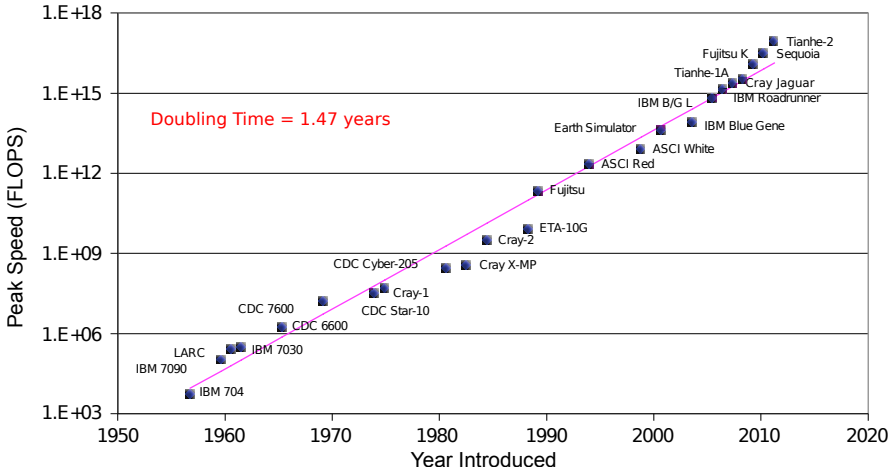


Figure 1.1: Growth of Supercomputing. Source: "High Performance Computing - History of supercomputer" [75].

both cases, parallel applications can process and/or generate vast amounts of data. However, since performance of memory and CPU has been improved at a much faster rate than performance of storage systems, the latter are usually identified as the mayor bottleneck for many computer systems. Therefore, in order to improve I/O performance, new devices have to emerge, but also new file systems able to efficiently handle such large amounts of data, in a scalable and fault tolerance manner.

Initially, parallel file systems were developed for specific machines. In 1988, Dibbers *et al.* [30] implemented an interleaved file system called The Bridge File System. It was implemented on the BBN Butterfly Parallel Processor, which was a shared-memory machine. The idea of Dibber *et al.* was to design a file system that operated in parallel and that maintained the logical structure of files while physically distributing the data. They used a Bridge server (equivalent to a nowadays metadata server) to map a file name and block into the corresponding local file name and block number. Blocks were distributed in a round-robin order among the I/O nodes. They, however, did not describe clients concurrently accessing files.

Few years later, Freedman *et al.* tried to improve the performance of the Intel Paragon XP/S (a machine with a message-passing architecture) creating a scalable parallel file system called SPIFFI [38]. SPIFFI was intended for use by extremely I/O intensive applications, and proposed three different type of shared file pointers to simplify the design of parallel applications. SPIFFI horizontally partitioned files in a round-robin fashion across a set of disks selected by users at creation time. There was no central metadata server. Instead, each disk node run a control thread in charge of managing file open and close requests. However, since metadata did not include file sizes, they kept a file's global size in a central location, and updated it whenever a file was closed.

Vesta [24] was a project developed by IBM at around the same time as SPIFFI. Vesta formed part of the basis for the AIX Parallel I/O File System on the IBM SP2. Vesta provided explicit control over the way data was distributed across the I/O nodes, and allowed the distribution to be tailored for the expected access patterns (e.g., partition of matrices into rows or columns, so as to fasten matrix-multiplication applications). File metadata was distributed among all

the I/O nodes by hashing complete pathnames. Vesta striped blocks across multiple disks at each I/O node transparently to the client. Vesta did not need to maintain directories to find files. However, a hierarchical structure of directories was emulated using Xrefs (directories references) so as to enable users to organize their files and list subsets of files. But, their hashing strategy presented several problems: no hard links, directory rename involved a huge amount of work, and, on configuration changes, all objects had to be relocated. So, although they did not use metadata servers-like nodes in this implementation, they talked about using them in the future to solve the hashing problems. They tackled sharing among files through pointers. Each process could have an independent file pointer into the shared subfile, or could share a single pointer with other processes. Vesta used a fast token-passing mechanism among the I/O nodes to guarantee concurrency atomicity of requests that span multiple I/O nodes, and to provide sequential consistency and linearization among requests.

Among the block-level parallel file systems, the most successful has been the General Parallel File System (GPFS), also developed by IBM. Although Vesta is one of the GPFS' ancestors, GPFS has essentially evolved from the Tiger Shark multimedia file system [43]. Tiger Shark was designed to support interactive multimedia applications, but its scalability, high availability, and on-line system management also enabled it to support non-multimedia applications such as scientific computing, data mining, etc. GPFS has inherited Tiger shark's features such as POSIX-compliant programming interface, block-level replication, wide striping for files, extensible hashing for directories, and on-line system management in case of hardware failure. Despite this features, GPFS is a block-oriented file system that, specially today, presents several limitations that restrict the level of parallelism the system can achieve (see Section 2.1).

In 2003, Mesnier *et al.* [65] proposed the concept of Object-Based Storage, which supposed a breakthrough in parallel file systems' design. Instead of seeing files as arrays of blocks, they proposed to see files as objects. This design replaced the block-level interface of conventional disks with an object-level interface and more intelligent object storage devices. This simplified the file systems' work, distributing low-level allocation decisions to storage devices themselves.

From that point on, different object-oriented parallel file systems started to arise. Although not initially described as an object-oriented file system, PVFS2 [58] abstracts from the block-level, seeing data and metadata as files, and assigning them unique handles. PVFS2 servers can act as metadata and/or data servers. Objects are statically distributed along the servers, which receive different handle ranges [22]. PVFS2 is not POSIX-compliant and does not implement redundancy.

Among the object-based parallel file systems, Lustre [20] has been one of the most successful, being used over 60 percent of the top 100 supercomputers [48]. Lustre started as a project in the Carnegie Mellon University by Peter Braam. Although Lustre's architecture has had a single metadata server for a long time, recently they have introduced some support for several metadata servers [46] that allows to manually create subdirectories in different metadata servers. Most metadata is stored in metadata servers, and clients access them to locate the data. Lustre supports POSIX semantics, but does not provide redundancy.

PanFS [101] is a commercial solution developed by Panasas for its computer appliance Active Stor. Metadata management is distributed among a set of manager nodes that handle the overall storage cluster, implement the distributed file system semantics, and handle recovery of failed storage nodes. Storage nodes store both data and metadata, providing parallel and

redundant access. Files are striped across objects using a per-file RAID layout. This way, they enable clients to compute and write parity of their files, moving that computation out of storage nodes to clients. Also, thanks to the per-file RAID layout, faults are constrained to individual files, and the recovery process is carried out in parallel by the metadata managers. PanFS provides a system POSIX interface, where clients have a single mount point to access the system.

Finally, Ceph [97], the youngest of the mentioned object-based file systems, is still in a non-production ready state. Ceph is a POSIX-compliant system made up of a metadata cluster, a data cluster and a group of monitors. Similar to PanFS, metadata and data are stored on data servers, and replicated among groups of data servers. Within each group, nodes replicate the data, and recover in case a node in the group fails. Clients can independently access any object through a distribution function and a cluster map that the monitors provide when clients join the system.

1.2. Motivation

Parallel file systems are fundamental in high-performance computing for reducing the I/O bottleneck, and providing scalability, coherence, and fault tolerance. Initial solutions [24, 30, 38] centered on storing data among the available servers, sharing the load and providing concurrent file access. However, they paid little attention to the metadata management. Metadata has always been left in the background, since metadata workloads have not been very high for small clusters, and a single metadata server has often been enough to respond [20]. However, as supercomputers increase their number of nodes and computational power, they allow to run larger workloads that need to access larger amounts of files and data, and, therefore, larger amounts of metadata.

Although the metadata storage is small compared to the data storage, metadata requests involve the same or even a higher percentage of operations than data requests [78]. As systems grow, workloads also grow and change. Nowadays, the average size of files in many storage systems, which store petabytes of data, is *decreasing*, and this, in turn, increases the number of files that a modern distributed storage system has to deal with [39, 64, 72, 92]. For instance, a modest 100 TB file system with files of 100 kB, must be able to manage 1 billion files [103]. This trend in file sizes is sustainable as clusters grow and become available for a larger variety of applications that generate many different workloads and file sizes [18, 26, 83]. Some of these applications, for example, create one small file per thread, instead of creating a single large file for all the threads. They do it this way because design and implementation are easier [35, 71].

Traditional procedures are also affected by a larger number of files and strain metadata services even more. There are many tasks that need to be performed regularly (even daily) to maintain a generic data repository: backup, backup migration, archival, indexing, tagging for files, etc. These tasks have the need to enumerate or identify sets of files in the data repositories for later processing. However, performing such tasks can disrupt the normal system operation, which forces them to be performed when production work is not being done [39]. Although there are some solutions that can speed up the search of files which meet some criteria [51, 52, 59], a more general approach would be to greatly improve the metadata throughput of current storage systems.

Given all this, we propose the design and implementation of a high-performance distributed and scalable metadata service by means of a new parallel file system architecture. This architecture allows us to have a metadata cluster as large as the data cluster. In order to achieve this goal, the new file system will merge data and metadata processing into a new type of server: the OSD+ device. By using OSD-like devices, we will leverage their inherent intelligence to carry out the metadata management, and distribute that management among a large number of servers. Through this design, we will also increase the overall system capacity and scalability.

1.3. Contributions

The aim of this PhD is to build a distributed metadata cluster by means of improved object-based devices (OSDs). Lots of solutions for data management have already been proposed for large clusters that work good [73, 99] and can be used, so we center our work on providing an efficient, scalable metadata service, which is ready to meet the increasing demand for metadata workloads in the petascale era.

The contributions of our research are the following:

- Our first proposal is the *Fusion Parallel File System* (FPFS). FPFS presents a design for a parallel file system where all devices work as both data and metadata servers. This is a simpler architecture with which we increase the metadata cluster's capacity, becoming as large as the data cluster, as well as system's performance and scalability.
- Our second contribution is the *OSD+ device*. We propose to use improved OSD devices to build the FPFS architecture. Regular OSD devices can only handle data objects and its related requests; our proposal is to extend OSD devices, so they can also manage metadata through what we call *directory objects*.
- Our third contribution is the design and implementation of *a metadata cluster based on OSD+ devices* for FPFS. We uniformly distribute metadata among all nodes in the cluster through directory objects. Our namespace distribution is able to minimize migrations in case of directory renames and cluster changes. Atomicity is guaranteed by a three-phase network-commit protocol, and by the local file system in each OSD+.
- Our fourth contribution is the design and implementation of *huge directories* in FPFS. These directories store thousands to millions of entries accessed concurrently by thousands of clients. We use directory objects in OSD+ devices to allow FPFS to dynamically distribute those huge directories among several servers to share their loads. This design improves the management of directory renames at the same time, avoiding migrations.
- Our last contribution is the design and implementation of *batch operations*. These operations embed hundreds to thousands of entries of the same type of operation in a single packet. Through these operations, we make a better use of the existing resources, shifting the bottleneck from the network to the servers in many cases.

1.4. Organization

This thesis is organized as follows. Chapter 2 presents the architecture and design of the Fusion Parallel File System. We introduce the basic components of the system, and the way they relate between each other and the clients.

Chapter 3 introduces the OSD+ device, describing the different types of objects it supports and their implementation. Also, key features are detailed such as renames, permission changes, metadata log, object migrations and hard links.

Chapter 4 describes the metadata cluster management and related issues: metadata distribution, client interaction, atomicity, security and fault tolerance. In this chapter we provide an evaluation of the FPFS metadata cluster performance and scalability.

Chapter 5 describes the management of huge directories with thousands of files in the FPFS metadata cluster. We modify and enhance directory objects to support their distribution among several nodes. This chapter includes the evaluation of huge directories' performance by means of an extensive set of experimental results using different backend file systems and devices.

Chapter 6 describes the design and implementation of batch operations, which embed hundreds to thousands of entries of the same type of operation in a single packet. We focus on the protocol change in order to implement the operations. The chapter includes an evaluation of batch operations for regular and huge directories.

Finally, Chapter 7 concludes this thesis by briefly summarizing our findings and suggesting future directions in this field of research.

Chapter 2

Fusion Parallel File System

This chapter introduces the basic components of the Fusion Parallel File System. Initially, we start with an overview of the most popular parallel file systems. Then, we describe the architectural design of FPFS, to further explain the elements and relationships between them and clients.

2.1. Related Work

There are plenty of different approaches to distributed file system nowadays. Here we describe the parallel file systems we have found are the keystone, whether it is because of their widespread use and/or the novel features they proposed at the time they were born.

GPFS

In modern file systems, the first to appear was the General Parallel File System (GPFS) [80] in the late nineties, developed by IBM. GPFS is a SAN file system where disks are connected over a switching fabric (e.g., fibre channel, iSCSI) to the cluster nodes. These nodes run the file system and applications, and, unlike in the object-based architectures, access in parallel the disks through a conventional block I/O interface with no particular intelligence on the disks.

GPFS stripes files in equal sized blocks in a round-robin fashion across different disks. The allocation is handled by, what they call, a central *allocation manager* and *allocation maps*.

Read and write operations on the disks are performed in parallel; hence, GPFS uses a distributed locking mechanism to synchronize access to file data, and, at the same time, to maintain POSIX read/write atomicity semantics. However, locking mechanisms limit parallelism, so they implement several techniques to scale distributed locking, such as, byte-range locking, centralized management of file metadata, segmented allocation maps, hints for disk space allocation or central coordinator for managing configuration changes.

Regarding fault tolerance, GPFS continuously monitors the health of the file system components to detect failures. Also, it sets different levels of file data and metadata replication, as well as RAID configurations on disks to avoid data loss. In addition to that, they implement storage device management through GPFS Native RAID (GNR) [89], a de-clustered approach to RAID that spreads data over all available devices and reduces the impact of drive failures. Finally, the system provides with snapshots to restore the file system to an old consistent state.

Metadata is stored as i-nodes and indirect blocks. GPFS uses a shared write lock that only conflicts with operations that require exact file size and/or mtime (modification time) to allow concurrent writers on multiple nodes. A metadata node, called *metanode*, is dynamically

elected among the nodes accessing the metadata of a file, with the help of the token server. The metanode is the only one that can read or write the corresponding i-node; operations that update a file's size or mtime non-monotonically require an exclusive i-node lock.

All metadata updates are recorded in journals or write-ahead logs stored on disks to maintain consistency, thus any node can recover any failed node anytime.

PVFS

Later, once file systems started to adopt the concept of object [65], the so-called object-based file systems begin to arise. The first one was PVFS [58]. Although it was not initially described as an object-based file system, it abstracts from the block-level view to see data and metadata as files. These files are identified through unique handles.

In PVFS, servers can act as MDSs (metadata servers) and/or IOSs (data I/O servers). Objects (both, metadata and data objects) are statically distributed across the servers by assigning each server a disjoint range of handles [22]. That mapping information is set on the configuration file, and clients get it at start-up, the first time they contact a server. The filename resolution is performed through a lookup operation that finally returns a handle, so clients can directly access files on the servers.

Metadata servers store metadata information and metadata hierarchy in a Berkeley Database, while IOSs store data as file in the local file system. Usually, data is striped among several servers in a round-robin fashion, although the striping pattern is configurable. However, PVFS does not implement redundancy, so to protect against data loss, it needs failover hardware. In order to handle server failures, PVFS uses H-A Hearbeat 2.0 [2].

Clients can access a PVFS file system either through the native PVFS library, or through the PVFS kernel module, mounting it as a regular file system. PVFS can be used with different APIs: the native API for PVFS, UNIX/POSIX and MPI-IO API. However, they do not support POSIX semantics; specifically, they support non-conflicting write semantics, since there is no distributed locking mechanism [27].

In order to provide atomicity to clients, they perform sequences of steps (called server requests) that result in what they tend to think of as atomic operations at the file system level. In case a process fails somewhere in the middle of an operation, there could be a big problem cleaning objects that were left in an intermediate state. However, those intermediate objects never get into the directory hierarchy, thus they can never be referenced and can be cleaned up without concern.

Lustre

Within the file systems using the object-oriented approach, Lustre [20] is used over 60 percent of the top 100 supercomputers [48]. It presents an object-based architecture made up of metadata (MDSs) and object storage servers (OSSs). Each MDS has attached storage devices called Metadata Targets (MDTs), and each OSS has attached Object Storage Targets (OSTs) that store objects through a particular object implementation. Although not considered as part of the file system, there is also a metadata manager (MSG) that stores configuration information for all the Lustre file systems in a cluster. The clients also contact the MSG to retrieve configuration information.

Lustre stores data as objects on the OSTs, that are internally implemented as files. Besides, Lustre supports configurable striping of files, so each one can have a different striping policy.

Nevertheless, there is no redundancy for data, so fault tolerance mainly relies on the backing storage devices (e.g., a RAID). OSTs are set in failure configuration pairs (active/passive, or active/active); thus, if a node fails, its failure partner will take over resources from the failed node. Lustre implements various mechanisms to improve the performance of the recovery process, such as Version Based Recovery (VBR), adaptive timeouts, or Imperative Recovery [102].

Previous to version 2.4 (v2.4) [54], Lustre only supported a single MDT, so the whole namespace was stored on that MDT. The latest version includes an early implementation of the distributed namespace (DNE) that allows several MDTs. By default, every directory is stored on the same MDT as its parent, but the administrator can manually move subdirectories to different MDTs. Previously, when a single MDT was supported, only on active/passive failover configuration was allowed. However, from v2.4 on, also active/active configurations are allowed for MDTs, just like on the OSTs.

Lustre stores most metadata on MDTs as i-nodes that contain file attributes, such as owner, access permissions, striping layout and access control, except for timestamps and object sizes that are stored on OSTs. Lustre implements prefetching of those file attributes on `readdir` calls to improve the performance of common operations like `ls -l`. Lustre supports POSIX semantics, except for *atime* or *flock/lockf*, so as to improve their performance.

PanFS

PanFS [101], a commercial solution developed by Panasas, is an object-based file system that separates data and metadata management on manager nodes and storage nodes, respectively. The storage nodes use their own local OSDFS [69] to manage files as objects, implementing the object storage primitives.

Storage nodes store both data and metadata, providing parallel and redundant accesses. Files are striped across objects using a per-file RAID layout; RAID-1 for small files and RAID-5 for large files. This way, they enable clients to compute and write parity of their files, moving that computation out of storage nodes to clients. Also, thanks to the per-file RAID layout, faults are constrained to individual files, and the recovery process is carried out in parallel by the metadata managers. The bandwidth of the rebuild process is increased by declustering the placement of the parity group element.

Initial data placement is uniform random, with the components of a file landing on a subset of available storage nodes. Each new file gets a new, randomized storage map. Later, data placement can change through passive or active balancing, changing the probability of a node of being chosen, or migrating data between nodes, respectively.

Metadata management is distributed among a set of manager nodes that handle the overall storage cluster, implement the distributed file system semantics, handle recovery of failed storage nodes, and provide an exported view of the system through NFS and CIFS. However, storage servers store the file-level metadata as objects' attributes. Metadata always starts out mirrored on two of the object's used to store the file's data. Directories are also stored on storage servers as special files that have an array of directory entries. Each directory is also mirrored in two objects so that the small write operations of directory updates are efficient.

Metadata namespace is divided into *volumes*, i.e., directory hierarchies that have quota constraint. Each volume is assigned by the administrator to a single metadata manager.

PanFS provides a system POSIX interface, where clients have a single mount point to access the system. It also provides POSIX semantics, and an additional *concurrent write mode* that optimizes interleaved, strided write patterns to a single file from many concurrent clients.

In order to improve performance, metadata managers provide clients with hints of the objects IDs where a file's metadata is stored. Clients cache that information and also capabilities they have been granted when accessing files and/or directories.

Metadata managers keep record of the object creation and directory updates, as well as, a log of the granted write capabilities. Managers use these logs in case of failure, to get the system back to a consistent state by checking the capabilities and operations performed before the failure.

Ceph

Finally, Ceph [97] is the youngest of the mentioned object-based file systems, and is still in a non-production ready state. It is a POSIX-compliant system made up of a metadata cluster, a data cluster and a group of monitors.

Ceph maps objects onto *placement groups* on the data cluster by applying a hash function on the object IDs. Then, each placement group (PG) is mapped onto several OSDs using CRUSH [98], a pseudo-random data distribution function, and a *cluster map*, which details the nodes and its state in the cluster. CRUSH efficiently distributes PGs across OSDs, minimizing data migration in case OSDs are added or removed from the cluster. Also, it allows to set placement rules to choose nodes in different failure domains, making PGs more reliable. Nodes within a PG are led by a *primary* node when performing tasks.

The data cluster entirely carries out data migration, replication, failure detection, and failure recovery through RADOS [99], an object storage service implemented by the OSDs themselves.

The group of monitors are also important, since they keep track of active and failed nodes, and keep the cluster map up-to-date. Although OSDs independently perform the fault detection, monitors are warned by the nodes when they detect a failure. This way, monitors update the cluster map.

Ceph stores both data and metadata on objects on a flat namespace, where each one has an unique identifier. Clients locate files' objects through their IDs, CRUSH [98] and the cluster map. First time a client starts up, contacts one of the monitors to obtain the cluster map. From then on, it can begin to access the system, since it can calculate any object's location.

Similarly to PanFS, Ceph stores file's metadata on the data cluster as object's attributes. Directories are stored as objects on the storage servers. Ceph implements *embedded i-nodes* [40], so each file's i-node is embedded within its dentry. This way, they improve the performance of the foresaid `readdir`-type operations, since they store all metadata together on disk.

Metadata management uses a Dynamic Subtree Partitioning [100]. This approach adaptively distributes responsibility for managing the file system directory hierarchy among the metadata servers as load imbalances occur.

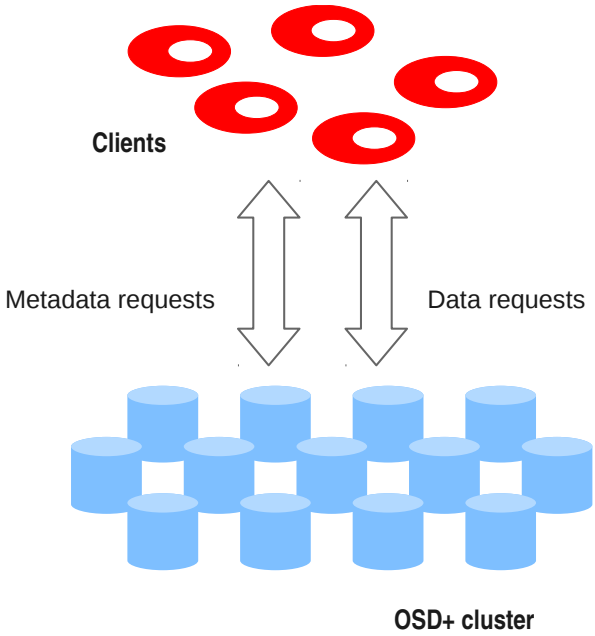


Figure 2.1: FPFS overview.

To ensure consistency, metadata servers contain a journal with operations that have not been committed to the directory storage objects yet. OSDs also deal with metadata recovery functions given that metadata is stored on the data cluster.

2.2. Architecture of FPFS

FPFS bases its data management on some of the techniques described in the above file systems. They proved to work well and provide good performance. However, FPFS proposes a novel metadata management as this thesis will show.

As we have seen, parallel file systems generally have three main components: clients, metadata servers and data servers. Data servers are usually OSD devices [65] that export an object interface and intelligently manage the disk data layout. Metadata servers, however, implement customized interfaces and permanently store information in private storage devices (or even in objects allocated in data servers [97, 101]).

Unlike these file systems, FPFS [15] merges data and metadata servers into a single type of server by using a new enhanced OSD that we call OSD+. These devices are capable not only of managing data as common OSDs do, but also of handling metadata. Thus, by using OSD+s, we increase the metadata cluster's capacity (becoming as large as the data cluster's), as well as the system's performance and scalability. The new devices simplify the complexity of the storage system too, since there is no difference between the two types of servers.

As Figure 2.1 shows, FPFS main components are clients and OSD+ devices. However, there is also a small set of monitors in charge of maintaining the master copy of the cluster map. We briefly describe all these elements in the following sections.

2.2.1. OSD+

Besides the low-level block allocation functions, OSDs can take advantage of their device intelligence by implementing more complex tasks, such as RADOS in Ceph [96], or Elliptics [73] in POHMELFS [74]. RADOS, for example, provides OSDs in the data cluster with serialization, replication, and detection and recovery from failures, in a semi-autonomous manner. OSD+s leverage this intelligence too by taking it a step further, delegating metadata management to storage devices.

Traditional OSDs deal with *data objects*, which support operations like creating objects, removing objects, and reading from and writing to a specific position in an object. Our design extends this interface to define *directory objects*, capable of managing directories. They support metadata-related operations like creating and removing directories and files, retrieving file attributes, or getting directory entries. Besides the usual operations on directories, OSD+s also provide functions to internally deal with metadata operations that may involve the collaboration of several OSD+s (e.g., renames, directory permission changes, and links). A detail explanation of the OSD+ devices is given in Chapter 3.

2.2.2. Cluster map

Cluster members see the nodes in the cluster and the cluster state through the cluster map. Essentially, the cluster map describes the cluster's state and object distribution in terms of *placement groups* (PGs). A PG is a group of nodes that act like replicas storing the same set of data. The number of group members is set by a given replication level, and the objects stored in the group are assigned by the distribution function (see Section 4.2).

The cluster monitors (see Section 2.2.3) are in charge of generating and updating the map, assuring it is always in a coherent state. Also, they provide all the members in the cluster with the latest map version.

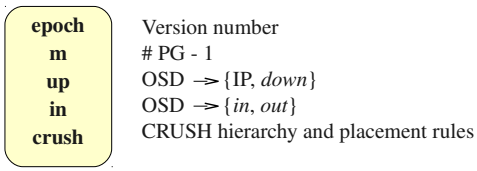


Figure 2.2: Components of the cluster map.

Figure 2.2 depicts the elements of the map. First field stores the *epoch* mark that unequivocally identifies each map's version; this mark increases as changes occur in the cluster such as an income, output, or device failure.

Map updates can be quite frequent given that, the bigger the cluster the higher the chance of a node to fail or join the cluster. These updates increase the amount of system information transferred. Hence, a reduced version of the map is used in order to minimize the amount of data transmitted in each update. Instead of using full maps, monitors create *incremental maps* that only contain the differences among members and/or the last state of the cluster map.

Regardless of the size of the map, flooding all nodes in the cluster with the updates is neither a viable option nor necessary, given that differences between maps are only significant

when they affect two nodes with some type of communication established (either two nodes of a PG, or between a server and a client). Nodes belonging to the same PG periodically exchange *heartbeat messages* to detect failures within the group (see Section 4.6). Servers tag each of these messages with the most recent *epoch* at that moment. Similarly, messages between clients and servers are also tagged. This way, the cluster members themselves can update and warn other members about map changes. In case a node receives a message with an old *epoch*, it automatically sends back incremental maps to the sender until the node is set up-to-date with the most recent version. To this end, every node keeps a history of old *incremental maps* and *epochs* of its fellow nodes in a PG.

The remaining fields describe the servers' states in terms of reachability and membership of PGs through *up* and *in* entries. For simplicity, the usefulness of heartbeat messages exchanged among PG members is to know whether a node is reachable (*up*) or not (*down*). Besides the reachable state, there is another state that defines whether a node belongs to a PG (*in*) or not (*out*). Therefore, a node may be assigned to a PG regardless of whether it is failed or not. However, despite there are two different dimensions, in the usual scenario, a reachable node has a PG assigned, and its state will be {*up, in*}. Similarly, if a node is failed, it will probably be {*down, out*}. However, there are two more possible states. State {*down, in*} indicates a node that is failed and whose data has not been remapped yet. State {*up, out*} points out an idle node because, even if reachable, it is not participating in the data storage.

The map's last entry stores the data distribution along the cluster hierarchy through CRUSH (see Section 4.2) and the *placement rules*. Previous map's fields inform CRUSH about the nodes among which data can be distributed. Aside from that, the placement rules restrict the physical location of the nodes in the same PG so as to avoid potential sources of failure (e.g. the same power source).

As a whole, given an object ID, the cluster map provides all the necessary information for any cluster member to figure out the location of that object in the system.

2.2.3. Cluster monitors

The main task of the monitors is to ensure system's coherency by maintaining the master copy of the cluster map. While servers are autonomous (they can locate, replicate, detect and recover from failures without any central service), the system's coherency needs to be maintained by means of the cluster map that all cluster nodes share.

Monitors are light processes that are usually run on the servers themselves. Communication between clients and servers with monitors is minimal; it comes down to when a device state changes (e.g. joins, fails, etc.). Each time a client joins the system, a monitor provides it with the latest cluster map's copy. When new OSD+s are added, monitors update the map and replicate that new version to the system.

Apart from maintaining the cluster map, monitors also play an essential role on recovery. Clients or servers warn the monitors about the failure of an OSD+, so they start up the recovery process (see Section 4.6) in the affected PGs.

Within the set of monitors, it is also necessary to assure coherency in monitors' replies. This is why they use the Paxos [57] algorithm, a fault tolerant algorithm for solving consensus in distributed systems. The algorithm tries to get as many nodes as possible available before performing an update. The number of available nodes must always be equal to or greater than $n/2+1$ in order to reach an agreement. The process begins with the election of a leader,

responsible for the management and classification of the nodes' proposals, and for the decision of whether to accept the proposals or not based on a consensus algorithm. Once accepted, the modifications can be sent to clients and OSD+s by any monitor.

In order to check a monitor's availability, the monitor leader sends *short-term leases* of T seconds to which the rest should reply with an *ack*. If monitors do not receive the leases in that window time, they will assume the leader is dead and a new reelection process will start. Similarly, if the leader does not get any *ack* back, it will assume a new leader reelection has been launched.

2.2.4. Clients

Similarly to other parallel file systems, to access the data of a file, an FPFS client first contacts the OSD+ containing the object of the directory of the target file. That OSD+ provides the file's data layout, so that the client can contact the OSD+s storing the corresponding data objects and perform read/write operations. Clients are able to directly contact any object any time through the cluster map and CRUSH.

The pathname of a directory is the input to locate its objects. Whereas Ceph or PVFS perform one lookup call for each element in a pathname, FPFS hashes the full pathname of a directory. A detailed explanation of how objects are located in FPFS is given in Section 4.2. This way, clients can directly contact the OSD+ that stores the corresponding directory object, which, in turn, contains the directory entry that is accessed, created, deleted, etc. For instance, a client opens the regular file `/home/usr2/docs/info.pdf` by sending an `open` message to the OSD+ containing the directory object of `/home/usr2/docs`. As reply, the OSD+ returns the data-object locations.

Similarly, when it comes to a metadata-related operation (e.g., `creat`, `unlink`, `mkdir`, ...), clients send requests to the object of the target's directory. One exception to this rule is `opendir`, where the target OSD+ is that corresponding to the pathname given as argument to the operation. Another particular case is `rename`, which receives two pathnames (the old and new paths). In this case, the request is sent to the OSD+ containing the object of the new path.

Note that there are operations, like `readdir` and `close`, that receive a file descriptor, not a pathname. Clients maintain a file descriptor table to map remote descriptors to local descriptors, thus they can work with open files.

2.3. Conclusions

In this chapter we have presented the architecture and general design of the Fusion Parallel File System, and introduce the main components of the system giving an overview of how they interact.

FPFS merges data and metadata servers in a single type of object-based device called OSD+. This design shows a simplified architecture compared to most parallel file systems that use two different type of clusters, and servers to handle data and metadata. FPFS stores and manages all in a single cluster. It makes a better use of resources since we do not need different types of servers, and increases the cluster capacity and scalability.

The design also allows clients to directly access objects in the cluster of OSD+s through a cluster map without the need to contact a metadata server.

Chapter 3

OSD+ devices

Recent parallel file systems [20, 49, 58, 97, 101] present an object-based architecture where conventional disks are replaced with object-based storage devices (OSDs). OSDs are "intelligent" devices able to store and provide with *objects* (or *data objects* in FPFS terminology) rather than simply put data in disk sectors. This way, low-level storage functions are transferred to these devices too.

FPFS improves existing OSD devices by adding the capacity to handle metadata. We extend the object interface to include the storage and management of what we call *directory objects*. These objects are able to store directory entries and carry out their related operations.

This chapter starts describing related work from the initial proposals approaching the concept of OSD to present works that attempt to make use of the inherent intelligence of these devices. Next, we describe an overview of the OSD+ device, and continue describing the different type of objects it supports and their implementation. Following, we detail issues related with directory objects, such as rename, permission changes, metadata logs, object migrations, and hard links. Finally, we give the conclusions.

3.1. Related work

While the capacity of storage devices keeps on increasing, the block-based interfaces have become a limiting factor. Gibson *et al.* [41] proposed in 1998 the Network-Attached Secure Disk (NASD) that can now be seen as the prototype of the current OSD devices. NASD modified the disk drives to provide devices with enough intelligence to manage tasks like file allocation.

In 2003, Mesnier *et al.* [65] proposed the object-based storage concept that sees a file as an object and a set of attached attributes, rather than seeing it as an array of blocks. The combination of the object idea and smart devices led on to the so-called object-based file systems [20, 58, 97].

Recently, Seagate announced Kinetic [81], a drive that is a key/value server with Ethernet connectivity. It has a limited object-oriented interface that supports a few operations on objects identified by keys. Kinetic could be seen as an early implementation of something similar to Gibson's proposal, but, due to its limited design, it still needs a higher level layer like Swift [88] to carry out basic operations, such as mapping large objects, coordinating race conditions on write operations, etc.

Despite the disks in the above proposals are more intelligent than traditional drives, object-based file systems usually do not take full advantage of that intelligence. In fact, the T10 OSD standard [53] describes OSDs as devices that passively respond to read and write commands, missing the capacity of OSDs to encapsulate significant intelligence.

Aware of the inherent intelligence, several proposals have arisen to exploit that ability. For instance, Weil has developed, as part of Ceph [97], a reliable autonomic distributed object store (RADOS) [99]. RADOS allows OSDs to work in a semi-autonomous way, so they provide the system with replication, failure detection and failure recovery. The basic idea is to group the OSDs in *placement groups* within which objects are replicated, and failures detected. A small cluster of monitors is in charge of maintaining a *cluster map* copy describing the cluster's members state. By means of that map, all members in the cluster are warned about the cluster changes, and OSDs are able to recover a fellow OSD in case of failure.

Also aiming to leverage the OSD intelligence, Devulapalli *et al.* [28] have designed iOSD, an OSD able to execute offloading computation avoiding the corresponding data movement. iOSDs are capable of process user specified computation by executing batch of operations that affect one or several objects. They have their own OSD implementation that is able to perform, at the same time, normal OSD commands and offloaded computation through a new command `OSD EXEC`. iOSD assures consistency, coherence and isolation by using different sandboxes for the offloaded computation requests.

Zhang *et al.* [106] employ OSD intelligence to provide attribute-based file accesses. Their proposal modifies an existent implementation of an OSD prototype that followes the T10 standard. By using the `SET/GET ATTRIBUTES` commands, they access and modify attributes, and store them in a SQLite database. Also, they use the *collection-object* type and `QUERY` command from the standard to index the list of objects matching attribute-based requests.

He *et al.* [44] propose to add the Hierarchical Storage Architecture (HSM) component to OSDs so the devices themselves perform the data movement. Offloading the HSM to the devices permits to reduce the data migration compared to block-based SAN systems, where data is transmitted twice, from the online storage device to the HSM host's memory, and from the memory to the lower-level storage device. Also migration tasks are more distributed among OSDs, although some coordination is also needed when migrating striped objects. However, their design, implemented on a Lustre [20] system, delegates the attribute management to MDS, and, as a consequence, MDS ends up being a bottleneck when traversing the entire file system tree to find out if a requested object is part of the layout of a file.

Finally, Ali *et al.* [13] pursue to support metadata operations on OSDs. They propose to leverage device intelligence to recouple metadata and data so as to store both on OSD devices and offload part of the metadata operations to OSD.

The approach of FPFS to leverage OSDs' intelligence is also about delegating metadata operations to the devices. However, although the aim is similar to Ali's, there are several crucial differences between the two proposals: directory implementation, atomicity, cluster size changes, recovery, and fault tolerance.

On the directory implementation, they propose to store entries either on attributes of a zero-length object, or as attributes on the first stripe of data objects. FPFS stores directory entries as empty files on a regular file system, with an idea similar to the embedded i-nodes [40]. While their two schemes still need of a directory server to provide consistency and atomicity on directory operations, FPFS skips the need of any extra type of server.

On the prototype implementation, they use a SQLite database to store object's attributes. This may be a complex software for an OSD that, in fact, obtains bad performance in their `lookup` experiments. Conversely, FPFS uses a regular file system as backend, making a

straightforward implementation that takes advantage of the atomicity, error checking, etc., already implemented in that file system.

Another important difference is atomicity. To manipulate directories and avoid using a directory server, Ali *et al.* also propose a new atomic operation, `compare-and-swap` [12]. Basically, they store entries of a directory as attributes of an empty object and translate directory operations into attribute operations. By doing so, they are depriving devices with real knowledge about directories, so they cannot deal with operations that require more than a single OSD atomically. That is, their proposal only provides atomicity at OSD level. Hence, operations that involve several OSDs, such as `rename`, `rmdir` of a non-empty directory, are not supported, or at least not in an atomic way. However, FPFS fully implements `rename` and `rmdir` using a three-phase-protocol [86] to ensure operations are atomic (see 4.4).

Their proposal also fails to address issues like cluster resizing when nodes join or leave the cluster, recovery, fault tolerance, etc. FPFS takes into account those issues implementing them or, at least, describing possible design solutions. On cluster changes, we minimize migrations through the distribution function CRUSH [98]. Also, the current implementation manages logs for recovery in case of failure, and applies lazy policies to handle migrations.

A key difference between the two proposals is that, since their OSDs are mainly passive devices, they are not able to intervene in fault detection, recovery processes, etc. Conversely, OSD+s are provided with more intelligence, so they are able to perform tasks for themselves and participate in those characteristics.

In general, their approach obtains a performance similar to PVFS's, being sometimes better or worse depending on the test. FPFS achieves significantly higher results than PVFS, thus, hopefully, it will also improve their proposal.

3.2. OSD+ overview

The key of OSD+s is their capacity of managing not only data but also metadata. As previously mentioned in Chapter 2, OSD+s support two type of objects: data objects and directory objects. The former are identified by an ID, whereas the latter use the full pathname of the directory they contain. On requests, clients specify the ID or pathname, based on the type of requested object and operation to be performed.

By identifying directory objects with the full pathname, clients can directly access them without performing a path resolution. This way, we avoid overloading higher directories in the hierarchy with resolution requests.

Directory objects have an entry per file. Each entry is made up of a file's name and its associated attributes that include the ID (or IDs) of the data object (or data objects) containing the file's data. Also, to maintain hierarchy, directory objects store an entry for each subdirectory. This way, all the metadata is contained in the directory object, making more efficient frequent access patterns that requests information of directory entries like `ls -ls`.

It is important to realize that, currently, there are no commodity OSD-based disks available, so mainstream computers exporting an OSD interface by running emulators [12], or other software applications, are usually used. Internally, an ordinary local file system stores objects. We do take advantage of this fact by *directly mapping* directory operations in FPFS to directory operations in the local file system. In this way, we export many features of the

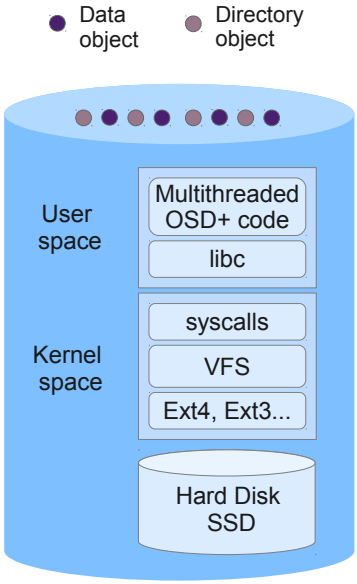


Figure 3.1: OSD+ layers.

local to the parallel file system, such as concurrency and atomicity, when a metadata operation involves only one directory (and, hence, only one OSD+). When a metadata operation involves more than one OSD+ (e.g, a rename), the participating OSD+s deal with concurrency and atomicity (see Section 4.4) by themselves, without client involvement. Note that a client issuing a metadata operation only contacts with the OSD+ containing the object of the parent directory of the target file.

Figure 3.1 shows the layers composing an OSD+. In the current implementation, each OSD+ is a user-space multithreaded process running on a mainstream computer that uses a conventional file system as storage backend. The Linux syscall interface is used for accessing the local file system that must be POSIX-compliant and support extended attributes (used by our implementation).

For every new established connection from a client or another OSD+, a thread is launched in the target OSD+. The thread lasts as long as the communication channel remains open; hence, we improve performance due to the absence of connection establishments and termination handshakes per message. In the current implementation, we use TCP/IP and UDP/IP protocols.

3.3. Files and Data Objects

Object-based file systems replace the concept of file as a sequence of disk blocks with the idea of file as an object. In a traditional system, a file was composed of data blocks and metadata blocks (i-node and indirect blocks). With the object-based approach, a file is an object that is made up of an object id, data, device-managed metadata (blocks physical location, size, creation/modification time, etc.), and user-accessible attributes that describe characteristics of the object (e.g., access patterns, quality of service, etc.) [65]. Therefore, a

file becomes an object and its attributes or, in other words, a file becomes its data and its related metadata.

Data objects are implemented as regular files in the OSD+s. Each data object has an ID, called *OID*, that is stored as an extended attribute in its corresponding FPFS file's metadata. An OID is made up of two values: an object name (a random number also used for the name of the internal regular file storing the data), and an OSD+ number (where the data object is stored). Data objects are created along 128 different directories in order to avoid too large directories, which usually obtain bad performance on regular file system.

Two different policies can be used for selecting the OSD+ where a data object is created: *same OSD+* and *random OSD+*. The former (used by default) stores a data object in the OSD+ of its file's directory. This approach reduces the network traffic during file creations because no other OSD+s participate in the operation. The latter chooses a random OSD+ instead. This second approach achieves a better use of resources by keeping a more balanced workload, although it increases the network traffic during creations.

Although data operations have not been completely implemented yet, realize that clients will be able to access data objects through OIDs stored in files' attributes. An OID indicates the destination OSD+ and the assigned object name of a file's data object in that OSD+, so clients can directly send data requests to the given OSD+.

Files, specially the large ones, can use several objects in different OSD+s to store their data. This way, they can perform read and write operations in parallel, improving the system performance and allowing clients to obtain a better transfer rate when accessing those files.

3.4. Directory objects

The design of OSD+ devices mainly differs from regular OSDs in the capability of handling metadata in addition to data. To this end the concept of directory object able to manage and storage metadata arises.

Internally, we implement a directory object as a regular directory whose pathname is its directory pathname in FPFS. Some techniques [34, 93] use an ID from the name instead. However, those approaches either require the ID to be unique, or mechanisms to control collisions. Using the full pathname is more straightforward and it also makes more sense in our design, given that clients can directly access the directory objects through their full pathname, which, moreover, is unique. Besides, this way, we replicate part of the global namespace within each OSD+.

Each directory object keeps an entry for every file and subdirectory to preserve the complete filesystem hierarchy and to provide standard directory semantics. These entries are created by means of empty files and directories in the local file system. The i-nodes of these empty entries are used for storing files and attributes. All attributes are stored on the directory entry, except for size and last modification time, which are stored as attributes of the objects for performance concerns, like Lustre [31].

This design is similar to embedded i-nodes [40] in the sense of keeping file attributes together with their directory entries, since everything is included in the same OSD+. This improves operations like `stat` or `ls -l` given that all the information can be provided with a single request. Also, it exports file attributes in the local file system to the parallel file systems, so clients can access all the usual attributes (timestamps, mode, etc.) and extended attributes

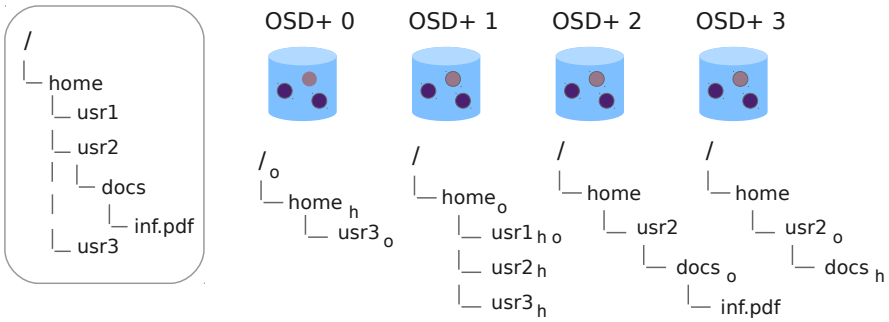


Figure 3.2: Example of mapping a FPFS file system to an OSD+ cluster.

directly in the empty file. Nevertheless, embedded i-nodes present a drawback: they fail to manage multiple hard links. We show how to solve this problem in Section 3.4.5.

Also, directory objects store permissions (see Section 3.4.2) in a dual-access control list (DACL) [21]. A DACL stores the path permissions and the directory permissions for a directory. The path permissions are the intersection of the parent directories' path permissions. In this way, we preserve the access control hierarchy, which only requires traversal in case of a directory permission change.

OSD+s internally use several directories in its local file system for different purposes. Altogether, there are four types of directories differentiated through extended attributes. The first type, *directory object*, is assigned the attribute **o**; their locations are determined by CRUSH and their path names (see Section 4.2). The second type, *hierarchy directory*, keeps the attribute **h** and refers to empty directories created in a directory object to keep the file system hierarchy (i.e., they represent subdirectories). The third one, *path directory*, does not store any attribute and is used for directories supporting the paths of the directories implementing objects. Finally, a *temporary directory* stores the attribute **t** and is used for lazy migrations (see Section 3.4.1).

Figure 3.2 shows how an FPFS directory hierarchy is mapped to a 4-OSD+ cluster. There are 1 regular file (`inf.pdf`) and 6 directories: `/`, `home`, `usr1`, `usr2`, `usr3` and `docs`. Directory objects (attribute **o**) are stored in OSD+s 0, 1, 1, 3, 0 and 2, respectively. Note that a directory object and its corresponding parent's directory object are usually placed in different OSD+s (but not always; see `/home/usr1`). Directories that preserve the hierarchy (attribute **h**) appear as subdirectories during the scan of a directory object, along with the names of the regular files in the directory object. Finally, path directories used internally for constructing a directory object's path are, for instance, `/home` and `/home/usr2` in OSD+ 2, which support the directory object of `/home/usr2/docs`.

3.4.1. Renames and temporary directories

The rename of a directory changes the location of its object, as well as the location of the objects of the underlying directories in the hierarchy. To avoid a massive migration of objects, FPFS performs a lazy migration, so that objects are only migrated when requested. Meanwhile, the object of the renamed directory is stored on a *temporary directory*, which is differentiated with the attribute **t** and is hidden to clients to avoid listing it on `scandir` operations.

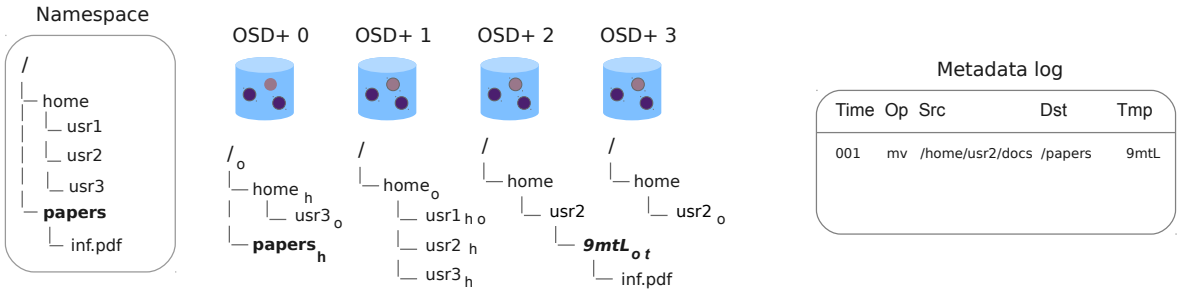


Figure 3.3: Hierarchy after performing `rename /home/usr2/docs /papers`.

During a rename, the parent of the destination directory is contacted by the client to check if it can be created, that is, if it has permission, if there is no file with that name already, etc. If so, that OSD+ generates a random name for the temporary directory and broadcasts the rest of OSD+s with the rename information, including the random name. Next, each OSD+ writes down the operation on the *metadata log*, and, in case they store that directory (either as directory object or path directory), change it to the temporary directory.

Thus, temporary directories are used for keeping objects waiting for migration and, at the same time, keep the system consistent after rename operations. Accordingly, we can divide temporary directories into two types: *temporary object* and *temporary path*. The former is the temporary directory that stores the object pending for migration, so, besides having the attribute **t** it also has the attribute **o**. The latter are the path directories that existed before the rename, and that are equally kept to maintain the hierarchy underneath. Unlike temporary objects, these only keep the **t** attribute.

Figure 3.3 shows the hierarchy of Figure 3.2 after performing `rename /home/usr2/docs /papers`. The hierarchy entry is updated after the rename, and the directory object is renamed into the temporary object `9mtL` in OSD+ 2, adding the attribute **t**. In this example there are no path directories for `/home/usr2/docs`, so there are no temporary paths directory after the rename.

Temporary directories are key to maintain the directory hierarchy consistent given that, after a rename, the source directory pathname should no longer exist on the file system's hierarchy. Otherwise, posterior operations trying to access or create the renamed pathname may not be processed correctly.

For instance, subsequent access operations that have `/home/usr2/docs` as destination path, such as `stat /home/usr2/docs` must fail returning there is no such a file or directory.

Also, subsequent creation operations that have `/home/usr2/docs` as destination, such as `mv /home/usr1 /home/usr2/docs` or `touch /home/usr2/docs`, must be successful in Figure 3.3.

Temporary directories are also used when a directory is removed. Since a directory can only be removed if empty, the removal deletes the directory object, hierarchy directory, and any path directories. However, there may be path directories storing temporary directories of a previous rename that have not been migrated yet. Therefore, `rmdir` operations change path directories into temporary directories to preserve pending metadata.

All OSD+s keep a *metadata log* (see Section 3.4.3) that, among others, contains information about the performed `rename` and `rmdir` operations. This way, the right metadata can be found during migrations.

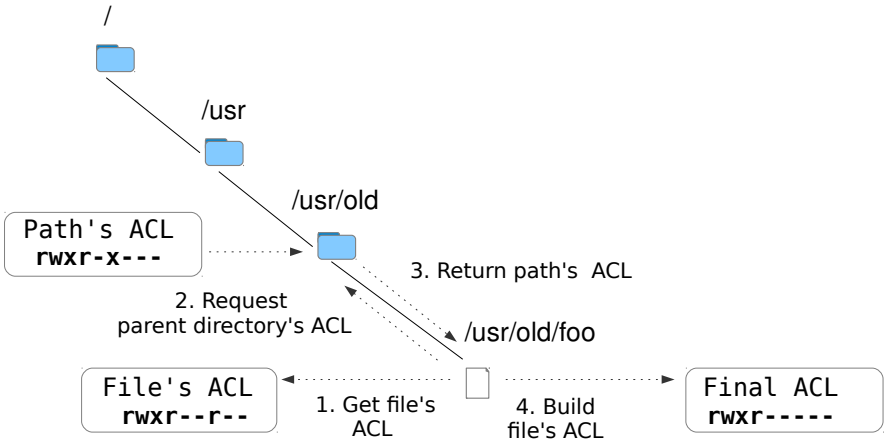


Figure 3.4: Performing a request to obtain file permissions of `/usr/old/foo`.

3.4.2. Permission changes

As aforesaid, directory objects also store directory permissions through dual-entry ACLs (DACLs), as those used in Lazy Hybrid (LH) [21], to avoid traversing the directory hierarchy when obtaining the directory permissions.

Each directory stores two ACLs. The first ACL contains the permissions of the directory itself, whereas the second one represents its *path permissions* (these are the intersection of the directory's own permissions and its parent's path permissions). Unlike LH, only directories have dual-entry ACLs in FPFS. A file's permissions are derived from its ACL, and its parent directory's dual-entry ACL, as shown in Figure 3.4.

Nevertheless, if a directory permissions change, all the hierarchy underneath is affected. Similarly to the lazy rename migrations, FPFS also performs a lazy permission update to avoid traversing the whole hierarchy on a directory permission change. Again, the update is carried out on demand, as clients request permissions that were modified.

Therefore, on a permission check, the OSD+ containing the target directory object searches in its metadata log for invalidations along the requested object's path. If they exist, it updates the DACL accessing the parent directory to get its DACL. Once the obsolete DACL is updated, the OSD+ calculates the requested ACL and returns the result.

Since parent's permissions might also be out-of-date, this process can be repeated recursively until the directory whose permissions have changed, or an updated directory, is reached. In this way, permissions are updated in a lazy fashion, minimizing the part of the hierarchy that is traversed.

3.4.3. Metadata logs

FPFS applies a lazy policy when it comes to operations that involve metadata migrations and permission changes. In order to delay those operations and still provide coherent replies to clients, OSD+s write them down in a *metadata log*. As we will see in Section 4.3, symbolic links are also stored in the metadata log for convenience, although they do not produce lazy migrations or updates.

Each time a `rename`, `rmdir`, or `chmod` is performed on a directory, all OSD+s are warned. OSD+s store an entry on the log with enough information to identify that operation and later complete their migrations or updates.

Specifically, the entries for `rename` and `rmdir` consist of the operation's pathnames, the temporary directory name, the operation's timestamp and the operation type. The entries for `chmod` contain paths whose permissions have been invalidated due to a permission change and the corresponding timestamp.

Log entries are sorted by time from the most recent to the oldest; this is important for two reasons. The first reason is to know in a efficient way (not having to go through the whole log) which entries are previous or posterior to a given entry. When an entry is checked against the log, apart from finding a match, we usually need to know if other operations have affected that object too and if they are older or newer (see Section 3.4.4). The second reason is that sorting by timestamp also ensures that consecutive operations on the same file will be processed correctly.

Operations included in the log are infrequent (see Section 4.7.1), thus the log will not grow much. Nevertheless, a maximum log size is set so entries are deleted as logs reach the threshold. Before performing a directory rename or a permission change, the OSD+ checks its log's size to see if it exceeds the threshold. If so, the OSD+ takes the oldest entry to perform the migration (see next Section 3.4.4) or DACL update (see Section 3.4.2), and deletes the corresponding log entry, informing the other servers with a broadcast message. Meanwhile, the initial operation that triggered the log trim is blocked until the ongoing broadcast message is sent.

OSD+s also try to avoid logs from reaching the threshold in advance. When their workload is low, OSD+s try to carry out pending operations in the log. Again, they check from the oldest entries on to see if they store any directory affected by the operations on those entries. If so, the OSD+ asks the other OSD+s involved in the operation whether their workload is also low to start performing the migration or DACL update. If that is the case, the OSD+ broadcasts the message to remove the log entry.

An optimization to reduce the number of log entries can be used when broadcasting rename operations. Actually, an OSD+ only needs to store the operation in its log if it is the destination OSD+ of the rename, or the rename affects any of the paths already stored in its log. This way, as soon as it performs the pending operation or remove the temporary directory, it can delete the log entry without notifying the rest. This not only prevents from writing down the entry, but also saves the last broadcast message and allows OSD+s to trim their logs independently.

3.4.4. Object migrations

FPFS lazy migrates objects that change their locations due to renames. Above we have described how to delay the migration; here, we complete the description by detailing the initiation and course of the object migration.

Figure 3.5 shows a client requesting an object that was renamed and has not been migrated yet to its new location. On step 1, the OSD+ first checks whether it stores the requested object or not. If not found, the object's path is checked against the metadata log to see if any path component is pending migration, in which case a migration process is launched.

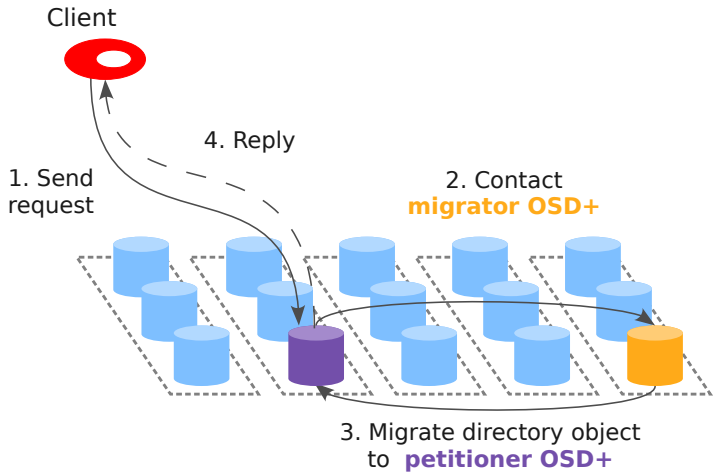


Figure 3.5: Migration of an object, requested by a client, that had a pending migration.

The migration process involves two OSD+s: *petitioner* (purple) and *migrator* (orange). The former asks the migrator (in step 2) to send a renamed object. Then, the migrator is in charge of sending the pending object to the petitioner (step 3). However, the object might be missing in the migrator due to a previous rename on the same path. In that case, the migrator initiates another migration process as petitioner to obtain the object and be able to send it to the initial petitioner.

Next, we describe the two roles independently:

Petitioner It starts the migration process when it receives a request of a missing directory object that it should have. First, it traverses the log for older renames that affect any component of the directory path. Once found, it works out the object pathname previous to the rename, and calculates its location (see Section 4.2). Next, it launches the process sending a migration request to the migrator, including the old pathname and temporary pathname.

Migrator In order to migrate the object, it first calculates the full temporary directory pathname replacing the temporary path on the old pathname sent by the petitioner. In case the path does not exist in the OSD+, it looks forward in the log for newer renames on the parents' components that affected the temporary pathname. If still missing, the object should be in a different OSD+ due to a previous rename that has not been migrated yet. Therefore, the process starts again with the migrator looking backwards in the log for a previous rename, and launching another migration process, acting now as petitioner. This process is repeated recursively until all chained renames are migrated.

Note that migration could be carried out more efficiently by directly migrating from the latest migrator to the initial petitioner. As it is now implemented, we perform as many migrations as migration processes are initiated. Luckily, consecutive renames on the same pathnames are not common.

For clarity, let's see a simple migration example by using Figure 3.3 and the most common scenario where only a single rename has been performed. Imagine a client requests `/papers`

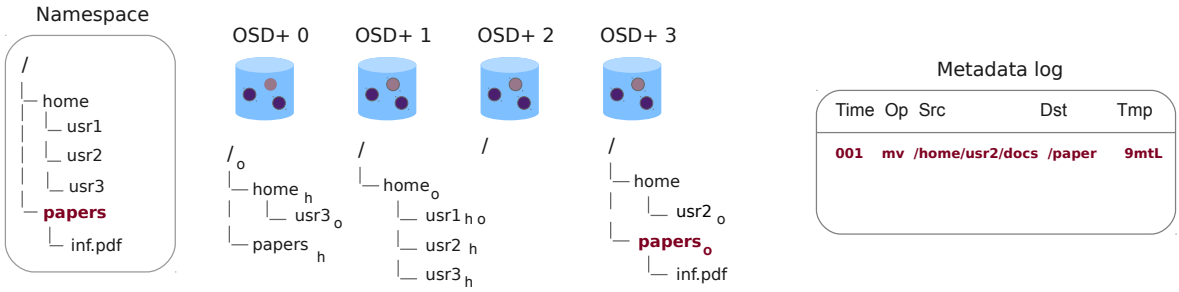


Figure 3.6: Directory hierarchy after migrating the object `/home/usr2/docs` to `papers`.

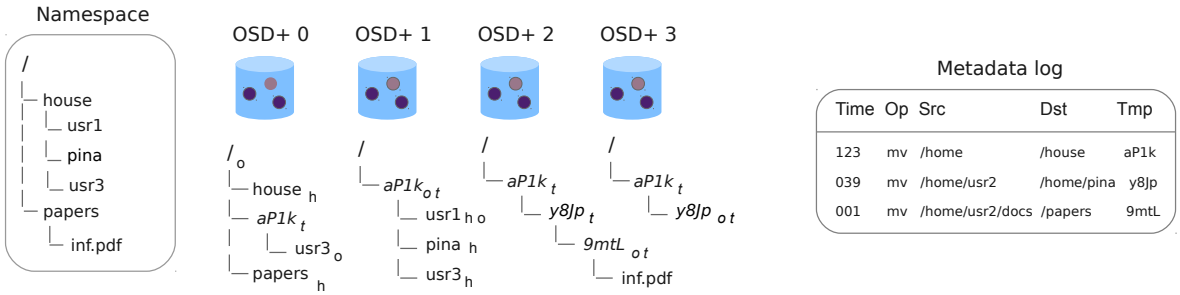


Figure 3.7: Hierarchy after performing three renames.

to OSD+ 3, where the directory object should be. The server does not have the directory object, so that it checks the metadata log for a previous rename. The metadata log reveals that a rename operation was performed with `/home/usr2/docs` as source name and `9mtL` as temporary name. Next, OSD+ 3, acting as petitioner, contacts OSD+ 2 (migrator), which is the server where the directory object of `/home/usr2/docs` was stored. Once OSD+ 2 is contacted, it builds the full temporary directory pathname: `/home/usr2/9mtL`. Once found, its content is moved to petitioner OSD+ 3 as Figure 3.6 shows, creating the directory object for `papers` and successfully concluding the migration.

The example in Figure 3.7 shows a rarer and more complex scenario where a path has experienced three renames. On the right, the figure shows the corresponding metadata log after the three renames. In this example, up to two migration processes can be initiated.

For instance, let's assume that a client requests the object `/house/pina`, which should be stored in OSD+ 1. As OSD+ 1 receives the requests, it finds the object is missing, so it checks the log for previous renames.

1st migration There is a match in the metadata log that has `/home` as old name. The server, acting as petitioner, builds the old pathname `/home/pina` and calculates its location, OSD+ 0, where the pending object should be. Next, it starts the migration process sending the old name together with the temporary name `aP1k` to the migrator, OSD+ 0. Once received, the migrator first builds the full temporary pathname `/aP1k/pina`, which is missing. To make sure the full pathname is correct, it looks forward in the log for newer renames that affected the temporary path, but, in this example, that is the latest entry.

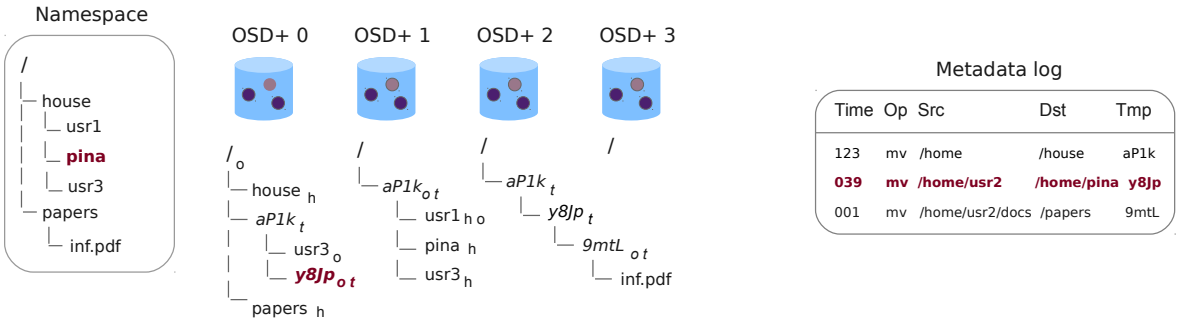


Figure 3.8: Directory hierarchy after migrating temporary object of `/home/usr2` to `/home/pina`.

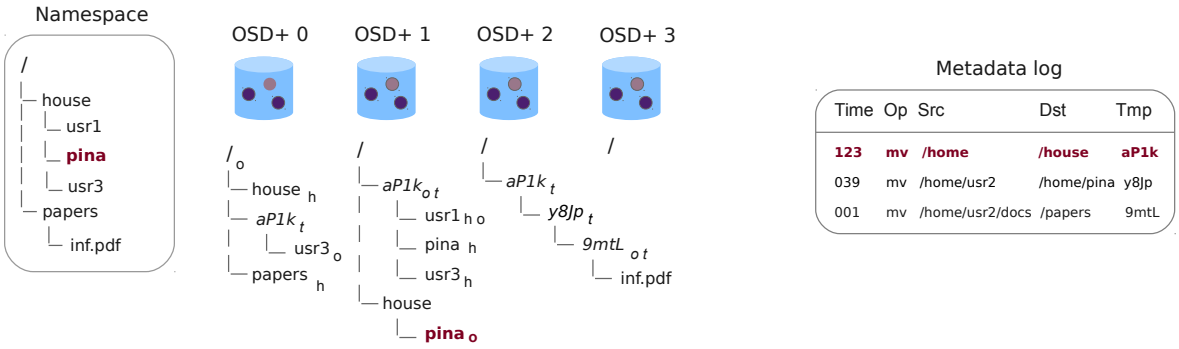


Figure 3.9: Directory hierarchy after migrating temporary object of `/home/pina` to `/house/pina`.

2nd migration Now, migrator OSD+ 0 starts looking for older log entries from the current. This time the match tells that `/home/pina` used to be `/home/usr2`. The old name now is `/home/usr2`, which was stored in OSD+ 3 (see Figure 3.2). Thus, OSD+ 0 becomes now a petitioner, and initiates another migration process with OSD+ 3 as migrator. As in the first migration, the initial temporary pathname calculated (`/home/y8Jp`) is missing, so the log is traversed forward for newer renames that affect the temporary path. There is a match on the rename of `/home` to renamed `/house`, that returns as temporary path `/aP1k/y8Jp`. This time, the temporary object exists, and its content is sent to the petitioner OSD+ 0 in the first migration shown in Figure 3.8. Once received, OSD+ 0 finishes the first migration sending the object to the initial petitioner OSD+ 1. The resulting hierarchy is shown in Figure 3.9. Note the temporary paths directories disappear when empty.

Migration process is fast (under 2 seconds in the performed tests) since directories are usually small. Moreover, for huge directories that store thousands of entries, FPFS uses several directory objects to store each huge directory to avoid the performance downgrade. As later described in Chapter 5, a rename operation on a huge directory would only affect an empty object, making migrations even faster.

3.4.5. Hard links

As mentioned before, directory objects store entries similar to embedded i-nodes, storing a file's attributes within its directory entry. However, a drawback of embedded i-nodes is that they cannot handle multiple hard links.

Ganger *et al.* [40] proposes to replace the i-node pointer field of conventional directories with a type that can take different values. In case of being a link, that field contains a link to the corresponding i-node. Weil *et al.* [97] also use embedded i-nodes in Ceph storing directory entries together with their i-nodes. They solve the handicap of hard links by means of an auxiliary table called *anchor table* [95]. Similar to conventional i-node tables, which are used for locating i-nodes based on the i-node number, this table stores references to i-nodes with more than one link.

FPFS works out this problem by storing an i-node with hard links as the i-node of a file supporting an empty data object, independently from the directory entries. Entries referencing that object only store the object reference, that is, the object ID.

The object ID is formed by a pair of values: the OSD+ ID and the object ID within that server. This pair is a sort of i-node number which makes the creation of hard links straightforward: the directory entry for the new hard link simply stores the new file name and the same object ID of the source file.

The empty object for the hard link stores all the file attributes together with a link counter to keep track of the number of links, allowing multiple hard links. Once the counter is decremented to zero, the object is deleted.

With these approach we avoid the need of an auxiliary table simplifying the implementation.

3.5. Conclusions

In this chapter we have introduced the OSD+ device, a new type of OSD device that handles not only data but also metadata requests. OSD+s manage metadata through what we call directory objects. Directory objects store file names and attributes, and support metadata-related operations. As in the traditional OSDs, data in OSD+s is stored in data objects, which mainly support `read` and `write` operations.

Our new OSD+ devices profit the existence of a local file system in the storage nodes. OSD+s directly map directory-object operations to directory operations in the underlying file system, hence exporting many features of the local to the cluster file system. This way of implementing directory objects and their operations allows us to achieve a great flexibility, simplicity and small overhead.

We have described design and implementation issues related with directory objects too. We have tried to minimize object migrations in case of renames, as well as to avoid hierarchy traversals in the case of permission changes. Also, since our design uses an idea similar to embedded i-nodes, we have proposed a solution to handle hard links.

Chapter 4

Metadata cluster

Modern distributed storage systems deal not only with a large volume of data but also with an increasing number of files. Accordingly, an efficient metadata management becomes a fundamental aspect to prevent bottlenecks and to achieve the desired features of high performance and scalability [72].

Although metadata is usually less than 10% of the overall storage capacity, its operations represent between 50% and 80% of all the system requests [78]. Metadata operations are also very CPU consuming, which means that a single metadata server can be easily overloaded by a few clients. Hence, to improve the performance and scalability of metadata operations, a cluster of servers is needed. PVFS2 [58] and Ceph [96], for instance, use a small set of servers as a metadata cluster, and Lustre provides an early metadata cluster implementation in v2.6 [46], where subdirectories can be created manually in different metadata servers. In our approach, FPFS uses OSD+ devices to provide the metadata service. Therefore, all servers in the cluster are part of the metadata cluster, so that it becomes as large as the data cluster.

In this chapter we describe the metadata cluster management in FPFS and related issues. We begin with the related work of metadata distribution and storage. Next, we specify the metadata distribution approach for FPFS. Following, we describe how clients interact with the OSD+s for metadata operations, and how to provide atomicity, security, and fault tolerance within the metadata cluster. Then, we evaluate the performance of our approach with experimental results. Finally, we give the conclusions.

4.1. Related work

As parallel file systems start to handle metadata with a metadata cluster, two different aspects of design arise: how to distribute the metadata management among the cluster, and where to store the metadata. We analyze the related work for each feature separately in the following sections.

4.1.1. Namespace distribution

The distribution of the file-system namespace across metadata servers is crucial to make a balanced use of resources and to get a good performance. It may also determine scalability problems related to certain metadata operations or changes in the cluster due to additions, removals or failures of servers.

Existing techniques go from a coarse-grain approach like Static Subtree Partition (used by Lustre [31], Coda [79], AFS [67], etc.) to a fine-grain approach like File Hashing [21, 60, 94, 107].

Static Subtree Paritition statically assigns portions of the file hierarchy to metadata servers, thereby preserving directory locality. However, it is vulnerable to distribution imbalances as the file system and workload change. A variant of this scheme is Dynamic Subtree Partition, used by Ceph [96], which dynamically delegates authority for directory subtrees to different metadata servers. Periodically, busy servers transfer authority for some subtrees to non-busy servers.

Fine grained techniques such as file hashing [24] provide load balance at the expense of losing locality. These techniques apply a hash on pathnames to locate metadata, having the advantage that clients can directly locate the MDS. Another advantage is that, usually, the load is evenly distributed, avoiding hot spots. However, there are several drawbacks such as the loss of directory locality, and massive data migrations due to a cluster size change or, for example, a rename. Also, the namespace hierarchy needs to be maintained in order to perform ls-like operations on a directory.

Lazy Hashing (LH) [21] is half-way between *Pure Hashing* [24] and *Directory Subtree Partitioning*, and tries to combine their best characteristics. LH mitigates the migration with a *metadata look-up table* (MLT), which maps hash value ranges to server IDs. Therefore, the client hashes the filename and uses the output as an index into the MLT. Even though this may slow down the metadata lookup, it also simplifies assignment of hash values and reduces the metadata movement due to metadata cluster resizing. Further, LH also applies lazy policies on renames and permission changes to defer a migration until a data is accessed again. In addition, it includes a dual-entry *access control list* (ACL) to avoid directory traversals when checking permissions.

Features introduced by LH have been widely borrowed by schemes such as *Dynamic Hashing* (DH) [94], *Directory Object IDentifier and Filename Hashing* (DOIDFH) [34] or *Mimic Hierarchical directory Structure* (MHS) [93]. All these schemes use LH's access control mechanisms.

DH is a file hashing technique that combines lazy policies and an MLT with several new strategies to dynamically adjust the metadata distribution. These techniques are: RElative LoAd Balancing (RELAB) for metadata redistribution; Elasticity for metadata movement as cluster size changes; and Whole Life-time Management strategy (WLM) to identify hot-spots.

DOIDFH is a directory hashing technique that keeps the directory hierarchy in every metadata server to prevent them from becoming hot-spots. Nonetheless, metadata-intensive workloads that modify the hierarchy can easily saturate the system.

Conversely, MHS, which also uses a directory hashing strategy, employs an indirect method to imitate the directory hierarchy and avoid the overhead of updating it. For this aim, MHS uses several conversion tables. On the one hand, there is a *buckets index table* that transforms directory pathnames into unique directory identifiers. This table is stored on every MDS and client, and may suffer modifications as the hierarchy changes, leading to metadata migrations. However, those migrations can be handled in a lazy fashion, moving the metadata when it is first requested. On the other hand, there is an MLT to distribute files' metadata. In order to access the table, the value of hashing the directory ID together with the filename is used. Equally to LH's MLT, whenever the MDS cluster size changes, metadata must be migrated.

Both, DOIDFH and MHS avoid data migrations due to directory renames by assigning to every directory a unique ID that never changes (they use a global index table for this

purpose). A main drawback is that, to assure ID's uniqueness, a single metadata server creates directories, thus that MDS could become a bottleneck.

Hierarchical bloom filter arrays (HBA) [107] is a technique based on Bloom filters that also adopts dual-entry ACLs from LH. HBA uses two levels of Bloom filter arrays on each MDS to indicate the destination MDS of a file metadata. Although it makes some optimizations to minimize migrations due to renames, HBA needs a hash recalculation and broadcast among the cluster to update the arrays. Besides, HBA needs a large amount of memory to store the arrays.

Like MHS and DOIDFH, FPFS uses a directory hashing [34, 93] approach, preserving the locality at a directory level, and keeping cache directory's contents on a single server (at least, for small directories). However, different to MHS and DOIDFH, which use a global directory identifier assigned on creation time, FPFS assigns identifiers to directories by applying a hash function on the dirnames, so any member of the cluster can independently calculate the ID without extra conversion tables. FPFS also adopts some LH's techniques, like pathname hashing to distribute metadata, dual-entry ACLs, and lazy migrations, although they are only applied on directories. This is an important difference with respect to file hashing techniques, since a rename does not produce a massive migration of file data; only directory objects are migrated. Permission changes do not produce a massive update of files' ACLs either, because a file's permissions are directly derived from its own ACL and its directory's. FPFS also uses a hashing function [96] that minimizes metadata migration on cluster changes, and handles links in a more straightforward and efficient way.

4.1.2. Metadata storage

Besides namespace distribution, another important issue regarding the metadata management, is the location of directories and object's attributes in an object-based parallel file system.

Ceph [96] stores everything in the objects located in the OSDs, including directories. However, the metadata cluster is in charge of the metadata management. Ceph uses embedded inodes [40], thus directories also contain the attributes of their files.

PanFS implements a different approach that stores file attributes in the first two data objects (that are mirrored) of a file. Similar to Ceph, PanFS stores directories as objects (mirrored in RAID-1) like special files that keep an array of directory entries.

Ali *et al.* stores directories as OSD empty objects, and directory entries as attributes of those objects [12]. The implementation stores entries of a directory within a *page* of its empty object by hashing their names, which entails name collision problems. Meanwhile, file attributes are stored as attributes of the first stripe of data.

Unlike these approaches, FPFS includes a different type of object, *directory object*, to store and handle directories within OSD+s. OSD+s take advantage of the features of the underlying file systems, avoiding the insertion of new data structures and software layers, and implement directory objects as regular directories in the backend file system (see Chapter 3).

4.2. FPFS metadata distribution

FPFS distributes directory objects (and so the file-system namespace) across the metadata cluster to make metadata operations scalable with the number of OSD+s, and to provide a

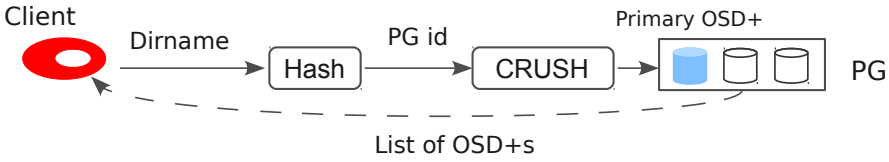


Figure 4.1: How to determine the location of a file in FPFS.

high performance metadata service [17]. Currently, distribution uses a deterministic pseudo-random function called CRUSH [97], which guarantees a probabilistically balanced distribution. However, other functions are also possible such as RUSH [50] (a family of algorithms from which CRUSH is based on), or Random Slicing [66].

As shown in Figure 4.1, CRUSH receives an integer as input (a sort of *id*) that, in our metadata cluster comes from hashing the full pathname of a directory object. Given a directory object, CRUSH outputs its *placement group* (PG), which is a list of devices made up of a primary node and a set of replicas. The devices in a PG are chosen according to device weights and placement rules that restrict the replica selection across failure domains, avoiding potential sources of failures and load imbalance.

Any party in the system is able to independently calculate the location of any directory object. CRUSH only needs a *cluster map* to compute the result, and that map is available for all the nodes in the cluster.

Although FPFS scatters directory objects across the cluster, FPFS also maintains the directory hierarchy of the parallel file system. Remember that each directory object keeps an entry for every subdirectory and file that it contains, if any (see Section 3.4).

Hash partition strategies present different scalability problems on cluster resizing, permission changes, and renames. FPFS addresses these issues through CRUSH and lazy techniques. FPFS lazily migrates directory objects in case of renames, and lazily updates permissions in case of permission changes, similar to LH (see Chapter 3). Nevertheless, note that, in our case, renames and permission changes only affect directories, and different I/O traces and studies show that these operations are infrequent for directories [21, 47, 84]. Also, opposed to LH, FPFS does not need the *Metadata Look-up Table* because CRUSH minimizes itself migrations on cluster changes (falls, additions, deletions, etc.). Therefore, the combination of the mentioned lazy techniques and CRUSH in FPFS, further minimizes the impact of these operations on the metadata cluster performance.

4.3. Symbolic Links

Placing directories by hashing their pathnames presents the problem of locating the correct OSD+s for paths that include soft links: any access to a subtree of the linked directory hierarchy will fail, since those objects do not exists. Note that this scenario does not appear with hard links, given that hard links between directories are not possible.

LH proposes the creation of shortcuts to deal with files whose pathnames contain symbolic links. A shortcut to one of these files is created the first time the file is accessed by traversing the directory hierarchy. Any subsequent access to the same file uses the shortcut. This approach, however, presents a problem when the access to a file fails, since there is no way to know if the failure is due to a missing file or the existence of symbolic links in the name.

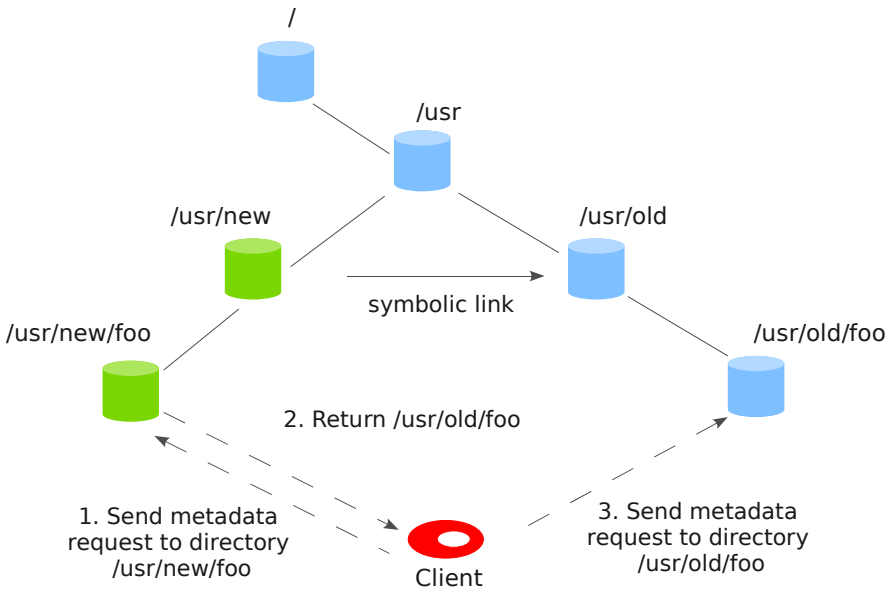


Figure 4.2: Access to a directory containing a symbolic link. `/usr/new` is a soft link to `/usr/old`.

The ambiguity in the latter always produces the traversal of the directory hierarchy up to the root directory when accessing to any missing file.

Our proposal to tackle with symbolic links is simpler than LH's, and does not suffer the hierarchy traversal for missing files. In FPFS, a symbolic link is treated as a directory rename, since the scenarios they create are similar, but not the same. In both cases, clients correctly request objects that happen to be stored on a different OSD+ due to a rename or a symbolic link. In the case of a renamed directory, the request fails because the object has not been migrated to the new OSD+ yet. In the case of a symbolic link, the requests fails because the link is just a reference to an object in a different path. As with renames, OSD+s store any symbolic-link operation in the metadata log (see Section 3.4.3) to later recognize when a client is accessing a path with a symbolic link. This way, the client can be forwarded to the right OSD+ (see Figure 4.2). However, the way to proceed in the case of symbolic links is not the same as in the case of directory renames. The differences are that: (a) any access to a directory whose pathname contains a symbolic link never produces the migration of the directory, and (b) a client accessing one of these directories receives the resolved path to contact with the original OSD+.

Note that, apparently, it would not be necessary to store symbolic links to files, since those links are always accessed through the directory objects containing the links. However, there are scenarios where a symbolic link to a file can end up being a symbolic link to a directory, therefore presenting the aforementioned problem. For instance, if a linked file is deleted, and then someone creates a directory with the same pathname as the deleted file, the link no longer points to a file but a directory. Therefore, symbolic links to files also have to be stored in the metadata log.

Fortunately, symbolic links are not very common, so they do not take up much space in the metadata log. For instance, in the benchmark pnnl-nwfs [25], symbolic links represent only 0.1% among all files in the system.

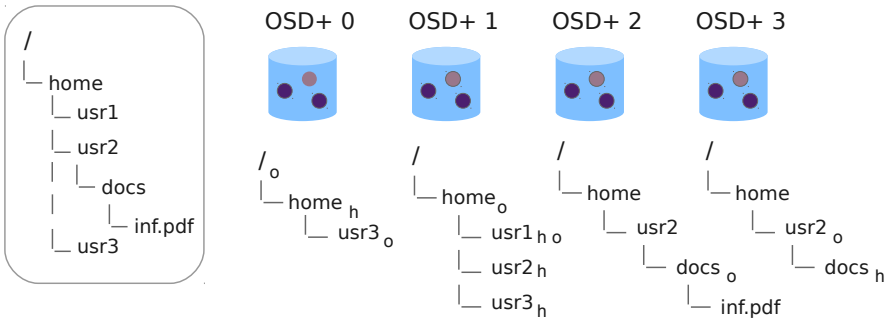


Figure 4.3: Example of mapping a FPFS file system to an OSD+ cluster.

4.4. Atomicity

Parallel file systems must assure its consistency by guaranteeing that all metadata operations are atomic. In FPFS, when a single OSD+ carries out a metadata operation (e.g., `creat`, `unlink`, etc.), the backend file system itself ensures its atomicity and POSIX semantics.

Nevertheless, there are operations that may involve two or more OSD+s, such as `mkdir`, `rmdir` and `rename`. For instance, on a rename, when the source and destination parent paths are different, the hash function will probably place them in different OSD+s. For those operations, we use a *three-phase commit protocol* (3PC) [86] to guarantee atomicity.

The three-phase commit protocol proceeds as follows. Initially, the coordinator checks whether the operation can be performed by asking the participants. Next, the coordinator verifies all nodes are ready to commit the operation. In case all are prepared, the coordinator finalizes the protocol sending a commit message. After this last step, the transaction will not be aborted; although the participants should acknowledge the commit message, the operation will be committed anyway.

In FPFS, the coordinator also acts as participant: apart from coordinating and checking the participants, the coordinator itself also performs operations. For example, in Figure 4.3, a client requested the creation of the directory `/home/usr2` to OSD+ 1, which contained the directory object $\texttt{/home}_\texttt{o}$. That OSD+ acted as coordinator and initially created the directory entry $\texttt{/home/usr2}_\texttt{h}$. If the creation was successful, it would contact the participant OSD+ 3 to complete the request, creating the directory object $\texttt{/home/usr2}_\texttt{o}$. In case the creation of $\texttt{/home/usr2}_\texttt{o}$ failed, the coordinator would conclude the operation removing the directory entry $\texttt{/home/usr2}_\texttt{h}$ and returning an error.

4.5. Client interaction

All members in the cluster can contact any directory object by means of its pathname and the cluster map. FPFS establishes communications between clients and OSD+s via TCP/IP connections and request/reply messages. Each OSD+ launches one thread to attend the requests of a client, and to perform operations on the local disk on behalf of the client. This way, servers reflect the workload generated by clients and can be configured accordingly.

Note that file systems like Lustre limit the number of threads on the metadata server [87], and that this number is usually much smaller than the number of clients; this decision distorts the clients' workload that the server sees, but it can improve the performance in some cases.

FPFS clients use a library, similarly to PVFS2/OrangeFS implementation. As requests, the library supports the most frequently used metadata operations (see Table 4.1 in Section 4.7.2): `mkdir`, `rmdir`, `opendir`, `readdir`, `create`, `unlink`, `open`, `close`, `lookup`, `stat`, `utime` and `rename`.

When an operation involves several OSD+s, the OSD+ contacted by a client carries out the operation transparently to the client. That OSD+ collaborates with other OSD+s by means of the aforementioned three-phase commit protocol (see Section 4.4) to perform the operation.

File systems must also prevent clients from doing malicious operations on the system. The current implementation entirely runs in user-space for fast prototyping and evaluation. However, in a production system, the client side of the file system should be implemented inside the kernel, and applications should access the cluster file system through the VFS interface. Kernel module and authenticaction remains as future work. Authentication of clients against servers should occur at mount time. To this end, mechanisms as Kerberos [70], or that described in the OSD standard [53], can be used.

4.6. Fault Tolerance

Our metadata cluster does not provide fault tolerance yet. In the future, a replication model will be necessary to protect the system against data loss. There exist several approaches that could be used. One approach is RADOS [99], which we have already described in Section 3.2. RADOS allows OSDs to operate with a relative autonomy when it comes to recovering from failures or migrating data in response to a cluster expansion. Another approach is Elliptics [73], a fault tolerant distributed key/value storage. Elliptics allows to distribute data across multiple replicas spread over several different physical location. In this context, "replica" is a logical group of one or more servers which form a *distributed hash table* (DHT) ring. Elliptics also automatically redistributes data in case of removed or added nodes.

In the case of FPFS, the distribution function that it uses, CRUSH, is able to return a set of nodes among which replicate objects. The size of this placement group depends on the chosen replication level. Therefore, given that we already have a way to get replication groups, two approaches are possible to perform replication.

First option is a RADOS-like approach, where the OSD+s drive the replication by themselves. The placement group that CRUSH returns is composed of a primary node and the replica nodes. Figure 4.4 shows how OSD+s would perform the replication of directory objects. Initially, a client would send a request to the primary OSD+ (in purple); that OSD+ would lead the replication by sending the client's request to the replica OSD+s (in orange). In case a node fails, recovery would also be performed by the placement group, updating the missing objects to fail nodes.

The second option is to delegate the replication to clients. Figure 4.5 depicts this option, where each client would send a request to all OSD+s in the same placement group. The request finishes when all OSD+s acknowledge the operation. In this case, servers would know nothing about the replication. This approach is more similar to PanFS's [101].

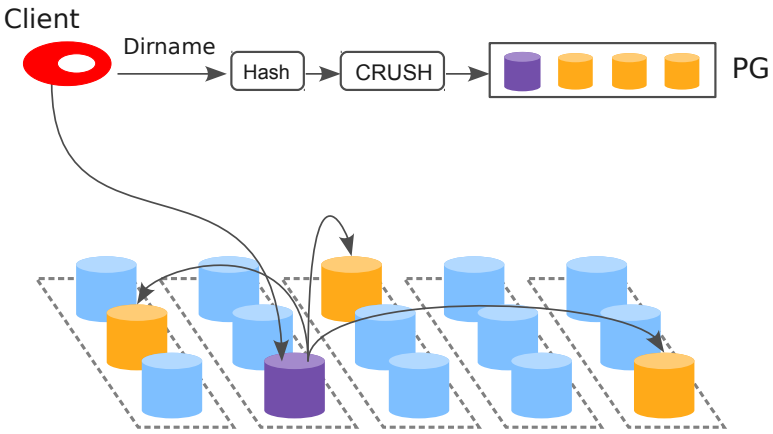


Figure 4.4: Replication handled by the OSD+s themselves.

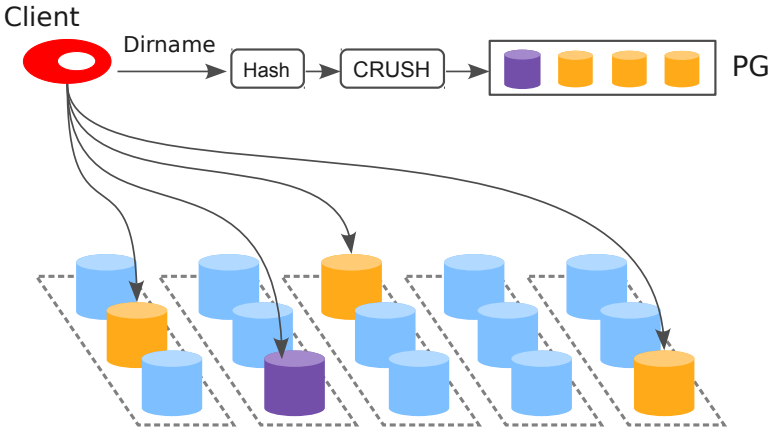


Figure 4.5: Replication handled by the clients.

4.7. Experiments and Methodology

In order to analyze the performance of the metadata cluster of FPFS, we run different benchmarks, and compare FPFS' performance with Lustre's. This section describes the system under test and the benchmarks run to carry out the analysis.

4.7.1. System under Test

The testbed system is a cluster made up of 16 compute and 1 frontend nodes. Each compute node has two Intel Xeon E5420 Quad-core CPUs at 2.50GHz, 4GB of RAM, and two Seagate ST3250310NS disks of 250GB. On each node, one disk has a 64-bit Fedora Core 11 distribution which supports Lustre version 1.8.2. We export the other disk, which is used as test disk, as either an FPFS OSD+ or a Lustre MDS–MGS/OST server. The interconnect is a Gigabit network with a D-Link DGS-1248T switch.

Several issues regarding Lustre and FPFS should be remarked. Although Lustre 2.0 existed at the moment of performing theses experiments and included new functionality to support a metadata cluster, a production-ready service was not available yet. We did not found

information to set up the service either. Lustre 2.0 also supported several file systems as backend, although only a customized Ext3 file system [90] ("ldiskfs") was present. Finally, we ran the tests on the latest version 2.0.0.1, but the results were generally worse than in 1.8.2, so they are not presented here.

Since Lustre ldiskfs can be considered as something between Ext3 and Ext4 [63], FPFS is evaluated using both file systems as backend. This way, we show that the improvements achieved by FPFS over Lustre are due, in many cases, to the smaller overhead and better performance provided by the OSD+ and metadata cluster implementation in FPFS. The use of different file systems, however, has also showed that each file system works better for specific workloads, although FPFS can easily be configured to use the proper file system for a given workload.

Regarding configuration issues, we have checked that the number of i-nodes in Lustre is large enough, so the number of i-nodes used never exceed half of the total (as referenced in the Lustre operations manual [87]). We have not performed any testing to determine the optimal number of MDS threads, so the default value is chosen.

Since the metadata performance depends on the formatting options, FPFS is formatted with the same options Lustre uses in its underling file system. Specifically, the backend Ext4 file system is formatted by executing the following command:

```
/sbin/mke2fs -t Ext4 -b 4096 -J size=400 -i 4096 -I 512 -O dir_index,extents,\
                                uninit_groups <device>
```

These format options set the journal size, i-node size, bytes-per-inode ratio, use of hashing in directories, use of extents and uninitialization of some data structures, respectively.

Finally, we have also tried to evaluate the latest version of the Ceph's metadata cluster [96], but different problems have prevented us from succeeding: an excessive memory use that produces swapping for some workloads, frequent kernel panics, and a poor performance in many cases.

4.7.2. Benchmarks

We use the following tests to evaluate and compare the performance of FPFS and Lustre on metadata workloads:

- **HP Trace**: it is a 21-hour trace collected in 2002 which is, in turn, a subset of a 10-day trace of all file system accesses done by a medium-sized workgroup using a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage space [47]. The trace was collected by instrumenting the kernel to log all file system operations at the syscall interface [77]. Specifically, the selected period, which is one of the most active, covers from 6am on the fifth trace day to 3am on the next day. Table 4.1 shows an overview of the metadata requests in the trace. Since we are only interested in metadata operations, we omit data operations (mainly, `read`, `write` and `mmap`).

  Firstly, we replay the trace rebuilding the directory hierarchy that must exist at the beginning of the trace. Secondly, we replay the full trace. To rebuild the hierarchy, we collect the information from the trace itself. Specifically, from the successful operations present in the trace. Obviously, files and directories created in the trace are dismissed.

Table 4.1: Overview of the 21-hour HP trace (metadata operations)

Operation type	*Count*	*%*
Lookup	13,908,189	71.55
Stat	2,827,387	14.54
Open	2,572,124	13.23
Unlink	67,883	0.34
Create	41,755	0.21
File rename	7,683	0.04
Mkdir	7,389	0.04
Rmdir	6,973	0.04
Directory rename	5	0.00
Total	19,439,388	100.00

Note that the initial recreated directory hierarchy is smaller than the original one in the real file system. However, our aim is to compare FPFS and Lustre with an actual trace, not to exactly recreate the original environment.

We replay the trace with a multithreaded program that allows us to simulate a system with concurrent metadata operations. The program takes into account possible dependencies between those operations.

- **Creation/traversal of directories**: this benchmark is made up of two tests: the first one creates directory hierarchies with empty regular files, and the second one traverses those hierarchies. The benchmark creates every directory hierarchy uncompressing the Linux kernel 2.6.32.9 source tree. All files in the source are truncated to zero bytes as we are only interested in metadata operations.

  In these tests, each process accesses its own copy of the Linux source tree.

- **Metarates** [91]: this one evaluates the rate at which metadata transactions are performed. It measures aggregate transaction rates when multiple processes (coordinated by MPI) read or write metadata concurrently.

  Our experiments uses 640,000 files in total, distributed into as many directories as processes (each process works with files in a unique directory). The program tests the performance achieved by each system for three types of metadata transactions[1]: create-close (in our tests, without calling `fsync` before closing a file), stat, and utime calls, which basically generate a write-only, read-only and read-write metadata workload, respectively.

[1] Note that the same create-close and stat metadata workloads can be generated by more up-to-date benchmarks like mdtest [68]. However, unlike mdtest, metarates also supports utime operations, which read and write the same metadata element (an i-node in our case) in each transaction.

4.8. Results

This section evaluates the performance of FPFS through the prototype built on a Linux environment (see Section 3.2).

Experiments use up to 8 out of the 16 nodes. For FPFS, we set up two configurations: one with 1 OSD+, and another with 4 OSD+s. For Lustre, we only set one configuration with 1 node running all its services (MGS/MDS, and one OST), equivalent to our FPFS configuration with one OSD+, where the single server stores both directory objects and empty data objects. This version of Lustre does not support several MDSs so we cannot set a 4-node configuration in Lustre that would be the equivalent to ours with 4 OSD+s.

For clients, we use 1 to 4 nodes depending on the test. Since we have not detected either a CPU or network bottleneck in the clients during the experiments, we run several processes per CPU and core (up to 256 in total, that is, 64 clients per node) to analyze the servers' performance under heavy workloads. The number of client processes per benchmark varies from 1 to 256 processes, in powers of two.

Figure 4.6 depicts the improvement (in percentage) of FPFS with 1 OSD+ over Lustre containing both an MDS/MGS and OST service, considering the operations/s achieved by each file system, for the three benchmarks. The graphs show the percentage of improvement of FPFS for the different number of clients.

Figure 4.7 shows FPFS scalability for 1 and 4 OSD+s and the same number of clients as the previous graphs.

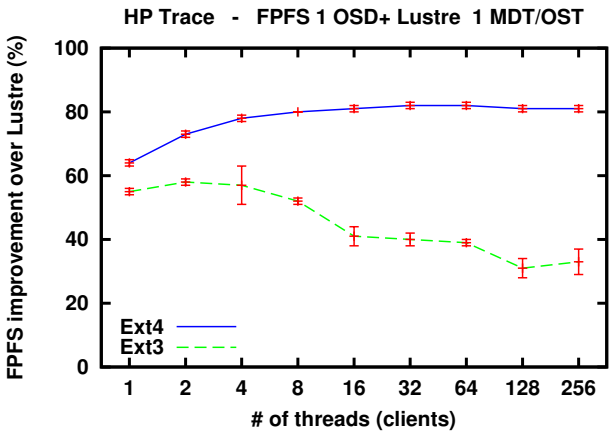
The results shown for every system configuration are the average of at least five runs of each benchmark. Confidence intervals are also shown as error bars, for a 95% confidence level. We format the disk between runs, and unmount/remount between the directory tree creation and traversal tests.

4.8.1. HP Trace

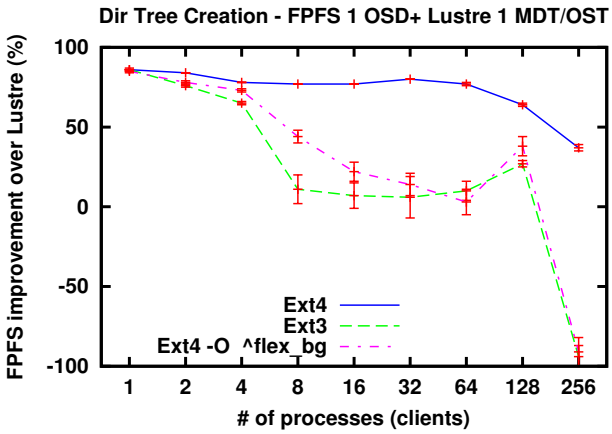
Figure 4.6.(a) shows the improvement of FPFS over Lustre in operations/s for the HP Traces benchmarks. Although Lustre is a full-fledged parallel file system and FPFS only implements an incomplete metadata service, both roughly perform the same operations. This fact, along with the large performance differences in this test, which reaches 82% for 16/32 threads and Ext4, ensures that FPFS represents a significant improvement with respect to Lustre in time-sharing environments.

Moreover, FPFS outperforms Lustre regardless of the backend file system used. This is mainly due to the thin layer FPFS adds on top of the backend file system which directly translates FPFS requests into backend file system requests, producing little overhead. Instead, Lustre adds several abstraction layers.

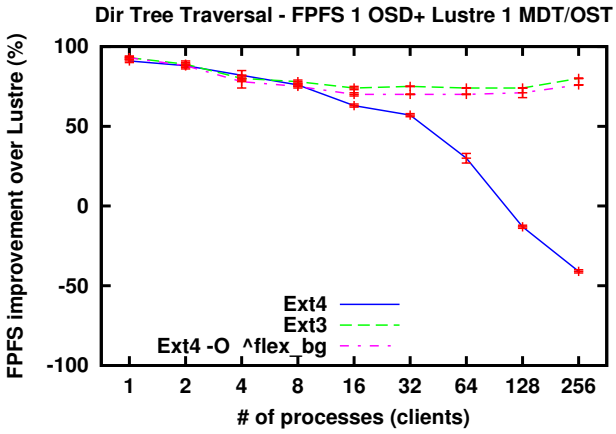
Figure 4.7.(a) shows FPFS scalability that reaches 3.04 for Ext4 and 3.70 for Ext3, when there are 256 threads. This value is smaller than the ideal 4, due to the dependencies between the metadata operations in the trace, which limits the parallel execution of operations. However, as the number of threads increases, so does the number of possible ongoing metadata operations. Accordingly, scalability is better for a large number of threads, showing that FPFS can properly deal with large time-sharing systems.



(a) HP Trace

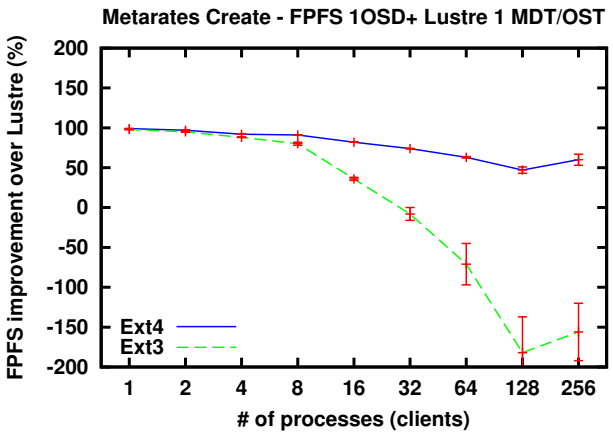


(b) Creation of directories

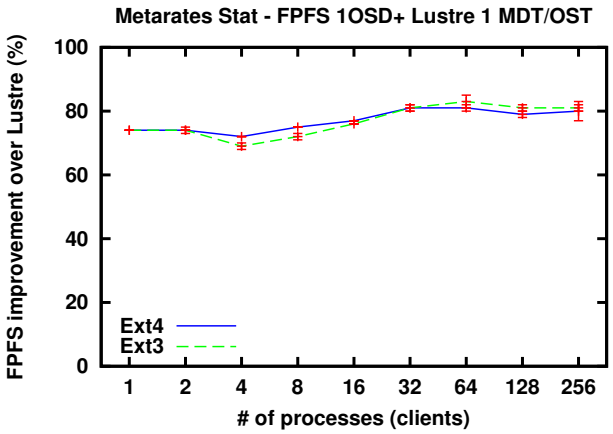


(c) Traversal of directories

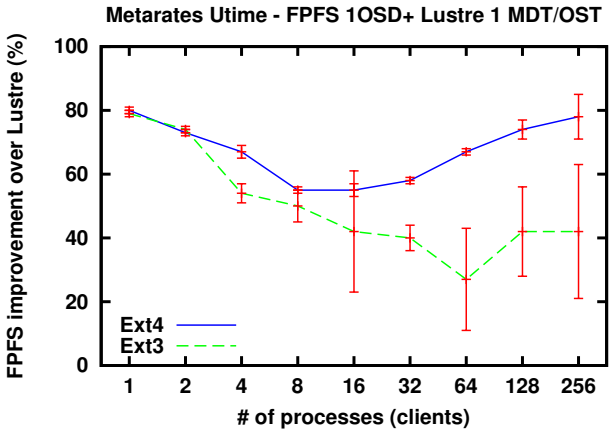
Figure 4.6: Improvement obtained by FPFS 1OSD+ over Lustre.



(d) Metarates: create-close transactions

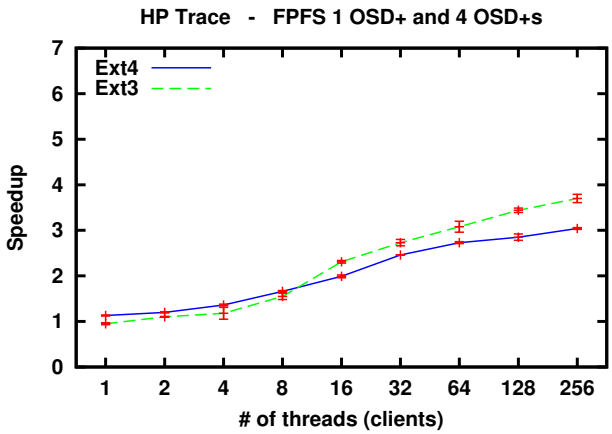


(e) Metarates: stat transactions

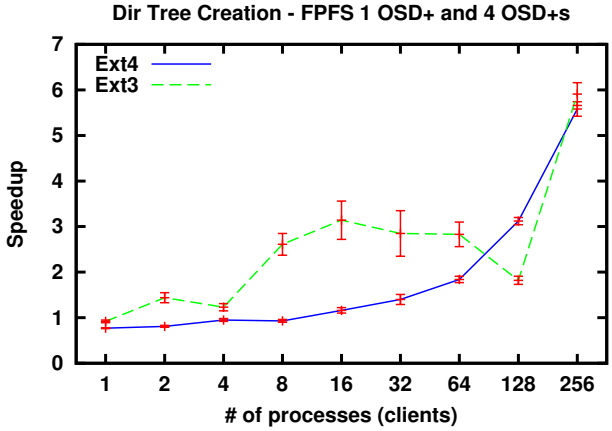


(f) Metarates: utime transactions

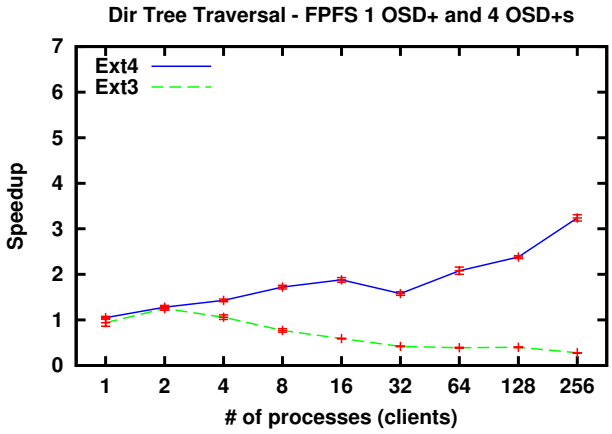
Figure 4.6: (Cont.) Improvement obtained by FPFS 1OSD+ over Lustre.



(a) HP Trace

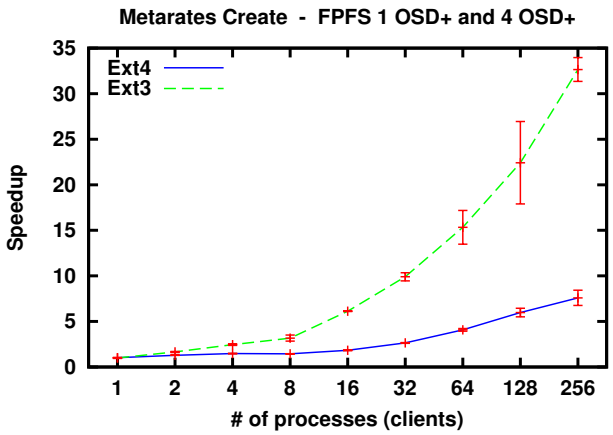


(b) Creation of directories

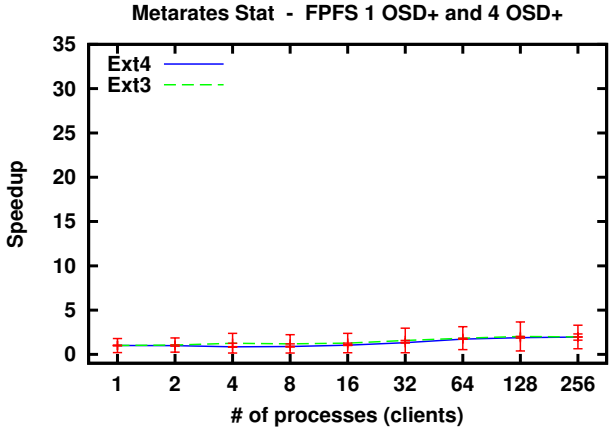


(c) Traversal of directories

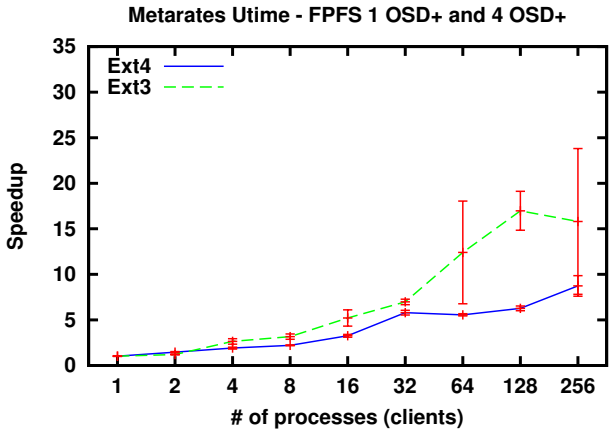
Figure 4.7: Scalability for FPFS 1 OSD+ and 4 OSD+s configurations.



(d) Metarates: create-close transactions



(e) Metarates: stat transactions



(f) Metarates: utime transactions

Figure 4.7: (Cont.) Scalability for FPFS 1 OSD+ and 4 OSD+s configurations.

4.8.2. Creation/Traversal of Directories

Like HP Trace, this benchmark creates/traverses thousands of files and directories (the Linux source tree used by each process has around 14,000 directories and 50,000 regular files). However, unlike the previous one, there are not dependencies between metadata operations carried out by different processes. Another difference is that not all the file system operations are needed (specifically, only the `create`, `opendir`, `close`, `mkdir` and `getdents` metadata operations are used), since we are just creating and traversing a hierarchy.

Figures 4.6.(b) and 4.6.(c) show, respectively, that the improvement in operations/s of FPFS over Lustre can reach 86% during the directory tree creation, and more than 90% for the directory tree traversal. However, results also show that performance depends significantly on the file system used and the number of processes.

In the tree creation test, Ext4 reduces the application time around 70%. Although, Ext3 initially improves Lustre too, from 64 processes on, the improvement drops even reaching negative results over Lustre. The opposite happens in the path traversal test. Ext3 improves Lustre around 70% to 80%, while the improvement for Ext4 drops sharply for 64 processes or more.

The different behavior of Ext3 and Ext4 is due to an exclusive Ext4's option, *flex_bg*, used by default when the file system is created. By tightly allocating bitmaps and i-node tables close together, this feature allows to build a large virtual block group that gets around some of the size limitations of regular block groups [56]. This flag significantly improves the directory creation for any number of processes, since it allows to create bigger block groups allocating more blocks sequentially. However, it downgrades the directory traversal for more than 64 processes since, as groups are larger, seeks between groups will also be larger. However, when *flex_bg* is unset, Ext4 roughly behaves as Ext3. Hence, in these tests, the underlying file system and formatting options can be decisive. The flexibility of FPFS allows to easily set up the system to obtain the best performance.

The scalability achieved in both the directory tree creation and traversal for Ext4 increases with the number of clients (see Figures 4.7.(b) and 4.7.(c)). It is greater than 4 for the creation test, which can be explained by looking at Figure 4.6.(b): with 256 clients and 4 OSD+s, every OSD+s is serving around 64 clients, and the performance of 1 OSD+ for 64 clients is much better than for 256 clients.

For Ext3, the scalability can also be quite good for the directory tree creation, but results for the traversal test are rather bad. We have not found a plausible explanation yet.

Regarding these results, where the performance downgrades for a given number of processes for Ext3 and Ext4 (depending on the test), it is pending to analyze if limiting the number of threads (as Lustre does) would improve performance, instead of having one thread per client, as we do now.

4.8.3. Metarates

The improvement of FPFS is large for a few processes, and decreases with the number of processes for the create test, as Figure 4.6.(d) shows. One of the reasons is that FPFS makes better management of large directories. Although the number of files is fixed, there are as many directories as processes are run, so the lower the number of processes, the greater the number of files per directory. Another reason is the backend file system. Note that this test

is similar to "Creation of a Directory Tree" (see Section 4.8.2), so the chosen backend system is significant. Hence, Ext3 obtains a much worse performance than Ext4 for a high number of clients on creation workloads, due to missing the flag *flex_bg*.

Figures 4.6.(e) and 4.6.(f) show the percentage in operations/s for the stat and utime tests. These great results (specially for Ext4) are due to the initial creation of files, and later execution of the operations. Accordingly, thousands of i-nodes and directory entries are already in the operating system's caches. Hence, the performance is limited by CPU and network bandwidth, and not by hard disks or directory sizes. Lustre's abstraction layers introduce a larger overhead in these cases, reducing the final throughput.

FPFS scalability is super-linear in create-close and utime tests (see Figures 4.7.(d) and 4.7.(f)), mainly due to the system's write-back caches, and the number of processes: when the number of processes is high, the increase of OSD+s reduces the application time, reducing the number of write operations to disk during the tests. The total cache size provided by four OSD+s also decreases the number of metadata reads from disk, which also improves the utime transactions. All this explains the big confidence intervals for utime too, because the amount of metadata written to disk greatly varies from run to run, and so does the application time.

In the stat test (see Figure 4.7.(e)), the scalability slightly increases with the number of clients, although it is clear that, with a single OSD+, the clients almost achieve the maximum possible performance.

4.9. Conclusions

In this chapter we have presented the design and implementation of a metadata cluster, based on OSD+ devices, for our FPFS file system. Metadata is managed by all the servers in the cluster, improving the performance, scalability and availability of the metadata service.

In such large metadata clusters, issues like directory distribution for load balancing, and atomicity of metadata operations are key. We face distribution by uniformly sharing out directory objects among all the servers. We guarantee atomicity of metadata operations involving several OSD+s through a three phase network-commit protocol, and by the local file system itself in each OSD+ for operations on a single directory.

FPFS distributes directories by hashing their pathnames. Hash distributions have to deal with data migrations due to renames and path traversals provoked by links. We manage rename operations in a lazy fashion and, given that, in our case, only affect directories, we highly reduce the number of migrations. Postponing migrations also allow us to delay the movement of metadata, avoiding overflooding the system. Besides, we manage links similar to rename operations, so no extra mechanism is needed, and we avoid path traversals with a few messages.

We evaluate the scalability of the metadata cluster, and compare its performance with Lustre's. The results show that a metadata cluster with a single OSD+ can improve the throughput of a Lustre metadata server by more than 60–80%, and that its performance scales with the number of OSD+s.

Chapter 5

Huge directories

A growing amount of metadata is not the only problem file systems need to face. Another related problem is the increasing use of huge directories with millions or billions of entries accessed by thousands of clients at the same time [14, 19, 39, 72]. This scenario appears, for instance, for data-intensive parallel applications that create a file per thread/process [35, 71] and for applications that use a directory as a light-weight database (e.g. check pointing) [76].

As we have seen, to deal with a large number of files, some parallel file systems use a small cluster of metadata servers [85, 97], while others expect to provide a similar service shortly [29, 82]. However, only a few parallel file systems provide (or plan to provide) some support for huge directories [29, 97, 105].

In this chapter we show how to integrate the management of huge directories with OSD+ devices in FPFS. Our approach leverages the existing directory objects in OSD+s to dynamically spread a huge directory (*hugedir* for short) across several objects that work independently. This way, we improve the performance and scalability of the file system.

We start this chapter with the related work. Next, we describe the design and implementation details of hugedirs within our metadata cluster, specifying issues like directory object's enhancement, rename management, or client implementation. We continue by evaluating the performance with an extensive set of experimental results using different backend file systems and devices. Finally, we give the conclusions of this chapter.

5.1. Related Work

The management of directories with millions of files, accessed by thousands of clients at the same time, is a problem recently identified in HPC systems by different authors [39, 45, 72, 82, 103]. This section describes some proposals able to manage those directories to some extent.

GPFS [80] distributes hugedirs through extendible hashing to map entry names to directory blocks. Given a directory, they apply a hash function to the entry name in order to locate the block containing the directory entry. They use the n low-order bits of the hash value as the block number, where n depends on the size of the directory. When a block is full of directory entries, extendible hashing splits the block. To calculate the number of the new block, they add a "1" in the $n+1^{\text{th}}$ bit position of the split block's number. Therefore, a large directory in GPFS is, in general, a sparse file with holes in the file representing directory blocks that the extendible hashing technique has not split yet. However, working at a disk-block level basis, and using several locking mechanisms limit the performance achieved by GPFS for some directory-related operations, including those on large directories [14].

Boxwood [61] also supports hugedirs by means of B-link trees, a sort of B*-tree where each node has a link that points to the next node of the tree at the same level as the current node.

Each tree is distributed among several servers. Nevertheless, since Boxwood relies on a global lock service for synchronized metadata accesses, it lacks the ability to effectively deal with a concurrently-accessed directory. Unlike GPFS and Boxwood, OSD+s supporting a hugedir in FPFS basically work independently, achieving good performance and scalability.

GIGA+ [72] is a POSIX-compliant scalable directory design that distributes directory entries over a cluster of server nodes. This technique incrementally hashes a directory into a growing number of partitions, which are migrated among metadata servers for load balancing. Severs individually perform migrations, without a system-wide serialization, synchronization or notification. GIGA+ also prohibits migrations if load balancing is unlikely to be improved. Unlike GIGA+, FPFS splits a directory just once; a directory can also be shared out right from the start. These approaches are more efficient for directories expected to be huge [104]. Also, an FPFS client discovers that a directory is huge on the first request, while a GIGA+ client may need several probes at different moments before addressing the correct server. Finally, GIGA+ uses a small set of metadata servers, while FPFS uses all the available OSD+s to handle hugedirs for better throughput.

In Ceph [97], each metadata server keeps a record of the popularity of metadata within a directory, and adaptively hashes a directory when it gets too big or experiences too many accesses. However, as our experimental results show, continually hashing/unhashing a directory depending on the workload or size can be problematic. Like GIGA+, but unlike FPFS, Ceph uses a small cluster to manage metadata operations, and this limits its throughput.

Shvachko [83] proposes the use of HBase for maintaining the HadoopFS namespace, making HBase a scalable replacement for the NameNode of HadoopFS. To partition the namespace, Shvachko analyzes several existing approaches such as hashing of file paths, Ceph-like partitioning, and fixed-height tiles. In theory, some of these approaches (e.g., hashing of file paths) can be used for splitting a big directory into several servers. Unlike FPFS, no concrete implementation exists yet.

Finally, OrangeFS (a branch of PVFS) [105] merges ideas from extendible hashing [33] and GIGA+ [72] to distribute directories. When creating a directory, they also allocate an array of *dirdata objects*, with each object on one metadata server. Then, they spread directory entries across the different dirdata objects. Unlike GIGA+, which always starts from a single partition and increases the number of partitions gradually, the initial number of active dirdata objects is configurable. The authors claim that the splitting process is expensive [104] and, for a directory that is expected to be large, it is better to use all the dirdata objects to get better scalability from the start. Lustre [29] has proposed a similar approach where directories are statically striped over several MDTs.

Our proposal for managing hugedirs in FPFS is similar to that proposed for Lustre and OrangeFS, although, unlike them, it is dynamic. In FPFS small directories are not initially distributed, and only directories that grow too large get shared out. Our focus, however, is not on proposing new mechanisms for hugedirs but on showing that distributed directories can be efficiently implemented in an OSD+-based metadata cluster. OSD+s are key for FPFS since, unlike data and metadata servers in Lustre and OrangeFS, they add a small-overhead thin software layer. That layer leverages the underlying local file system to provide their services. Thanks to the use of all the OSD+s devices in a cluster, we claim that FPFS can provide better performance for hugedirs accessed by thousands of clients than other parallel file systems.

5.2. Design and implementation

So far, we have assumed that every directory object is handled by a single OSD+. This is probably the most efficient approach for small directories, specially when we use hard disks, given that striping across multiple servers would lead to an inefficient resource utilization. Particularly, directory scans would incur disk-seek latencies on all servers to only read tiny portions [72]. However, since the use of hugedirs is common for some HPC applications, new mechanisms are necessary to deal with them, specially when thousands of clients work concurrently on the same hugedir.

FPFS proposes a dynamic distribution among OSD+s to efficiently manage hugedirs [16]. A directory is considered huge when it stores over a established number of files; once this threshold is exceeded, the directory is shared out among several nodes in the cluster. A directory can also be distributed right from the beginning if the threshold is set to 0.

The OSD+s supporting a hugedir are composed of a *routing OSD+* and a group of *storing OSD+s*. The former contains the *routing directory object* and is in charge of providing clients with the hugedir's distribution information. The latter has the *storing directory objects*, which store the directory's content and are the OSD+s contacted by clients aware of the directory's distribution. The routing OSD+ can also be part of the storing group in case it keeps any directory's content (see Section 5.2.3), it happens as in small directories, where the routing and storing objects are the same.

The routing and storing OSD+s are described through what we call a *distribution list*. The list stores the IDs of the OSD+s that make up the group. It starts with the ID of the routing OSD+, followed by the IDs of the storing OSD+s.

5.2.1. Enhancement of directory objects

In order to support hugedirs, the implementation of directory objects described in Chapter 3 needs to be extended. A directory object can now play two different roles, or both at the same time: the routing function, providing clients with the distribution list, and the storing function, keeping a directory object's content.

As described in Section 3.4, OSD+s store directory objects like regular directories in the backend file systems. The implementation uses different types of directories to maintain the hierarchy, handle lazy renames, etc., which are differentiated through extended attributes. In the same way, the implementation now includes the **r** attribute to mark the routing role, and the **s** attribute to mark the storing role. For simplicity, OSD+s mark directory objects for small directories (which have both roles) with **o**, as previously implemented. Only when a directory becomes huge, they explicitly add the **r**, or **s** attribute, along with the **o** attribute, depending on the object's role.

Figure 5.1 shows the mapping of a directory hierarchy where there are 3 directories, but no hugedirs (note we only mark directory objects with **o**). Next, Figure 5.2 shows the same hierarchy, where directory `usr3` has become a hugedir and has been distributed. We mark the routing object, OSD+ 0, with the **r** attribute and store the distribution list as extended attribute (not shown) to be able to inform clients about the distribution. The storing objects present the attribute **s**. In the example, the routing object, OSD+ 0, is part of the storing-object set since it also stores directory content; hence, it has both attributes, **r** and **s**.

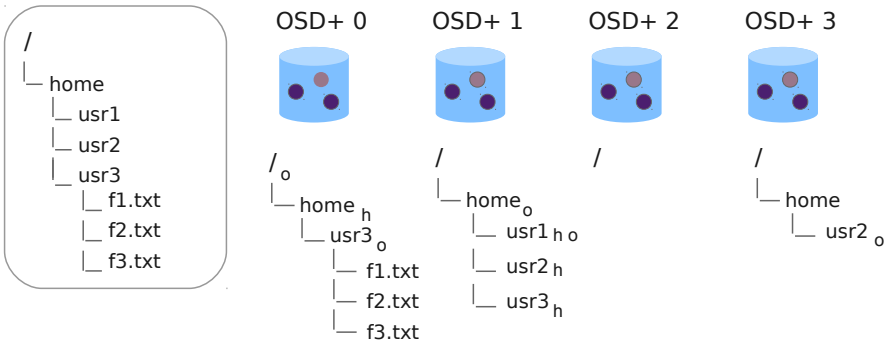


Figure 5.1: Example of mapping a FPFS file system to an OSD+ cluster.

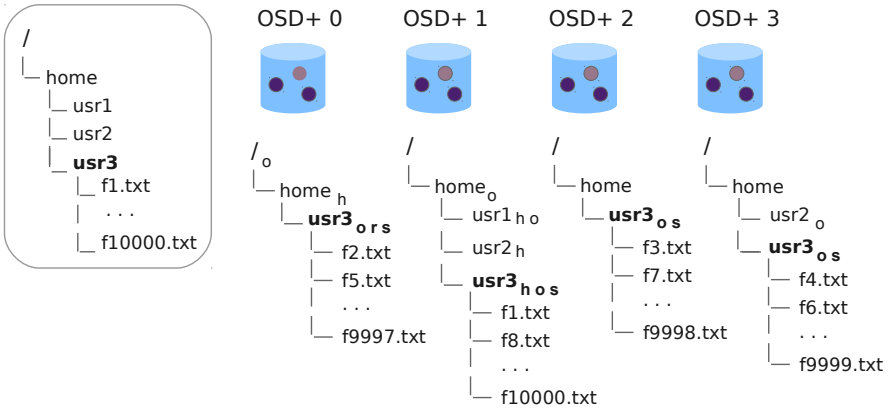


Figure 5.2: Example of mapping a FPFS file system with a huge directory `usr3` to an OSD+ cluster.

5.2.2. Location functions

Hugedirs distribute their entries among several nodes by using their distribution lists and Equation 5.1. We select the OSD+ where to place a file by hashing its name and using the hashing value as index in the *osd_set* list, which is the list of storing objects. The result is the storing OSD+ where the file should be stored.

$$oid = osd\_set[hash(filename)] \tag{5.1}$$

However, the routing object is located following the directory-level hash function used so far for locating the OSD+ ID (OID) of a directory object:

$$oid = CRUSH(hash(dirname)) \tag{5.2}$$

This function is used to place regular directories, as well as to place the routing objects of hugedirs. This way, clients unaware of a hugedir's distribution are informed of the distributed state of the directory. The routing OSD+ notices if the client is aware of the distribution of the directory, and in case it is not, it sends the client the corresponding distribution list. Clients store the list in memory and use it hereafter to calculate the location of the hugedir's entries.

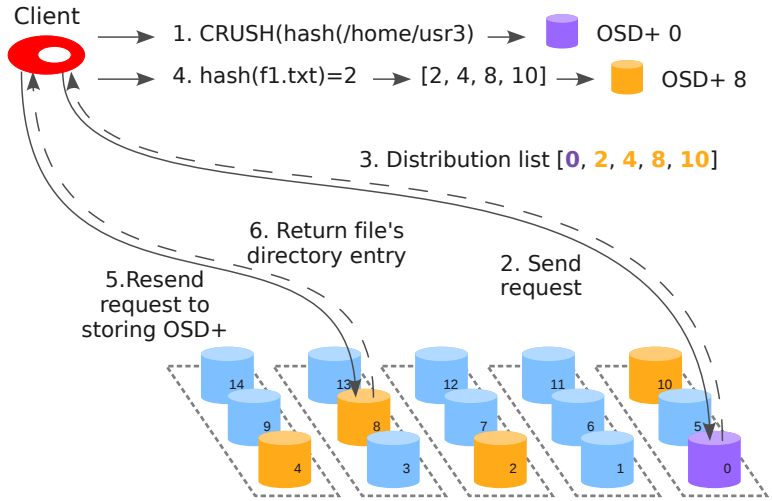


Figure 5.3: Example of a client requesting a file on a hugedir.

To clarify, let see the example in Figure 5.3 where the hugedir `/home/usr3` is distributed among 4 storing OSD+s: 2, 4, 8, and 10. Initially, the client does not know the directory is a hugedir and applies the distribution function in Equation 5.2 to access the file `/home/usr3/f1.txt` (step 1). Once the clients knows the destination OSD+, sends a stat request to the OSD+ 0 (step 2). The routing OSD+ realizes the client does not know about the distribution and replies with the distribution list (step 3). Next, the client calculates the location of the storing OSD+ with Equation 5.1, applying a hash on the file name `f1.txt`. This hashing returns the value 2, which is used as index on the list of storing OSD+s. The returned node is OSD+ 8 (step 4) that stores the file's directory entry. The client sends again the request to OSD+ 8 (step 5), obtaining the status information of the corresponding file entry as reply (step 6).

5.2.3. Renames

The above Equation 5.1 gets rid of a directory's name by using its distribution list instead. This way, we avoid migrating the storing directory objects of a hugedir on a rename; the routing object is the only one that changes its location, since it always depends on its name. Thereby, we get quite fast hugedir renames.

The new routing OSD+ after a rename can be either an OSD+ already in the storing group (Figure 5.4.(a)), or a different OSD+ in the cluster (Figure 5.4.(b)). In the former scenario, the new routing node shares the routing and storing roles, just like the initial routing OSD+ does when a directory becomes huge.

Note that renames of regular small directories (i.e., non-distributed directories) could be performed in the same way. However, we would not save up much time, since there is not a big difference between moving a routing object and migrating a small directory object.

5.2.4. Clients

Clients are not aware of a hugedir's distribution until they issue a regular operation on it. Then, the routing OSD+ sends the distribution list to the clients that cache the list to find

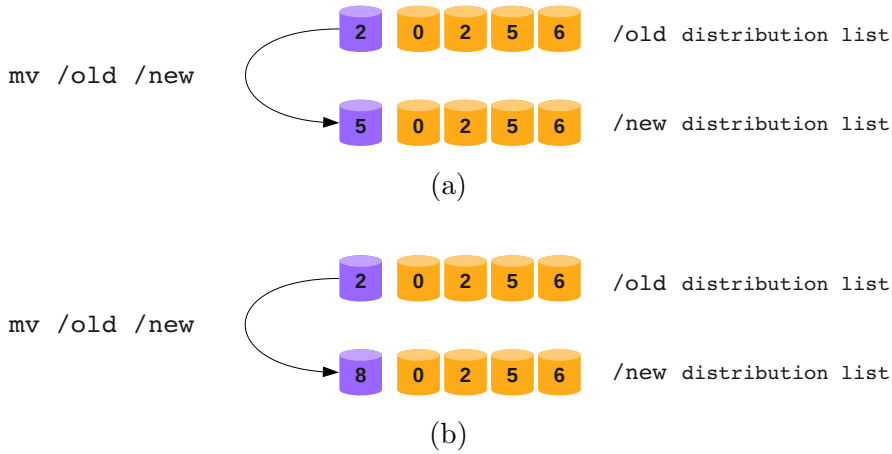


Figure 5.4: Distribution lists after performing a hugedir rename where the new routing OSD+ (see first cylinders) (a) was already in the list and (b) was not part of the list.

the directory's entries hereafter. Once a client knows about the distribution, it will follow Equation 5.1 to further create and access files, improving performance and scalability while preserving POSIX semantics.

Eventually, this cached information may be out of date due to the rename or deletion of a hugedir. Hence, to keep coherency, each directory object stores a *timestamp* with its creation time, and includes it in any message. In turn, clients cache that time to later check whether a hugedir has changed.

Timestamps assure coherency in scenarios where, for instance, we create a new directory with the same name as an old hugedir that was renamed or removed. In such a situation, the OSD+ storing the new directory object acknowledges requests sent by outdated clients, but they will notice that the embedded timestamp is more recent and, therefore, that the directory has changed.

In case a hugedir is just renamed or removed, the directory will no longer exist with that pathname. Thereby, requests sent by outdated clients to the storing OSD+s will simply fail.

Whenever a client detects that the timestamp has changed or that the operation has failed, the client cleans up the cached information for the directory and retries the operation following Equation 5.2. Note that, after retrying, if the directory exists and is huge, the client will receive a new distribution list.

5.2.5. POSIX semantics

In the case of distributed directories, storing objects independently handle disjoint subsets of entries, and no duplicated entries are possible along the objects since Equation 5.1 unequivocally allocates them.

Each storing object has its own size and timestamps, and no information is centralized or updated in the routing OSD+ when we modify a storing object. This way, we prevent the routing object from becoming a bottleneck. This is why POSIX semantics is not maintained by directory objects but clients. When a client wants to know a hugedir's size, it gathers the storing objects' sizes and adds them up to obtain the overall size. Likewise, to obtain the

latest timestamps, the client picks the most up-to-date values among all the storing objects. So it is the responsibility of whoever wants the information to obtain it.

Remember that, in case of non-distributed directories, POSIX semantics are maintained by the underlying file system, exporting them to the above FPFS layer. Since FPFS objects are directly mapped to directories in the local file system (see Section 2.2), this is made straightforward.

This approach of tasking the clients with the burden of getting up-to-date information also applies when a client issues, for instance, an `ls -l` command. On an `ls`, the target OSD+ (or OSD+s in case of a hugedir) sends the directory object's entries to the client. As explained in Section 3.4, directory objects store in the same OSD+ their entries (files and subdirectories) as empty files and directories, mainly to maintain the file system hierarchy. Those entries also store metadata information like permissions or timestamps. However, some of this information (e.g., sizes and timestamps) may be obsolete given that operations on data and directory objects do not usually update their corresponding hierarchy entries. Keeping it up-to-date would require to significantly increase the network traffic (because objects and their directory entries are usually located in different OSD+s), and could make the directory object containing the entries a bottleneck. Both things would downgrade the system's performance. Thus, in case a client is interested in meta-info like sizes or timestamps, it should ask the corresponding objects for that information. Note that if a directory entry corresponds to a hugedir, the client will determine its meta-info from among all the storing OSD+s, as we have already explained.

5.2.6. The distribution process (A directory getting huge)

There are three different states for a directory: "non-distributed", "distributing", and "distributed". The first is the most common state, where all directories storing less entries than the threshold belong to. The second corresponds to the redistribution of existing entries to comply with Equation 5.1; in this state, no requests can be performed on the hugedir. Finally, the "distributed" state is set once the redistribution finishes.

Every OSD+ records its directory activity through an AVL tree. The tree monitors all non-distributed directories accessed by clients until their state changes into "distributing". At that point, the tree blocks any new request on the future hugedir and checks when outstanding requests finish in order to begin the redistribution of existing entries. The redistribution process is led by the node containing the directory's entries, that acts as routing OSD+ and relocates those entries by sending them in parallel to the other storing nodes. As soon as redistribution is completed, all requests received during the process are returned to the clients together with the distribution list, so the requests can be forwarded to the appropriate OSD+s. Once distributed, the tree is also used to reply future incoming requests from clients unaware of the new state of the hugedir with the distribution list.

Note that, we do not erase the relocated the entries from the initial storing object during the redistribution due to the poor performance of the remove operation on the local file systems (see Section 5.3). Instead, they are just ignored: the initial storing object checks whether it is still responsible for an entry before sending it back to a client to avoid incoherent results, such as duplicated entries (e.g., during a hugedir scan). This checking is performed on-the-fly through a hash function, and it is far quicker than deleting files or marking them as "deleted"

with extended attributes. Thanks to this approach, we have speed up the distribution process on hard disks by *an order of magnitude*.

OSD+s mark a hugedir's directory objects internally as either routing or storing objects (or both) by means of extended attributes (see Section 5.2.1), as well as directory states and distribution lists. However, as we have already explained, they also record a directory's state in the AVL tree the first time it is accessed in order to get a faster access.

5.2.7. The refolding process (A hugedir getting small)

When a hugedir's size decreases below the threshold, it is not huge anymore. To re-join a distributed directory, its routing OSD+ would lock the directory and its storing objects, and would lead a process where all storing objects would send back their entries. Afterwards, the storing objects would be deleted and the directory unlocked. But, is the collapse of a hugedir worth this overhead?

Theoretically, only `scandir`-like operations would take advantage of the refolding, because it would avoid reading small objects from several servers. All other operations would perform well, or even benefit from the distribution, if many clients access the directory at a time. So the overhead of collapse is only worthwhile for a specific case, and it can be even useless if a now-small directory becomes huge again.

Therefore, the default policy should be not to collapse a small hugedir. However, we must still provide the operation to allow an administrator to manually join a hugedir if she considers that the performance would improve depending on the workload and underlying storage devices. Moreover, results in Section 5.4.4 show that collapsing small distributed directories could be recommended in some cases.

5.3. Experiments and Methodology

The analysis of hugedirs includes different types of devices and backend file systems. This section describes the system under tests and benchmarks used on the experiments.

5.3.1. System under test

The testbed system is the same used in the previous chapter (see Section 4.7.1). The difference is that, we now use two different storage devices for the test drive in every OSD+: an HDD and a SSD. The HDD test disk is the same as that in the previous chapter, a Seagate ST3250310NS of 250 GB. The SSD test disk is an Intel 520 Series SSD of 240GB.

The default I/O scheduler, CFQ [32], is set for HDDs, whereas Noop is set for SSDs [55], since it usually provides better results for these devices.

Metadata performance depends on the backend file system, as shown in Section 4.7.1, as well as on formatting and mounting options. As backend file systems we use Ext4 and ReiserFS. We use the same formatting options for Ext4 as in Chapter 4. In the case of ReiserFS, we use the option `--journal-size 32749` to set the journal to 32749 blocks (of 4 kB), which is its maximum allowed size when not on a separate device. Mount options are quite similar for both file systems, and try to increase the metadata performance obtained by each one. For Ext4, we use `noatime`, `nodiratime` and `data=writeback`, while we use `notail`, `noatime`

and `nodiratime` for ReiserFS. We have not used the `discard` option in Ext4 for issuing `trim` commands to the SSD-OSD+s since ReiserFS does not support it.

For comparison purposes, we also use two versions of OrangeFS. For distributed tests, we use orangefs-2.8.3-EXP, which does not allow to set a non-distributing configuration. For non-distributed tests, we use orangefs-2.8.3-20110323.

Apart from those systems, as we did in the previous chapter, we tried to evaluate version 0.80 of Ceph (the latest at the time of this writing), but we run into several problems when using more than one MDS: very low performance (about 452, 1960 and 108 empty files created, stated and deleted per second, respectively), MDSs crashes, and memory leaks (each MDS process took more than 6 GB of RAM). When using a single MDS (as recommended in the Ceph documentation) and a single client, performance improved for stat (6667 ops/s), but remained low for create and unlink (about 526 and 294 ops/s, respectively). Moreover, we still experienced some crashes and memory leaks. With one MDS and 256 clients, performance slightly increased for create (741 ops/s), basically remained the same for stat (6256 ops/s), but decreased for unlink (165 ops/s); also, memory leaks were even worse in some cases. Considering these results, we concluded that it was not worth comparing with Ceph.

5.3.2. Benchmarks

The benchmarks we use are the following[1]:

- *Create*: each process creates a subset of empty files in either shared or non-shared directories. Metadata operations used are `fpfs_creat` and `fpfs_close`. This benchmark basically generates a write-only metadata workload.
- *Stat*: each process gets the status of a subset of files in shared or non-shared directories. Metadata operation used is `fpfs_stat`. This is a read-only metadata workload (remember that `noatime` and `nodiratime` mount options are used).
- *Unlink*: each process deletes a subset of files in shared or non-shared directories. Metadata operation used is `fpfs_unlink`. This is a read-write metadata workload.
- *mdtest*: each process creates several sets of empty files in a shared directory. Each creation is preceded by a barrier that synchronizes the progress of the processes. With this benchmark we emulate checkpointing applications.

5.4. Results

In the experiments we evaluate the performance and scalability achieved by FPFS for hugedirs. We use HDD-OSD+s and SSD-OSD+s as backend devices. By using SSD devices, we remove the seek overhead that limits the number of IOPS in hard drives. Also, we compare results of FPFS with those of OrangeFS. With this evaluation, we fulfill a deep study of hugedirs. This study not only shows the good performance of FPFS, but also reveals unexpected but interesting results that provide a better understanding of hugedirs' effect on the system.

Through the experiments, we have analyzed four different aspects of the hugedir support:

(a) Throughput and scalability for a single shared hugedir.

[1]Similar tests can be performed by means of well-known benchmarks such as metarates.

(b) Performance when several shared and non-shared hugedirs are accessed in parallel, and at the same time.
(c) Performance when emulating checkpointing applications.
(d) Performance when there are one shared and one non-shared hugedir accessed concurrently.

As we did in tests in Chapter 4, for clients, we run several processes per CPU and core (up to 256 altogether) to analyze the servers' performance under heavy workloads. For servers, we extend configuration to 1, 2, 4 and 8 servers; those are OSD+s in FPFS, whereas they are set as both metadata and data servers in OrangeFS to make an equivalent comparison.

When using hugedirs, we spread out the directory when its size is greater than 244 kB, which is equivalent to distribute it when the directory has over 8,000 files. We calculate it that way because the `stat` operation does not provide total number of files in a directory. This threshold is based on the observation that 99.99% of directories contain less than 8,000 files [72]. Also, we distribute files uniformly among the OSD+s, so storing directory objects supporting the hugedir are of equal size. The time taken by the redistribution of existing entries is less than 2 seconds.

As earlier said in Section 5.2.6, we do not immediately erase the migrated entries from the routing OSD+ due to the remove operation's bad performance on local file systems. Instead, we ignore them. However, we could delay the removal and definitely erase the entries as the system's load gets low.

Results are the mean of, at least, five runs. We show, as error bars, confidence intervals for a 95% confidence level. We format the test disks between runs for *create*, and unmount/remount then between tests for the rest (except for mdtest; see Section 5.4.3).

5.4.1. Baseline performance

Let us start with a baseline for the performance of various file systems with the *create* benchmark for both hard disks and solid state drives. Table 5.1 compares results obtained by running this test locally on Ext4 and ReiserFS to those obtained on a separate single client and single server instance of FPFS, OrangeFS, HadoopFS, and NFSv3. Except for FPFS, that uses both Ext4 and ReiserFS, the other networked file systems use Ext4 as backend.

For FPFS, the *create* benchmark is linked with the FPFS library that implements POSIX-equivalent file operations. A similar approach is used for OrangeFS and HadoopFS. Local file systems and the NFSv3 client perform file operations through system calls.

As we can see, local file systems deliver high directory insert rates. Any networked file system achieves a modest performance on both HDD and SSD devices, being FPFS the one that obtains the best throughput.

This low performance of networked file systems is mainly due to network overheads. In our experiments, the time taken to exchange 1,600,000 messages of 64 bytes in size between two cluster nodes is 80.66 seconds. This is roughly the same network traffic produced by FPFS with Ext4 and HDD-OSD+s when creating 400,000 files. This creation process takes 109.64 seconds (this time yields 3,648 creates per second, as Table 5.1 shows). Considering that Ext4 needs 22.19s to locally create 400,000 files, we can conclude that FPFS overhead is small ($109.64 - 80.66 - 22.19 = 6.79$s), and its limiting factor is the interconnect.

When a 20Gbps Infiniband interconnect and IP over Infiniband (IPoIB) are used, the exchange of 1,600,000 messages takes 41.98 seconds, almost half the time taken by our Gigabit

Table 5.1: File create rate in a single directory on a single server and 400,000 files in total.

	File system	Creates/s	
		HDD	SSD
FPFS (library API)	Ext4	3,648	3,806
	ReiserFS	3,859	3,811
Local file systems	Ext4	18,023	21,945
	ReiserFS	22,324	24,742
Networked file systems	NFSv3 filer	1,479	1,518
	HadoopFS	262	596
	OrangeFS	253	699

Ethernet. In this scenario, FPFS performance improves by 33%, although its overhead increases a little, up to $74.02 - 41.98 - 22.19 = 9.85$ seconds.

Therefore, with better interconnects, better protocols that generate less message exchanges, and even specialized operations, a networked file system could greatly improve its throughput. For instance, we can improve the performance, without changing the interconnect, by modifying the `open`'s operation. Note that every file creation requires 4 network messages in FPFS: two (request and reply) for an `open` or `creat` call, and another two for closing the returned file descriptor. We could augment the `open` function with a new `O_NONFD` flag. This flag would only make sense along with the `O_CREATE` flag, and it would produce a file creation without returning a file descriptor. This would reduce the amount of network messages by half during creations of empty files.

Regarding SSD devices, performance results for FPFS slightly improve due to the interconnect limitation. In the other networked file systems, depending on how they store data, the SSD devices can significantly improve the throughput, like in HadoopFS and OrangeFS. In any case, the performance is still significantly lower than FPFS performance.

The number of concurrent clients is also important to fully exploit hardware resources (particularly, the interconnect) and increase overall performance. Table 5.2 shows the file create rate achieved by FPFS with a single OSD+ device, and 1, 8 and 64 clients running on the same client node. Results for 1 client are the same as those in Table 5.1. When there are 8 clients, each one creating one-eighth of the 400,000 files in the shared directory, the total file create rate is multiplied by four. Note that, for 64 clients, however, the rate hardly increases with respect to that for 8 clients because the nodes in our cluster have 2 processors with 4 cores each (8 in total) and, with more than 8 clients, CPUs cannot be profited much further.

5.4.2. Single shared huge directory

This section analyzes the performance and scalability on a single shared hugedir concurrently accessed by hundreds of processes to create, get the status and delete files. There are

Table 5.2: File create rate by FPFS in a single directory on a single server and 400,000 files in total, for different numbers of clients.

	File system	Creates/s	
		HDD	SSD
1 client	Ext4	3,648	3,806
	ReiserFS	3,859	3,811
8 clients	Ext4	14,978	17,812
	ReiserFS	17,863	19,047
64 clients	Ext4	15,512	19,058
	ReiserFS	18,620	20,037

256 clients spread across 4 nodes, which work on equally-sized disjoint subsets of the files. In total, the hugedir contains $F \times N$ files, where F is either 200,000, 400,000 or 800,000, and N is the number of OSD+s. Setup $400000 \times N$ is similar to that used by Patil *et al.* [72]. When there is only one OSD+ (N is equal to 1), there is no possible distribution. However, when there are 2 OSD+s or more, the directory is distributed among N servers. The storing list within the associated distribution list will be made up of the IDs of those OSD+s. By changing the number of files per directory, we can also analyze the effect of the directory size on performance and scalability.

HDD-OSD+

Figure 5.5 depicts the throughput in operations/s obtained by FPFS with HDD-OSD+ devices, when 256 clients access concurrently a single shared hugedir. Figure 5.6 shows the speedup achieved for the same test.

The first figure shows that ReiserFS gets better performance during *create* and *unlink* operations, whereas Ext4 is better during *stats*. As we can see, directory size determines the performance obtained by FPFS with Ext4 to a large extent, achieving better performance for smaller objects. Note that clusters of hundreds or thousands of OSD+s are expected, so a hugedir distributed among many OSD+s will typically use small directory objects; this will improve its throughput. Performance, however, hardly depends on the directory objects' size with ReiserFS as backend system.

It is important to realize that, either with Ext4 or ReiserFS, and just 8 OSD+s, FPFS is able to create, get the status of, and delete more than 80,000, 100,000 and 70,000 files per second, respectively. These numbers exceed today's requirement for 40,000 files creates per second in a single directory [71] and prepare FPFS for the Exascale-era.

Figure 5.6 shows that the directory size also determines the scalability reached by Ext4 and ReiserFS. We compute the speedup by comparing the performance obtained when not distributing and when distributing the files in a single shared directory. Note that scalability cannot be calculated from Figure 5.5 since the number of files varies with the number of OSD+s, and we have to compare directories of equal size.

As the number of OSD+s increases, we obtain outstanding results, specially for Ext4, achieving a superlinear scalability in all the tests. This is mainly due to the fact that Ext4 performance gets worse as the number of entries in a directory grows, as already explained. Hence, by distributing the management of a directory, we are not only sharing out the workload among various servers, but also creating smaller storing directory objects on the local file systems. With ReiserFS, performance also decreases with the directory size, but the downgrade is much softer. Only when a directory is really huge (millions of files), the downgrade is noticeable and ReiserFS can achieve a superlinear scalability too.

We have measured the effect of the directory size in Figure 5.7. We create a single shared hugedir with 800,000 files per directory object. The figure reflects how Ext4 and ReiserFS performances vary over the course of the time for the *create* test. Ext4 performance decreases as storing objects get larger, from roughly 95,000 ops/s at the beginning to 45,000 ops/s at the end of the test. Also, performance drops every 5 seconds are due to the metadata commit interval in Ext4. ReiserFS behaves quite differently, and its performance only decreases from roughly 145,000 ops/s to 115,000 ops/s, getting a sustained high rate of operations per second. The performance drop after second 30 is due to an asynchronous commit and transaction timeouts in ReiserFS.

Disk caches also explain why performance increases with the number of OSD+s, and why ReiserFS's scalability is usually worse than Ext4's. For both Ext4 and ReiserFS, more OSD+s imply a bigger "aggregate disk cache", so disk performance is improved since more disk blocks fit in cache, and blocks are evicted later. ReiserFS, however, produces a quite "random" access pattern from a cache's point of view [42]. This randomness limits the benefits that ReiserFS can take from a larger cache, so its scalability is usually linear and smaller than that achieved by Ext4.

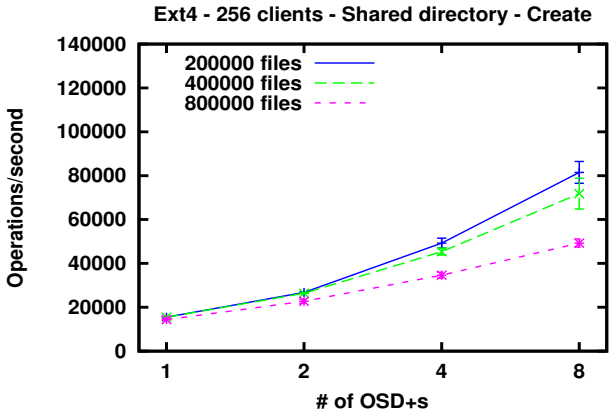
SSD-OSD+

Figure 5.8 depicts the throughput in operations/s obtained by FPFS with SSD-OSD+ devices, when 256 clients access concurrently a hugedir. Figure 5.9 shows the speedup achieved for the same test.

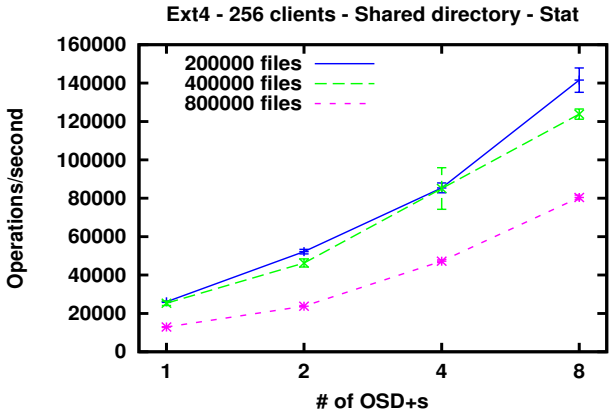
As we have seen in the results for HDD-OSD+s, the directory size significantly determines FPFS performance in all tests. Conversely, results for SSD-OSD+ devices in Figure 5.8 say that the directory size only affects the *unlink* test. Results for *create* and *stat* do not depend on the directory size due to the seek-free feature of the SSD devices. This fact also explains why scalability is linear in these tests.

For *unlink*, the problem is that, if a directory grows, then disk writes increase by a factor much greater than the increase in the number of files. Consequently, FPFS gets a better performance when there are 200,000 files per object than when there are 800,000. This also makes FPFS scalability superlinear in this test, because when we share out the directory, we create smaller directories in each OSD+, reducing this way the increase of disk writes.

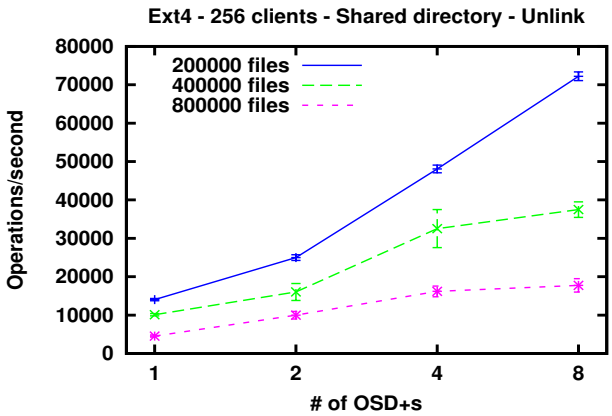
For ReiserFS, performance hardly depends on the directory objects' size and only there are some differences in the *unlink* test. Moreover, although ReiserFS increases disk writes by a factor much greater than Ext4 for a very high number of files during unlinks, its performance is roughly the same for the three directory sizes in this test. The fact that ReiserFS does not have block groups produces less head seeks too, so the use of SSD drives does not make a big difference, and results are quite similar to those obtained with HDD-OSD+s devices.



(a) create

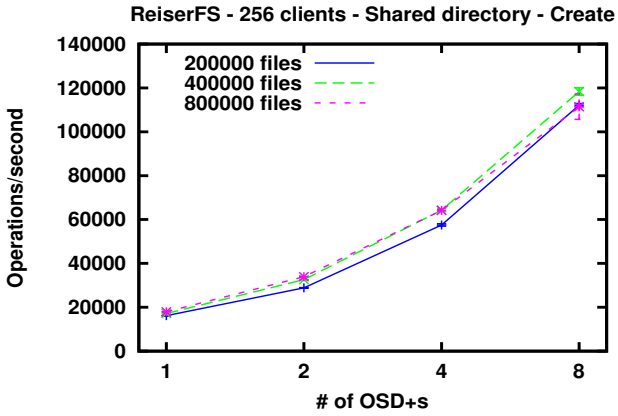


(b) stat

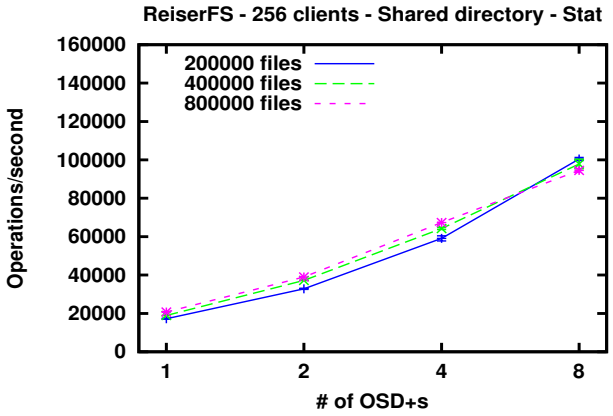


(c) unlink

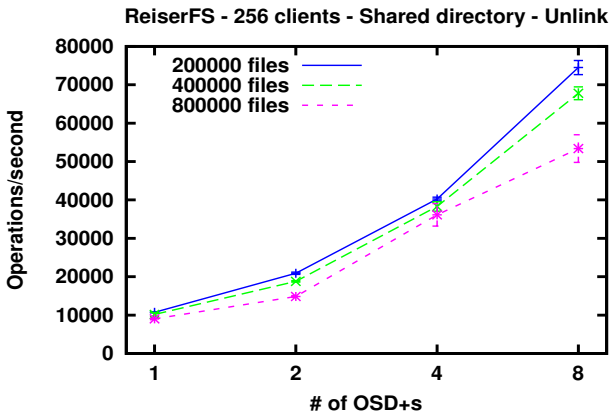
Figure 5.5: Operations per second obtained by FPFS with HDD-OSD+s and Ext4 when using one shared directory.



(c) create

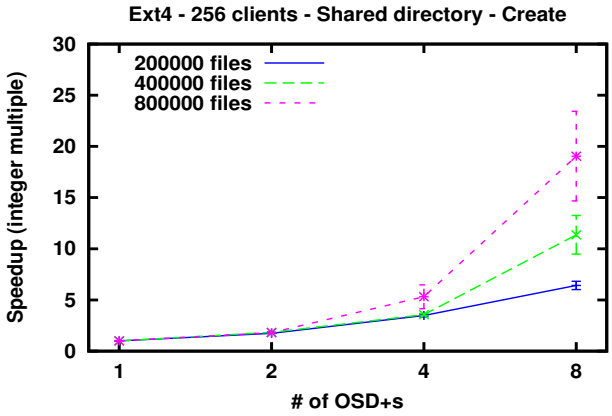


(d) stat

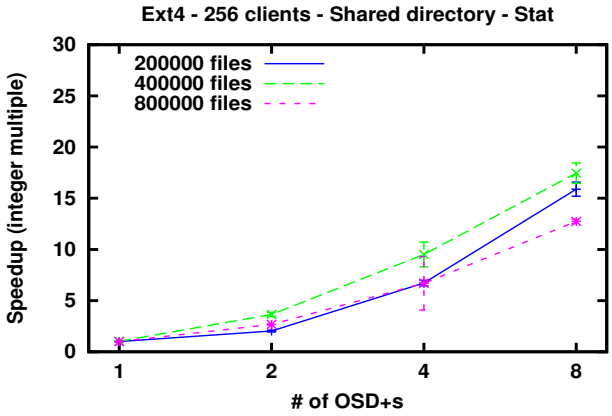


(e) unlink

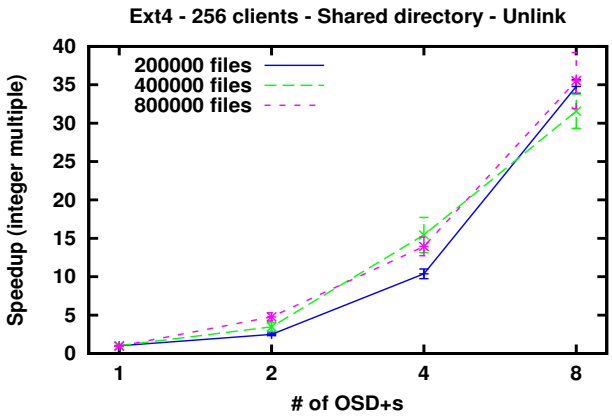
Figure 5.5: (Cont.) Operations per second obtained by FPFS with HDD-OSD+s and ReiserFS when using one shared directory.



(a) create

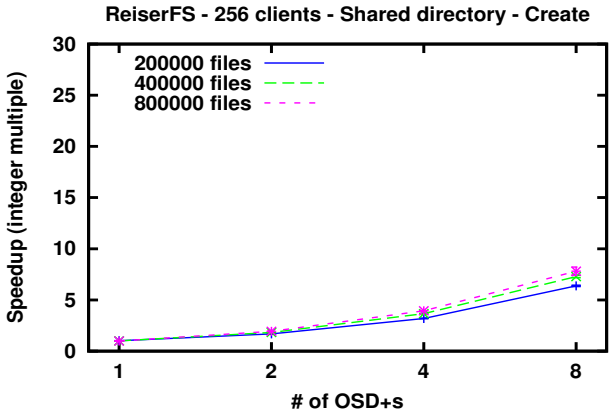


(b) stat

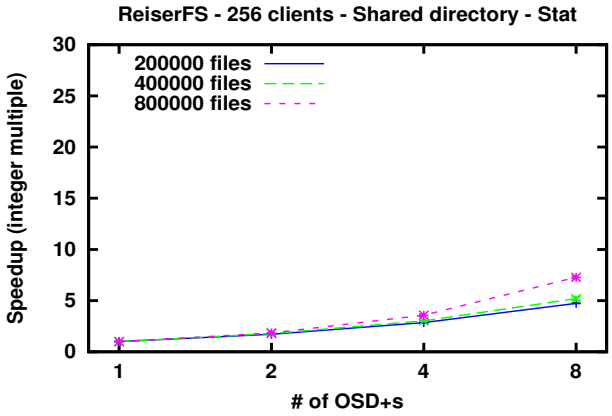


(c) unlink

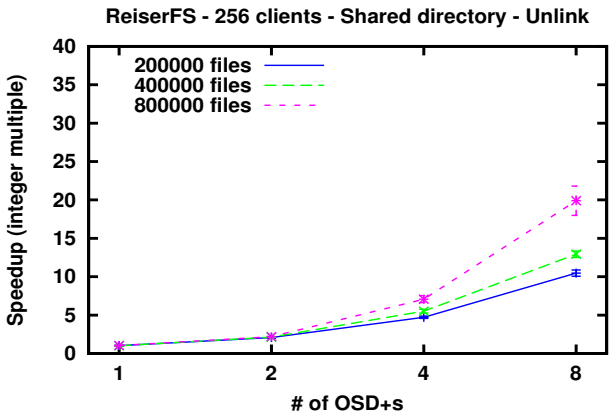
Figure 5.6: Scalability obtained by FPFS with HDD-OSD+s, and Ext4 as file system when using one shared hugedir.



(d) create

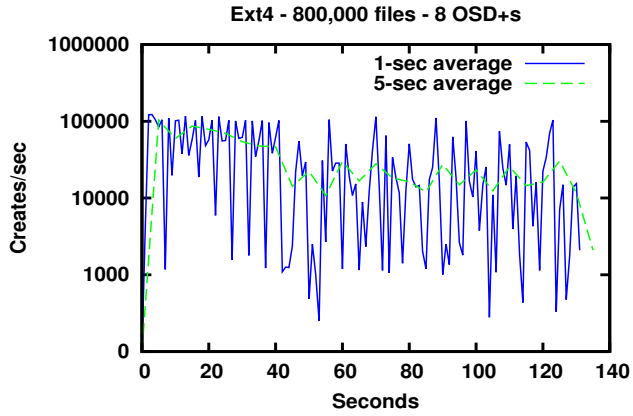


(e) stat

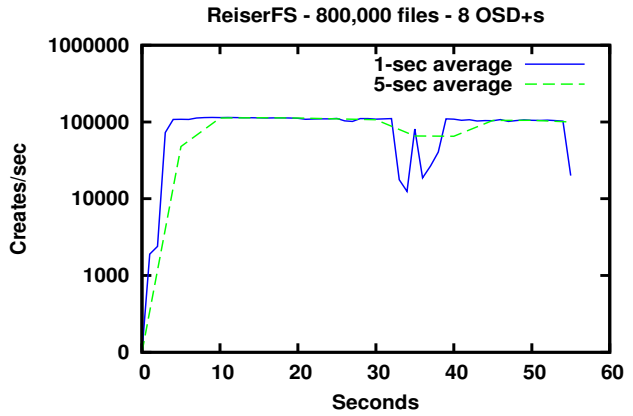


(f) utime

Figure 5.6: (Cont.) Scalability obtained by FPFS with HDD-OSD+s, and ReiserFS as file system when using one shared hugedir.



(a) Ext4



(b) ReiserFS

Figure 5.7: Directory size effect on performance. Graphs show the average number of create operations per second every second, and every 5 seconds. Note the special scale in the Y-axis.

According to Figure 5.9, scalability is linear for *create* and *stat*, whereas it is super-linear for *unlink* mainly because Ext4 and ReiserFS performance downgrades as a directory grows, as we have already explained (see Figure 5.8.(c) and 5.8.(f)).

Note that, because of the buffer cache in the OSD+s, *create* is dominated by the network overhead, so results for SSDs and HDDs are "similar". However, there are important differences for *stat* and *unlink* since both produce read requests that must be served by the drives.

OrangeFS

Figure 5.10 depicts the throughput in operations/s obtained by OrangeFS with SSD devices. Figure 5.11 shows the speedup achieved. Note that, there are no results for *stat* and *unlink*, 800,000 files and 8 OSD+s, because, when we performed the tests, there were continuous communication timeouts.

Results with OrangeFS 2.8.3-20110323 (which does not distribute directories) were inconsistent between runs in this test, probably due to some sort of bug in OrangeFS or the Berkeley DB used as backend. Hence, to obtain results without distribution, we used version 2.8.3-EXP instead, but using a single server. Also, considering the results in the baseline case (see Section 5.4.1), where OrangeFS achieves a quite modest performance with respect to FPFS, we decided to focus on OrangeFS on SSDs only.

Figure 5.10 shows that results obtained by OrangeFS are very similar for different directory sizes. Unlike FPFS, OrangeFS stores directory entries in a few files of a Berkeley DB, so the backend file system does not affect the performance.

OrangeFS is able to create and delete around 9,000 files per second, and stating 10,000. FPFS increases these rates at least by an order of magnitude. For *create* and *unlink*, throughput improves as the number of servers grows, but they stay the same for *stat*. Analyzing the workload of the servers during the *stat* test, we have seen that OrangeFS does not distribute the directory uniformly; there is always one server working at least twice the rest.

Figure 5.11 shows scalability achieved, which is nearly linear for *create* and *unlink*, but remains close to 1 for *stat* due to the aforementioned workload imbalance.

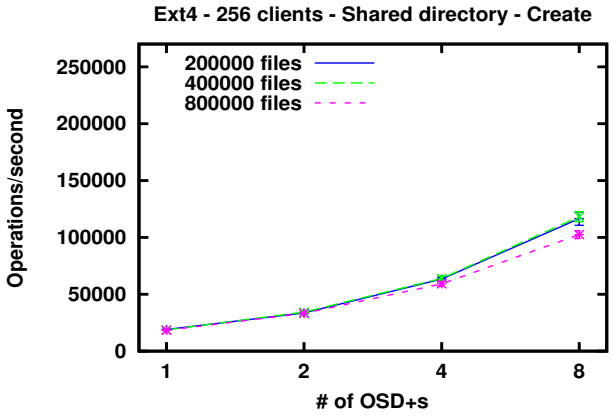
5.4.3. mdtest

Since we have not fully implemented data objects yet, we have discarded usual checkpointing applications. Instead, we have run *mdtest* to emulate them, but only for metadata operations. In this benchmark, *mdtest* creates 100 empty files per process (in order to create a single shared directory big enough to split), and executes 10 iterations (plus a first warm-up iteration that is dismissed). Each iteration creates a new shared directory. Barriers are used between iterations to synchronize the progress of all the processes. Each data point in the graphs is the average of those 10 iterations.

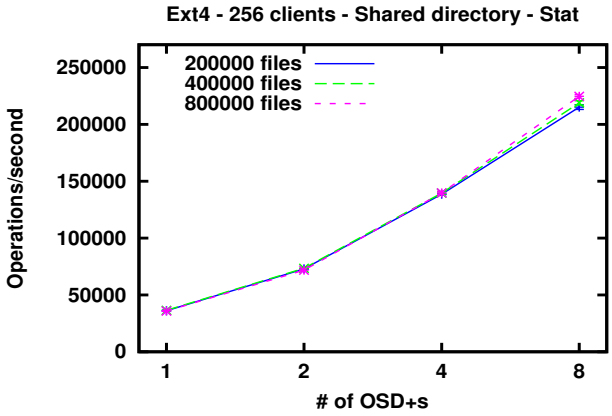
We tested three configurations of FPFS that we call *always*, *dynamic* and *never*. *always* distributes the single shared directory from the very beginning (on the first request). *dynamic* distributes the directory when its size is greater than 244kB (around 8,000 files). *never* implies that there is no distribution. Due to its low performance, OrangeFS was discarded.

Figure 5.12 shows the throughput in operations/s obtained with HDD-OSD+ devices. Figure 5.13 shows the speedup. Figures 5.14 and Figure 5.15 show the equivalent results for SSD-OSD+s, which are virtually the same as those for HDD-OSD+s. This similarity is mainly because, as in *create*, the performance is dominated by the network overhead. Ext4's and ReiserFS's results are also basically the same except for 8 OSD+s, where the smaller size of the directory objects also helps Ext4 to increase its throughput.

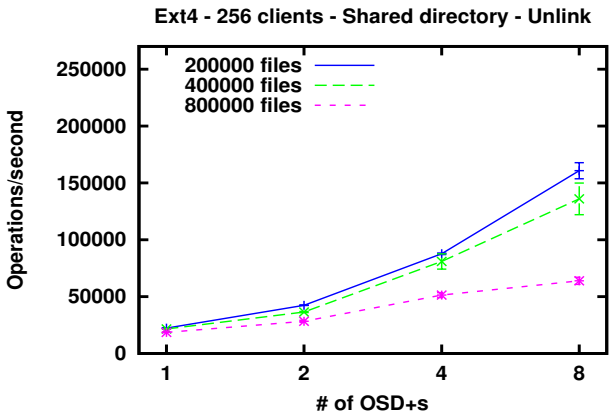
As showed, there are noticeable differences between *always*, *dynamic* and *never*. *dynamic* gets worse results than *always*, but better than *never*. The problem here is that, although the redistribution of entries is fast, each run takes only a few seconds and, hence, the small overhead of the redistribution is noticeable. FPFS achieves the best results with *always*, because its early distribution of the shared directory avoids the cost of redistributing existing directory entries, and allows clients to perform more operations in parallel from the beginning. However, in long-lasting runs (as those showed in previous sections), differences between *always* and *dynamic* would be negligible. These results confirm that, if a directory is expected to be big but not huge, it is better to distribute it from the start.



(a) create

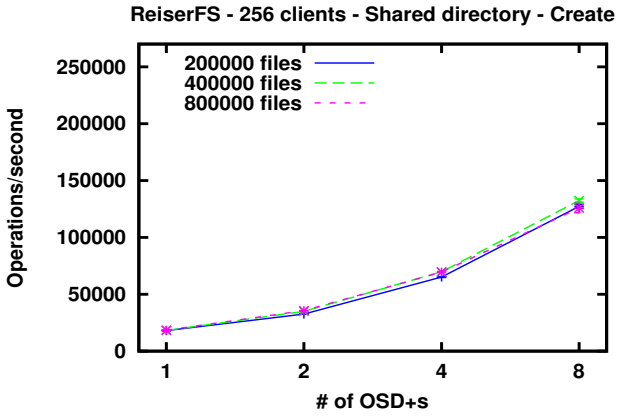


(b) stat

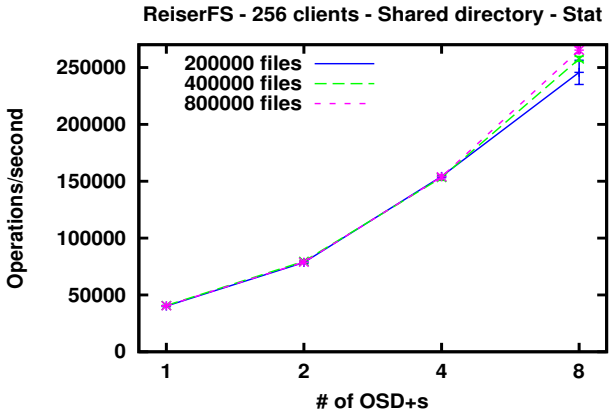


(c) unlink

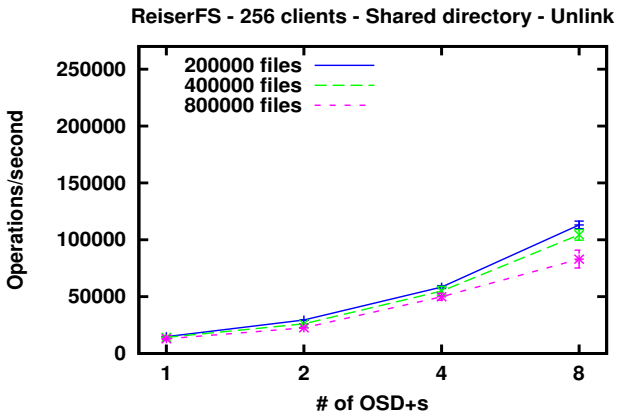
Figure 5.8: Operations per second obtained by FPFS with SSD-OSD+s and Ext4 when using one shared directory.



(d) create

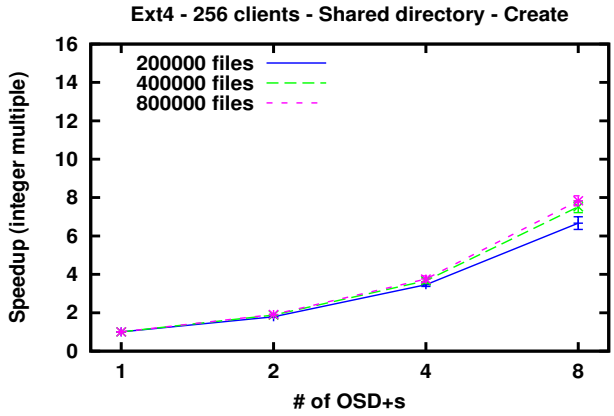


(e) stat

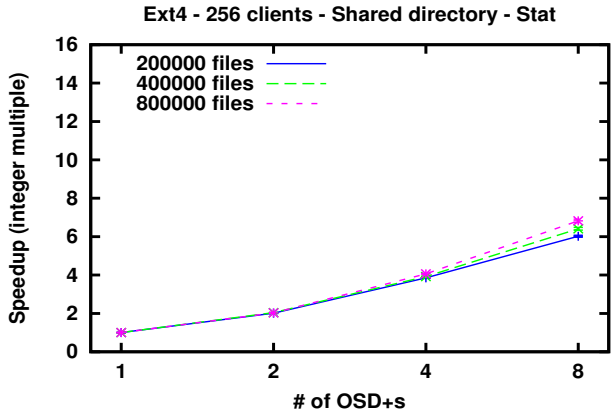


(f) unlink

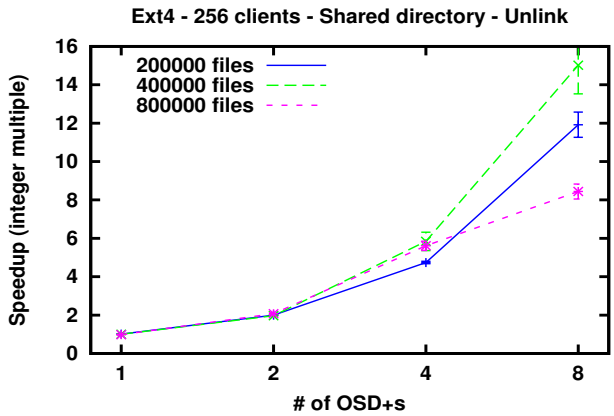
Figure 5.8: (Cont.) Operations per second obtained by FPFS with SSD-OSD+s and ReiserFS when using one shared directory.



(a) create

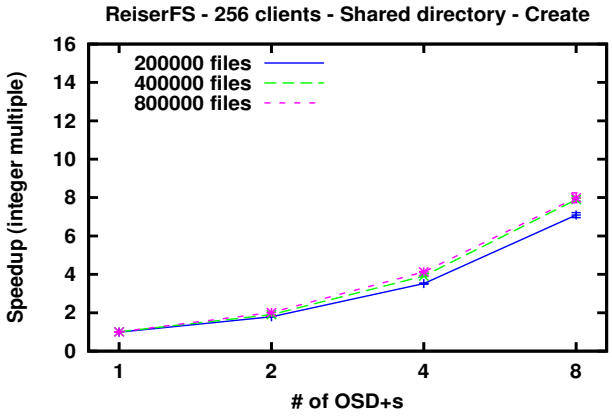


(b) stat

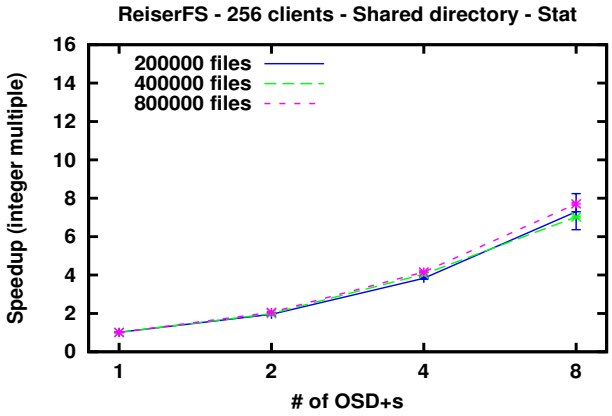


(c) unlink

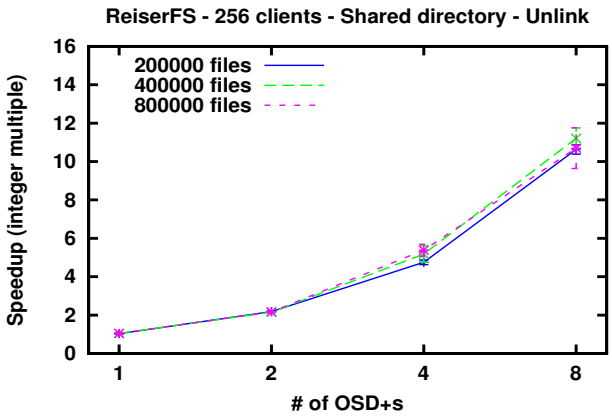
Figure 5.9: Scalability obtained by FPFS with SSD-OSD+s, and Ext4 as file system when using one shared hugedir.



(d) create

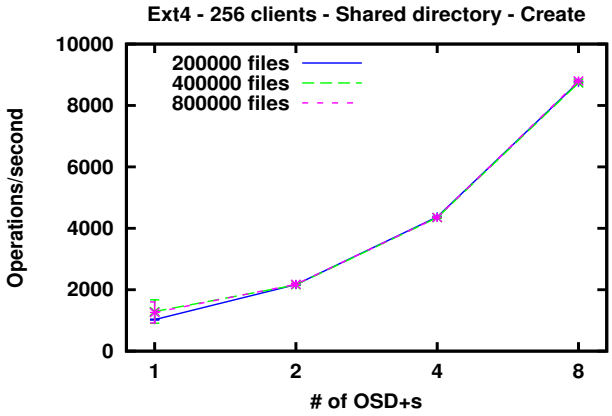


(e) stat

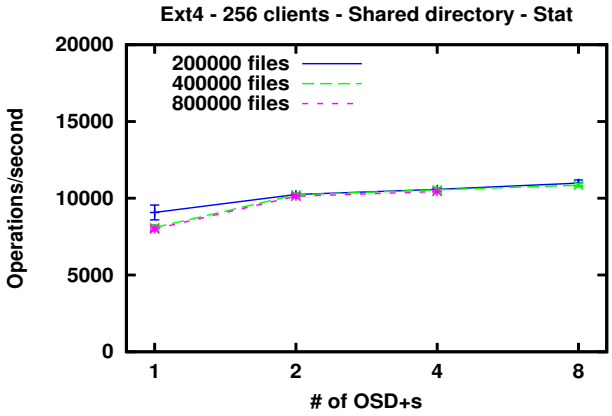


(f) utime

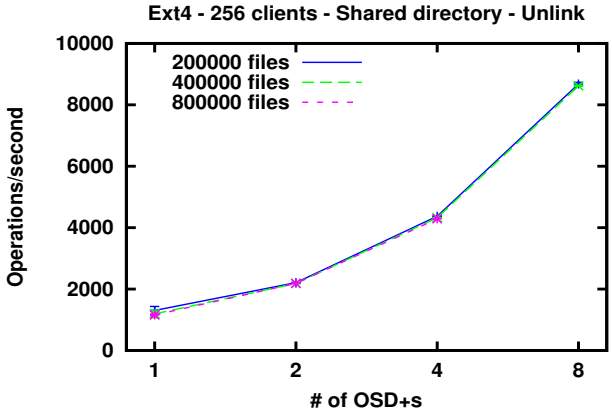
Figure 5.9: (Cont.) Scalability obtained by FPFS with SSD-OSD+s, and ReiserFS as file system when using one shared hugedir.



(a) create

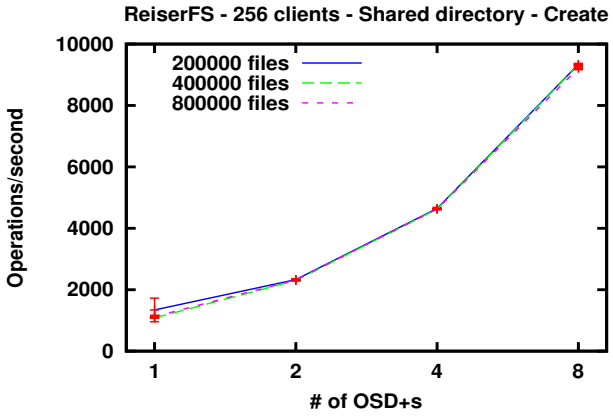


(b) stat

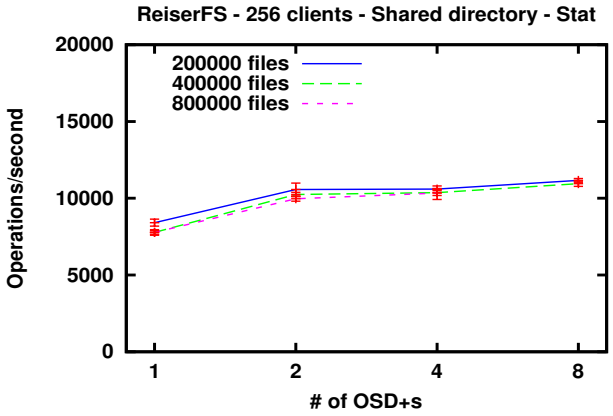


(c) unlink

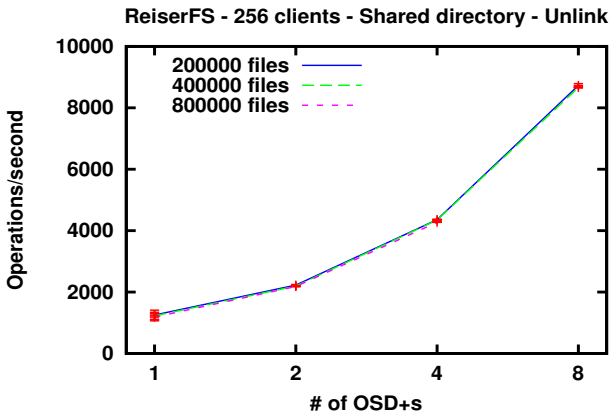
Figure 5.10: Operations per second obtained by OrangeFS with SSD-OSD+s and Ext4 when using one shared directory.



(d) create

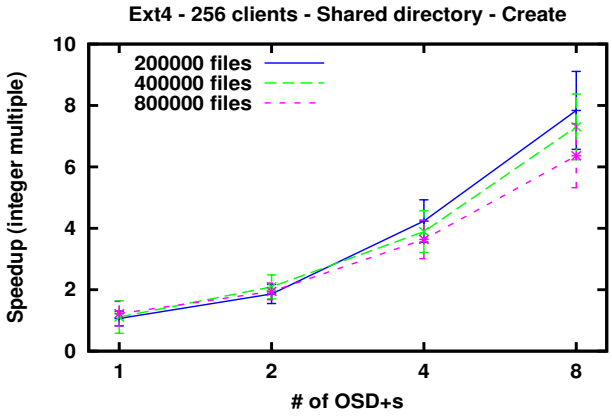


(e) stat

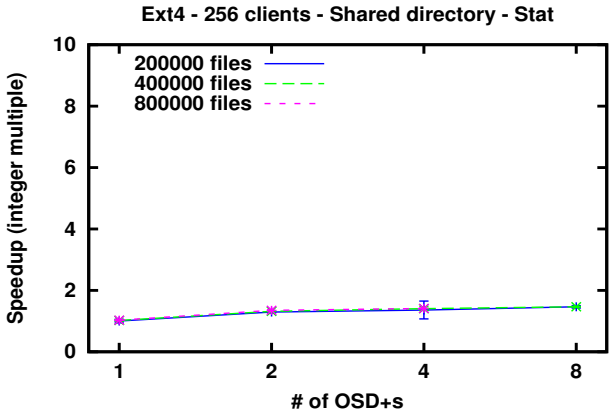


(f) unlink

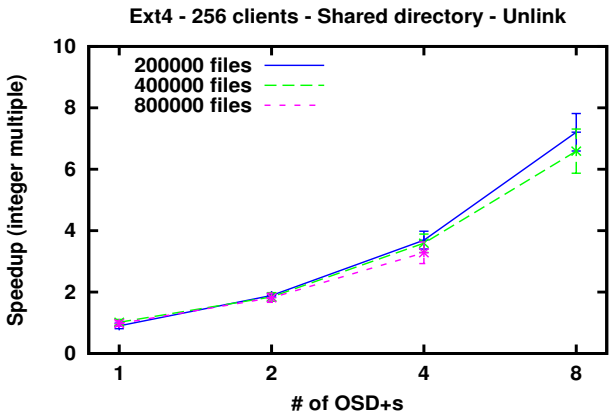
Figure 5.10: (Cont.) Operations per second obtained by OrangeFS with SSD-OSD+s and ReiserFS when using one shared directory.



(a) create

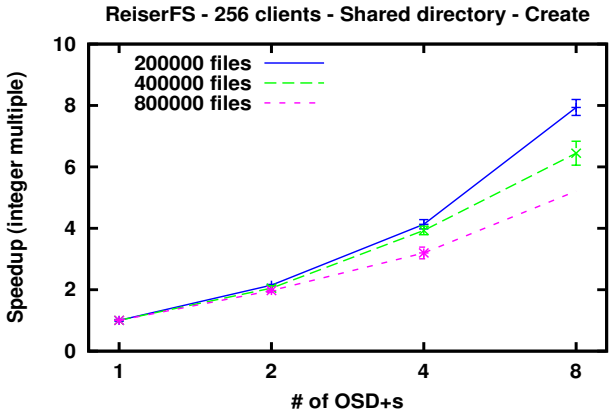


(b) stat

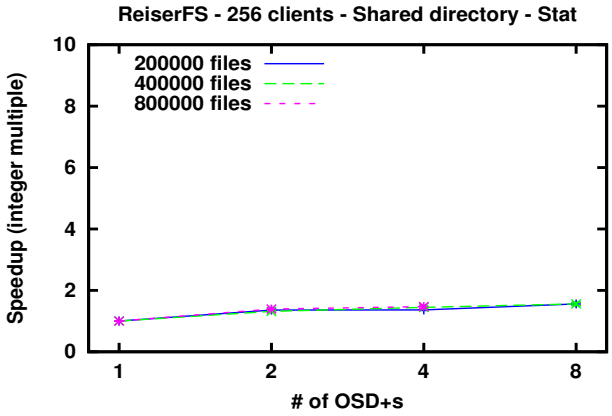


(c) unlink

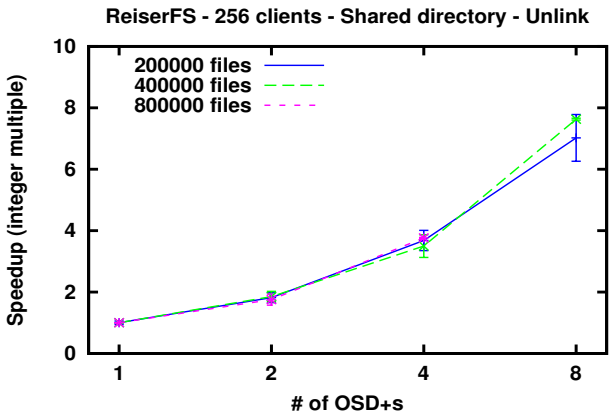
Figure 5.11: Scalability obtained by OrangeFS with SSD-OSD+s, and Ext4 as file system when using one shared hugedir.



(d) create

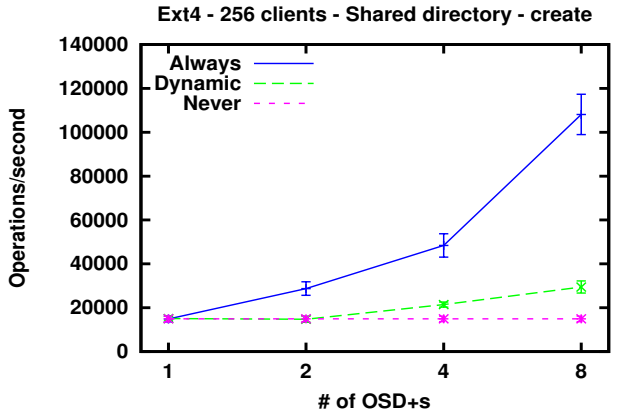


(e) stat

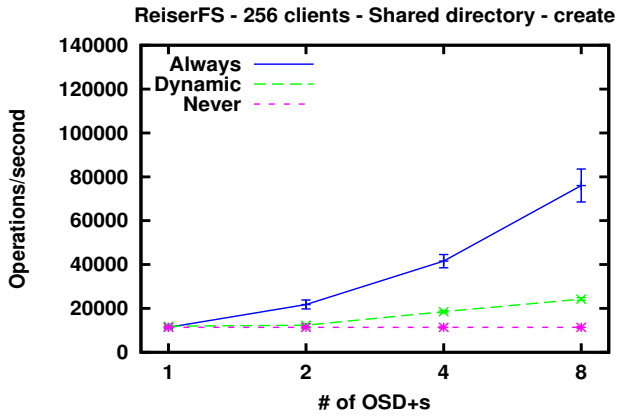


(f) utime

Figure 5.11: (Cont.) Scalability obtained by OrangeFS with SSD-OSD+s, and ReiserFS as file system when using one shared hugedir.



(a) Ext4



(b) ReiserFS

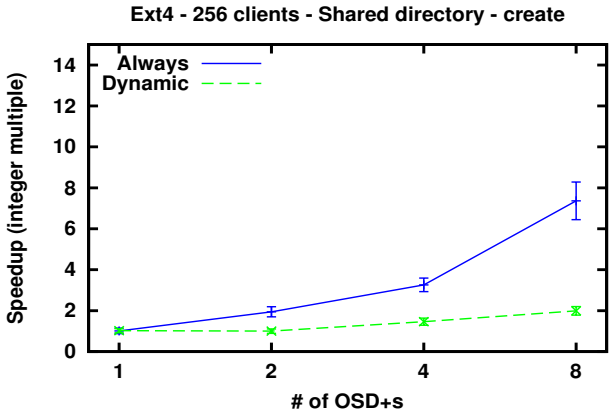
Figure 5.12: Operations per second obtained by FPFS with Ext4 and ReiserFS on HDD-OSD+s for *mdtest*.

Finally, FPFS gets the smallest figures for *never*; the performance slightly decreases when the number of OSD+s increases. The reason is that, each iteration creates a new directory that can be located in any OSD+ what, in turn, produces more connection between clients and servers, introducing a small overhead.

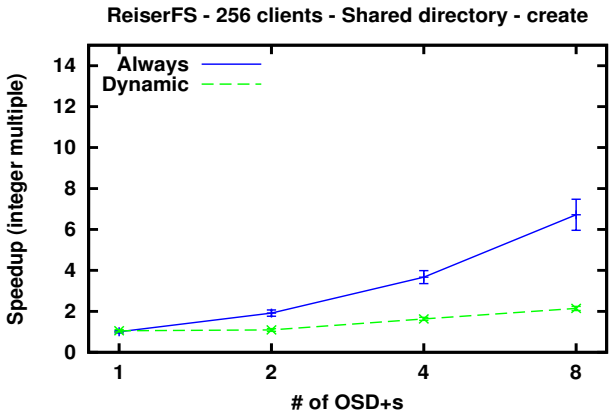
5.4.4. Multiple huge directories

As previous section shows, distribution is beneficial for a hugedir accessed by hundreds or thousands of clients at the same time. However, results can be rather different when there are multiple hugedirs. Therefore, in this section we evaluate the behaviour of FPFS when several hugedirs are concurrently accessed by a few clients.

In the following tests there are 8 directories, each containing 320,000 files (2,560,000 files in total), and each accessed by 1, 16, and 32 processes. Note that 1 client per directory is a case of non-shared directories, and that, with 32 clients per directory, there are 256 clients altogether.



(a) Ext4



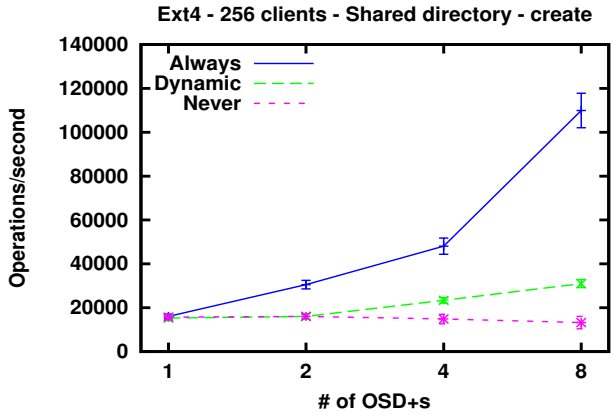
(b) ReiserFS

Figure 5.13: Speedup (respect to *never*) obtained by FPFS with Ext4 and ReiserFS on HDD-OSD+s for *mdtest*.

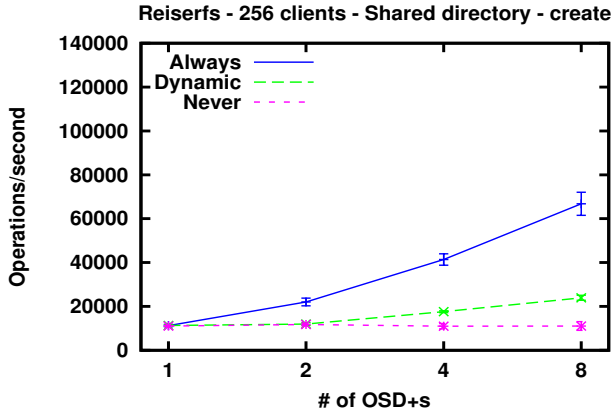
Tables show, for each number of processes, absolute application times when hugedirs are never distributed in the first column. The other two columns show relative application-time variations, in percentage, with respect to the absolute times, when hugedirs are distributed dynamically (i.e., when a directory exceeds 8,000 files), and always (i.e., when threshold is 0). Confidence intervals (not showed) are smaller than 10% of the mean. Note that a positive/negative percentage means an increase/decrease in time (hence, a worse/better performance).

HDD-OSD+

Table 5.3 shows the absolute application times and relative application-time variations when using HDD-OSD+ devices. According to the tables, distribution usually downgrades performance in workloads dominated by disk writes (*create* and *unlink*), whereas it can even reduce the application time when the workload is dominated by reads (*stat*). When there are 16 or 32 clients per directory, distribution can improve the performance in some cases, e.g.,



(a) Ext4



(b) ReiserFS

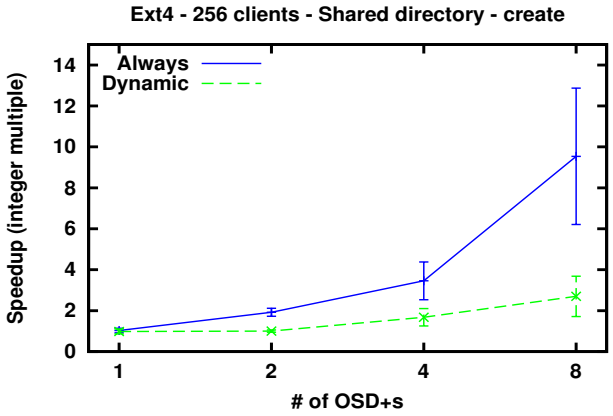
Figure 5.14: Operations per second obtained by FPFS with Ext4 and ReiserFS on SSD-OSD+s for *mdtest*.

for ReiserFS and *stat*. However, performance decreases in many other cases, e.g. Ext4 with 2 or more OSD+s.

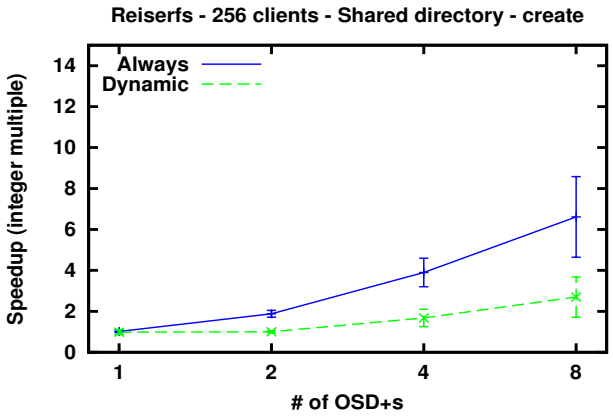
The main reason for these results is that distribution makes disk accesses less efficient, because there are always 8 directory objects per server accessed almost at the same time. Since the underlying file system spreads directory objects over the disk, this increases head seeks and trashes disk caches. This is specially relevant for Ext4, which divides the disk into block groups, so directory objects are placed further apart. Without distribution, the number of objects per server is reduced to $8/N$, where N is the number of OSD+s. Less objects means less head seeks and better performance.

In ReiserFS, downgrades are smaller. Even there exist more cases where the distribution of directories improves the performance over non-distributed directories. This is because ReiserFS puts directory objects close on disk, which produces shorter seeks.

The way the redistribution process of directory entries is carried out also affects the results. We do not remove entries copied from the routing to the storing objects. This fact allows us to reduce the redistribution time considerably. Note that this issue does not appear when



(a) Ext4



(b) ReiserFS

Figure 5.15: Speedup (respect to *never*) obtained by FPFS with Ext4 and ReiserFS on SSD-OSD+s for *mdtest*.

there is only one OSD+ or the threshold is 0, since there is no distribution or directories are always distributed, so no redistribution is performed.

SSD-OSD+

Table 5.4 shows the absolute application times and relative application-time variations when using SSD-OSD+ devices. The results show that the seek-free feature of SSDs mitigates the downgrade HDD-OSD+s suffer to a large extent in this test when directories are distributed.

With HDDs, there was an increase of head seeks due to having 8 directories spread along the disk. Now, results for ReiserFS are similar to those obtained by FPFS with Ext4 in the *create* and *stat* tests, because the placement strategies of both file systems do not affect the device performance. Yet, distribution is still worse for some configurations, because there is still disk contention and cache trashing. In the *unlink* test, although the distribution increases application times for both file systems due to the aforementioned disk contention and cache

Table 5.3: Performance got by FPFS on HDD-OSD+s with Ext4/ReiserFS when 8 hugedirs are accessed concurrently.

(a) Ext4

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)
create	1	215.27	1.11	-1.17	152.48	-0.18	-0.19	154.98	-0.54	-0.45
	2	150.67	32.00	31.01	60.48	42.25	43.09	63.06	33.57	32.32
	4	106.87	53.78	50.00	33.53	56.58	36.03	34.27	45.16	25.64
	8	92.75	51.11	48.51	21.29	78.75	69.56	21.59	46.78	40.52
stat	1	243.18	-1.06	0.08	203.00	0.39	2.06	207.79	-0.48	-0.50
	2	126.56	7.87	6.17	77.01	4.36	0.21	77.62	2.56	-0.94
	4	76.60	11.15	8.73	24.06	11.21	5.50	23.73	13.64	5.23
	8	68.98	12.89	11.77	12.63	16.18	11.53	11.91	28.17	22.99
unlink	1	512.57	-0.21	-0.95	1099.27	10.77	2.79	1522.21	-10.25	-5.48
	2	163.31	40.89	38.53	258.33	54.12	47.61	280.84	62.12	59.80
	4	85.97	66.94	59.44	80.67	103.53	28.30	95.90	110.65	53.80
	8	63.73	79.04	74.81	21.49	268.43	262.04	22.02	270.44	131.73

(b) ReiserFS

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)
create	1	134.93	-0.60	-0.71	115.56	-0.86	1.35	125.80	-0.94	0.46
	2	120.71	11.69	9.51	55.84	5.82	2.57	55.69	13.18	13.34
	4	101.09	12.07	11.82	26.76	9.05	6.72	27.03	10.67	7.48
	8	89.67	10.79	11.08	17.07	3.82	0.50	17.13	-3.02	-5.86
stat	1	98.70	-0.24	-0.25	123.81	0.28	0.16	123.54	-2.41	-0.69
	2	96.18	2.71	0.30	46.32	-14.10	-14.24	50.16	-22.09	-23.37
	4	106.56	-5.38	-5.04	35.25	-24.80	-24.67	34.09	-28.33	-28.58
	8	67.54	7.55	6.39	28.00	-30.52	-30.79	25.69	-31.63	-31.91
unlink	1	171.28	-0.25	0.11	183.56	-0.29	0.31	236.28	-4.44	-6.24
	2	106.25	3.63	2.00	90.42	9.99	4.84	93.34	40.98	36.99
	4	105.33	-0.65	0.14	55.16	-13.60	-16.12	54.34	10.40	3.89
	8	65.14	8.13	6.60	35.63	-25.58	-28.33	35.47	-9.41	-12.75

trashing, it downgrades more the performance, in relative terms, for Ext4 than for ReiserFS. This is because Ext4 writes many more blocks to disk than ReiserFS for this test.

OrangeFS

Table 5.5 shows the absolute application times and relative application-time variations when using OrangeFS and SSD devices. These results were obtained with version 2.8.3-EXP of OrangeFS that distributes directories from the beginning. Hence, column "dynamic" makes no sense and has been removed. As with the single shared hugedir, we focus on SSD-OSD+ devices following the baseline results that showed a poor performance for OrangeFS.

Table 5.4: Performance obtained by FPFS on SSD-OSD+ devices with Ext4 and ReiserFS when 8 hugedirs are accessed concurrently.

(a) Ext4

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)
create	1	138.54	0.09	0.43	81.39	-0.15	-0.56	84.68	-0.35	-0.11
	2	121.82	14.37	13.28	39.94	11.81	10.47	42.17	9.09	6.27
	4	98.60	19.82	20.95	24.50	15.11	3.74	25.28	1.05	0.43
	8	87.47	23.10	20.59	17.35	13.02	5.85	17.70	-2.00	-1.71
stat	1	78.87	0.12	-0.66	33.13	0.08	-0.17	34.43	-1.28	-0.75
	2	73.04	4.14	4.06	18.65	-2.84	-3.25	19.46	-6.94	-8.63
	4	68.33	5.35	5.88	13.37	-4.23	-7.75	12.98	-15.95	-13.74
	8	66.93	5.56	5.14	11.63	-4.26	-5.42	10.62	-3.80	-3.62
unlink	1	122.82	1.12	-0.10	181.86	-2.12	-2.35	191.35	-0.13	2.10
	2	75.69	10.86	10.84	52.63	50.63	38.98	53.59	83.65	60.73
	4	65.79	19.15	18.75	24.59	91.18	65.92	25.23	95.39	81.80
	8	56.31	24.29	22.06	14.57	5.19	3.88	14.57	25.36	5.29

(b) ReiserFS

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)
create	1	132.05	-0.92	-0.94	110.58	1.13	-0.44	113.07	1.66	0.57
	2	119.86	9.13	8.50	51.14	8.30	5.91	53.81	8.15	6.43
	4	99.84	10.34	10.50	25.61	9.40	6.47	26.07	9.05	6.19
	8	88.52	10.94	11.28	16.46	3.19	-0,67	16,39	-4,87	-8,44
stat	1	76.46	-0.18	-0.49	41.49	0.09	-0.26	42.84	-0.38	-0.09
	2	70.89	4.87	3.90	20.16	4.41	4.21	21.10	3.43	1.45
	4	68.08	3.90	4.35	11.68	1.00	1.70	11.69	-4.78	-4.84
	8	66.64	4.39	5.67	10.57	1.11	0.25	9.77	1.82	2.07
unlink	1	160.64	-0.58	-0.61	173.02	-1.10	1.10	190.02	-1.93	-1.06
	2	84.53	1.88	1.68	82.98	5.02	2.79	86.80	12.23	11.26
	4	67.13	7.53	9.42	40.43	5.11	5.76	40.99	17.83	15.03
	8	62.70	5.89	6.08	20.93	5.17	3.08	21.55	10.21	10.18

As we have seen, the backend file system does not affect results in OrangeFS due to the Berkeley DB they use to store directory entries. Tables 5.5.(a) and (b) present practically the same times and percentages of improvement.

Similar to FPFS, results show that distribution is not worth most of the time. Compared to FPFS, distribution in OrangeFS is able to achieve better reductions in the application time in some cases of the *create* test. However, for the *stat* test, distribution increases the application time over 20%. This is probably due to the fact that OrangeFS does not uniformly share the load among the servers for this test.

Note that if we compare absolute times, OrangeFS overall performance is 10 times lower than FPFS performance.

Table 5.5: Performance obtained by OrangeFS with Ext4 and ReiserFS when 8 hugedirs are accessed concurrently.

(a) Ext4

		1 client/directory		16 clients/directory		32 clients/directory	
Test	#OSD+	Never(s)	Always(%)	Never(s)	Always(%)	Never(s)	Always(%)
create	1	1680.94	-1.83	1394.17	-0.15	1419.68	-1.37
	2	1303.93	1.59	735.27	6.00	728.31	5.20
	4	889.53	6.33	388.02	3.22	372.10	4.34
	8	681.81	3.69	222.04	0.30	225.17	-9.35
stat	1	348.82	1.86	332.85	10.80	344.56	8.66
	2	300.10	20.00	196.73	33.47	190.51	23.15
	4	283.96	22.98	99.00	37.19	100.92	43.80
	8	275.39	24.20	57.07	80.07	51.71	75.66
unlink	1	2069.41	-0.29	2108.63	-3.10	2141.82	-2.61
	2	1220.93	2.84	1054.56	3.03	1055.06	2.15
	4	806.51	6.76	508.18	6.75	524.04	4.11
	8	708.99	4.67	271.02	1.10	268.74	2.18

(b) ReiserFS

		1 client/directory		16 clients/directory		32 clients/directory	
Test	#OSD+	Never(s)	Always(%)	Never(s)	Always(%)	Never(s)	Always(%)
create	1	1649.77	-0.02	1320.86	2.82	1340.10	0.84
	2	1232.25	3.97	683.50	6.68	694.30	3.79
	4	880.91	2.64	360.07	8.39	352.31	4.88
	8	662.93	7.48	203.65	11.23	217.65	-8.59
stat	1	352.81	6.98	345.86	10.96	354.86	6.49
	2	302.33	19.55	198.91	28.43	199.38	23.96
	4	283.37	22.61	100.04	53.80	101.76	49.66
	8	274.95	24.73	54.92	65.73	49.78	63.13
unlink	1	2097.61	1.24	2140.96	-1.91	2173.56	-2.04
	2	1226.95	2.42	1055.36	2.81	1065.61	3.26
	4	809.14	7.43	509.78	6.75	522.51	4.46
	8	714.00	4.62	260.54	5.99	263.78	5.09

5.4.5. Mixed huge directories

The question to be answered at this point is if distribution of hugedirs is a good idea or not. To answer this question we have run the following test where a distributed hugedir and a non-distributed hugedir are accessed at the same time by 128 clients each. There are always 1,280,000 files per directory, evenly shared out among the clients. Figure 5.16 depicts the performance achieved by FPFS in operations/s with SSD-OSD+ devices for this test. We could not perform this test for OrangeFS, since version orangefs-2.8.3-EXP distributes any directory.

Results show that, as the number of SSD-OSD+s increases, so does the throughput in operations/s of the distributed hugedir, significantly outperforming the non-distributed hugedir for the same workload. For instance, by using 8 SSD-OSD+s and Ext4, the distributed hugedir achieves 117,299, 166,057, and 90,068 ops/s for *create*, *stat* and *unlink*, respectively, whereas the non-distributed one only gets 16,714, 32,677, and 15,921 ops/s. OSD+s store only one directory (the distributed directory), except for the OSD+ containing the non-distributed directory, which stores two. For this OSD+, results also show that, as the workload of the distributed directory decreases, the performance of the non-distributed directory slightly increases.

Therefore, if enough resources are available, distribution is clearly beneficial. In this vein, our approach of distributing a hugedir across a set of OSD+s in the cluster maximizes the use of existing resources. Generally, clusters of hundreds of nodes are expected, so hugedirs will probably be shared out on disjoint sets of OSD+s. This way, hugedirs do not interfere each other, achieving great performance.

5.5. Conclusions

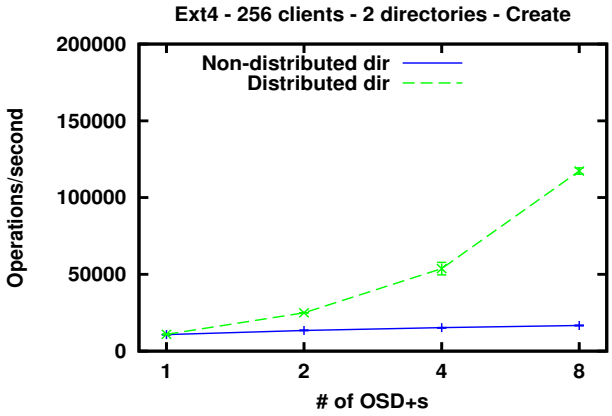
In this chapter we have presented a technique for FPFS and OSD+ devices to deal with hugedirs accessed by thousands of clients concurrently. Directory objects in OSD+s allow FPFS to dynamically distribute a hugedir among several servers. Storing objects supporting a hugedir work independently, providing a good performance and scalability while maintaining POSIX semantics. We have also optimized the redistribution of existing directory entries, and avoided massive metadata migrations on renames.

Results show that FPFS exceeds today's requirements of HPC applications regarding huge directories (a billion files per directory, more than 40,000 files created per second, etc.), outperforming OrangeFS by one order of magnitude. By using a Gigabit interconnect, just 8 HDD-OSD+ and Ext4 as backend file system, our proposal is able to create more than 70,000 files/s, stat more than 120,000 files/s and delete more than 37,000 files/s for a directory with 3,200,000 files. When the backend file system is ReiserFS, these numbers are 118,000, 97,000 and 67,000 files/s, respectively. These rates are even better with SSD-OSD+ devices, which create, stat and delete more than 118,000, 218,000 and 135,000 files/s, respectively, in the case of Ext4, and 132,000, 257,000 and 104,000 files/s when ReiserFS is used. Scalability is usually linear, and even super-linear in some cases, enabling FPFS to easily meet more demanding requirements by just adding more OSD+s.

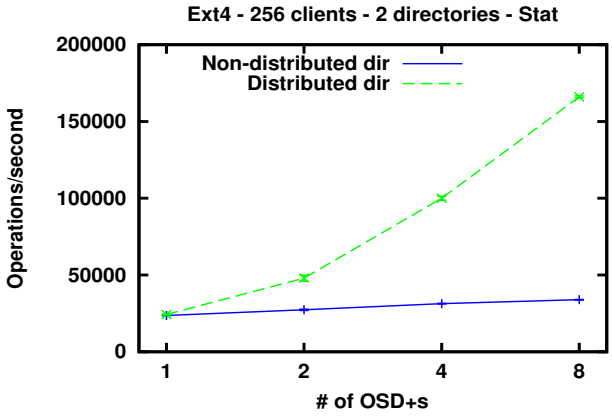
For checkpointing applications, mdtest shows that distribution is also beneficial. Ext4 achieves 110,000 files/s with 8 SSD-OSD+s, whereas ReiserFS gets around 70,000 files/s.

Experiments, however, have produced unexpected results too. While distribution improves the results when many clients access a hugedir, it can downgrade the performance when a few clients access several hugedirs concurrently. SSDs largely mitigate this problem by removing the seek overhead that limits the number of IOPS in HDDs. FPFS also addresses this problem by spreading hugedirs across the cluster, preventing hugedirs from sharing OSD+s to a large extent.

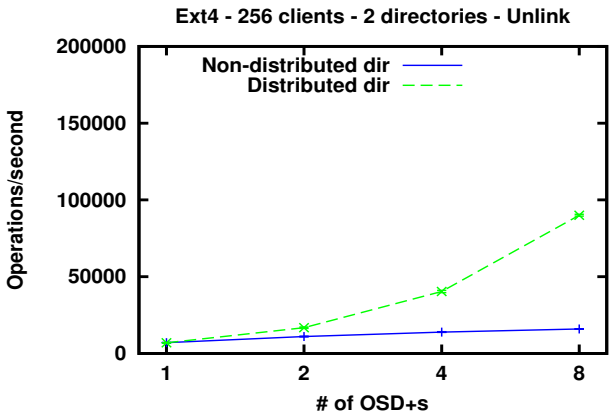
Results question distributions purely based on directory sizes. The number of processes accessing a directory, and the resource availability in the servers are more important. However, both things can vary quickly, and continuously changing the servers a directory is split into



(a) create

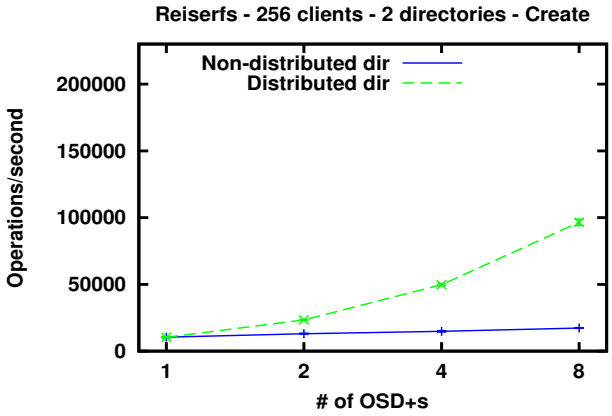


(b) stat

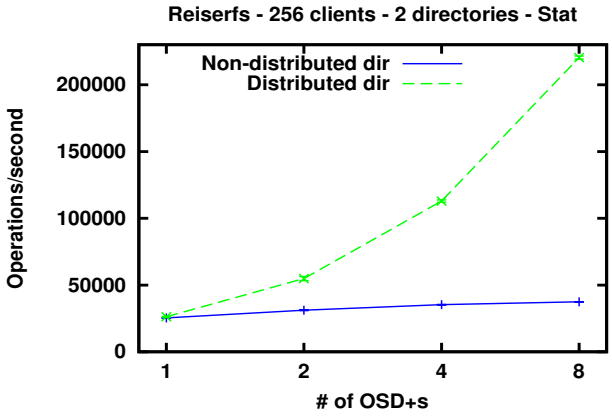


(c) unlink

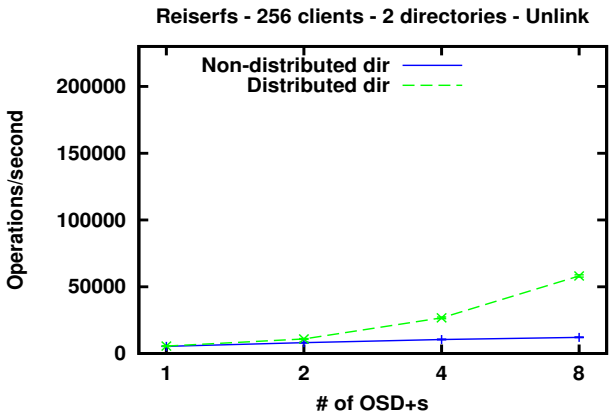
Figure 5.16: Operations per second obtained by FPFS with SSD-OSD+s and Ext4 when using a distributed hugedir and a non-distributed hugedir concurrently accessed.



(d) create



(e) stat



(f) unlink

Figure 5.16: (Cont.) Operations per second obtained by FPFS with SSD-OSD+s and ReiserFS when using a distributed hugedir and a non-distributed hugedir concurrently accessed.

seems inefficient. We plan to find better ways to split directories as future work. Baseline results also show that interconnects can limit the performance of a networked file system for metadata operations, we address this issue in the next chapter.

Chapter 6

Batch Operations

Parallel file systems involve several components, such as the CPUs, I/O subsystems, and networks. Depending on the type of workload and available resources, any of those components can end up being a bottelneck, downgrading the whole system performance.

Specifically, there are high network traffic workloads where the interconnect cannot process the amount of packets that clients and servers exchange. With the purpose of bypassing this bottleneck, we have designed *batch operations* (*batchops* for short), which embed hundreds to thousands of entries of the same type of operation into a single packet. These operations make a much more efficient use of the network, shifting the bottleneck from the network to the servers in many cases. With batchops, we remove network congestion by reducing the number of packets (and, therefore, the number of headers), and by saving network delays and round-trips.

Batchops are a particularly useful tool not only for applications but also for parallel file systems that migrate data or that, as FPFS does, distribute or migrate directories.

In this chapter, we start with related work about similar ideas proposed in other areas. We continue describing the design of batchops on regular directories and huge directories. Following, we present the experimental results achieved and finally, we give the conclusions.

6.1. Related work

To the best of our knowledge, batchops have not been proposed in the parallel file systems field, except for NFSv4 [62]. NFSv4 reduces latency for multiple operations by bundling together different RPC calls. For instance, operations `lookup`, `open`, `read` and `close` can be sent once over the wire, and the server can execute the entire compound call as a single entity.

However, ideas similar to batchops have been used in many different areas. For instance, Linux kernel 3.14 [10] includes a new feature, called *automatic TCP corking*, to help applications doing small write()/sendmsg() to TCP sockets. This feature allows to delay the dispatch of messages in a socket in order to coalesce more bytes in the same packet, and thereby lower the total amount of sent packets. This technique complements batchops, although a study of performance of both is postponed to the future.

Similarly, but in the grid computing area, Chervenak *et al.* [23] use what they call *bulk operations* in the implementation of a *Replica Location Service* (RLS). RLS provides a mechanism for registering the existence of replicas and discovering them within a Grid environment. They store *catalogs* that map logical names to target names. In turn, clients send queries to the servers in order to discover replicas associated with a logical name. Among the operations they support, they include bulk operations to add/delete entries and/or attributes

to the catalogs, and to perform query operations on them. They include 1,000 requests per bulk operation. Their experiments show a significant performance improvement for a single client. However, as the number of clients increases the performance advantage of bulk queries decreases. We obtain a similar behaviour in our experimental results, although our improvement with batchops versus no-batchops is much higher than theirs, and batchops still make sense with a large number of clients.

OpenStack Swift also includes in its Object Storage API two bulk operations: delete [37] and archive extraction [36]. Bulk delete can remove up to 10,000 objects or containers (configurable) in one request. The archive extraction allows to expand a tar file into a Swift account in a single request. Only regular files are uploaded; empty directories, symlinks, etc. are not uploaded.

Another area where reducing the number of requests is especially useful is Internet. Services like Google or Facebook try to reduce the number of HTTP requests by batching operations together. Google [11] uses batch requests in Google Base [7], Google Spreadsheet [6], Google Calendar [8] and Google Cloud Storage API [9]. Specifically, the Google Cloud Storage API provides with batch requests to batch API calls together and reduce the number of HTTP connections clients have to make.

In a similar vein, Facebook provides its Ads API [1] and Graphics API [3] with batch requests to send several requests of the same type in a single HTTP request. Depending on the type of operation, the maximum number of requests per batch operation varies.

6.2. Design

Batchops embed in a single packet hundreds or thousands of entries that perform the same operation on a given directory. Figure 6.1 shows the formats of a regular operation and of a batch operation. Each batchop results in a single packet that includes: operation type, directory name, list of directory entries, and operation parameters (including semantics).

Note that, in the case of batch operations, the directory name is specified separately, since is the same for all the entries. Batch format also includes the field `semantics` that tells how to act on failures. `semantics` can have two values: *perform all operations*, or *stop on failure*. The first option tells the server to perform the operation on all the entries regardless of their success. The second option tells the server to stop on the first failed operation.

Once the packet is received on the server's side, all the operations are processed taking into account the semantics parameter. As reply, the server returns the errno values returned by the performed operations, and, if any, another extra information.

Therefore, semantics not only inform servers about the operation, but also inform clients about the reply they will receive. Figure 6.2 shows the reply packet format for a `stat` operation. First field after the operation type is the number of performed operations, which is also the number of elements in the list of *errno* values. Note that with semantics perform-all-operations that size will be the same as the number of requested entries. Next field is the number of successful operations. Following to that field is the *list of errnos*, which contains the returned errno value for each performed operation. Next, there is the field *list of infos*, which contains the stat information for each file. The length of that list is the same as *#succ values*.

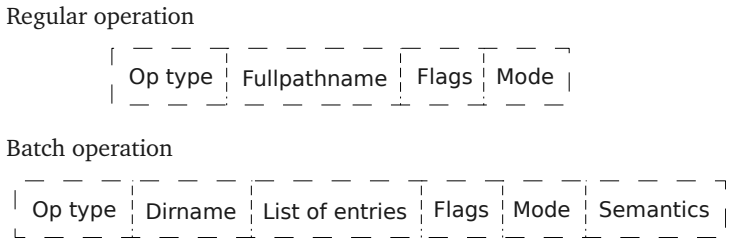


Figure 6.1: Request packet format.

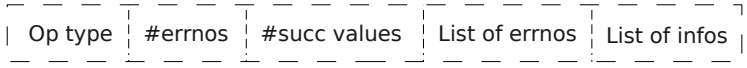


Figure 6.2: Reply packet format for a `stat` operation.

The `stat` operation returns extra information besides the *errno* value. For the remaining operations, however, the packet formats is the same but without the extra fields *#succ values* and *list of infos* `stat` needs.

6.3. Implementation

Operations supported by our implementation of batchops are: `openv`, `closev`, `statv` and `unlinkv`. All of them, except for `closev`, follow the packet format described in the above section. In the case of `closev`, instead of a directory name and list of file names, we send a list of open file descriptors. The reply follows the same format as the rest, with the lists of succesfull values and errnos values.

As explained in previous chapter, FPFS handles huge directories by storing them among a group of OSD+ devices. Therefore, bathops on hugedirs have to be handled differently than on regular directories. In order to exploit hugedir distributions, clients perform a batchop on a hugedir by sending batch packets in parallel to every OSD+ composing the hugedir. Each of those packets contain the directory entries of the original batch packet that are stored on the destination OSD+. Once the client receives all servers' replies, it sorts them in the same order in which they were initially requested.

Note that, in case of hugedirs, semantics stop-on-failure changes a bit. Since requests are sent in parallel to different servers, there is no way to know (or, at least not without losing parallelism) which operation failed "first". Therefore, the meaning of this semantics is modified for hugedirs, so that processing of requests will stop on the first failure in each OSD+.

To clarify the use of batchops on hugedirs, let's look at the example in Figure 6.3. A client performs an open batch request (`openv`) to open sixteen files on the directory `/home/usr3` (step 1), which is distributed. The client is already aware of the distribution of the directory and its corresponding distribution list (0 as routing, and 2, 4, 8, 10 as storing OSD+s) (step 2). Before sending the requests, the client calculates the storing OSD+ of each file through the distribution list and the distribution function of hugedirs (Equation 5.2 explained in previous chapter). Next, client is ready to send in parallel four open batch requests to the storing OSD+s (step 3). Once the servers perform the operations, they reply the client with the list

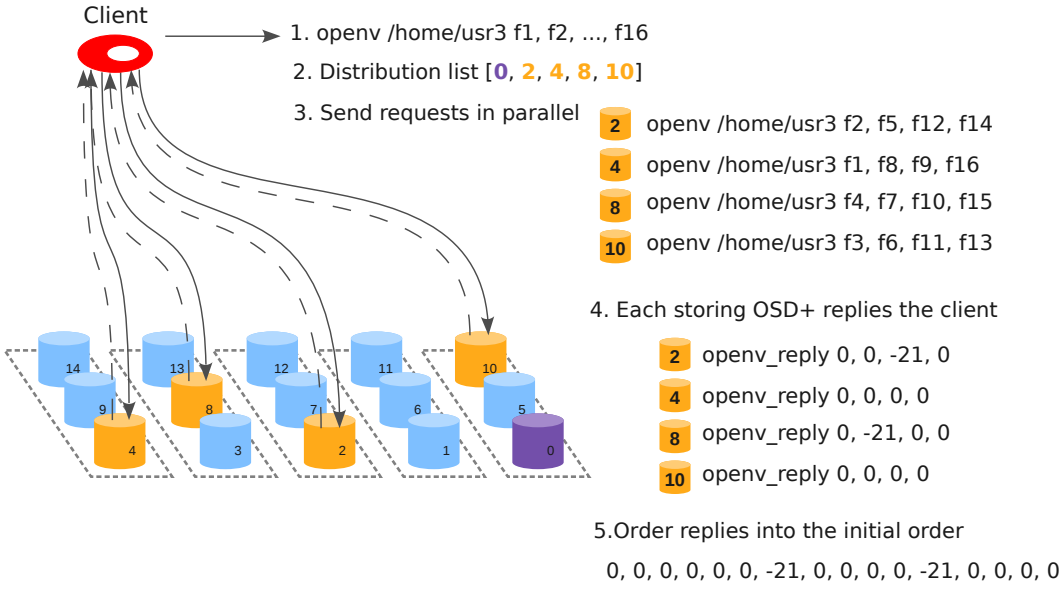


Figure 6.3: Example of a client requesting a batch open (`openv`) on a hugedir.

of returned outputs (step 4). Finally, the client sorts the replies in the initial order in which they were requested (step 5).

Note that an application does not need to know whether a directory is distributed or not in order to issue a batchop to it. The FPFS library takes care of the distribution, and transparently performs the requests in parallel and reorganizes the replies when a directory is distributed.

6.4. Experiments and Methodology

The testbed system and benchmarks are the same as those used in the previous chapter. See Section 5.3.1 and Section 5.3.2 for more information.

6.5. Results

In the experiments we evaluate the performance and scalability of batchops in FPFS. We use HDD-OSD+s and SSD-OSD+s as storage devices, and Ext4 and ReiserFS as backend file systems.

Through the experiments, we have analyzed four different aspects of the batchop support:

(a) Optimum number of operations per batch operation.
(b) Throughput and scalability for a single shared directory.
(c) Performance when several shared and non-shared hugedirs are accessed in parallel, and at the same time.
(d) Performance when there are one shared and one non-shared hugedir accessed concurrently.

FPFS performance and behaviour obtained for batchops with HDD-OSD+ devices and SSD-OSD+s are quite different. In results for SSD-OSD+ devices, batchops are usually beneficial, however, for HDD-OSD+ there are some cases where the performance decreases.

With HDD-OSD+ devices, more factors influence the performance, and, sometimes, it is not clear how batchops affect the results as a whole. For instance, initially, batchops provide some clear benefits: reduce the number of network operations and the network overhead, and increase the amount of operations per second sent to servers. Besides, batchops reduce the application time, which lessens the chance of a block being rewritten, and hence, they also reduce the number of writes to disk. However, batchops can also decrease performance for HDD-OSD+ devices because of the way files are allocated along the disk. On the one hand, when files are created with batchops, a set of i-nodes for the same client can be allocated together on disk. On the other hand, if files are created without bathops, i-nodes are more likely to be stored in an interleaved pattern. This two forms of allocation affect performance, mainly because of two factors: head-seeks and the disk cache usage.

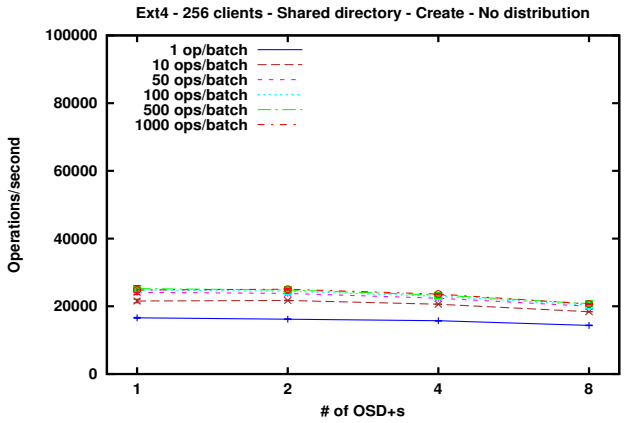
We performed some internal tests (not shown here) to see the behaviour of the stat and unlink tests after creating files with and without batchops. Also, we calculated the number of read and write operations for each of these configurations. In those tests, we could see that for read-only workloads (stat), having the files created in an interleaved pattern (non-batch) obtained better results due to the prefetching in the different caches, given that clients helped each other by bringing to cache i-nodes from other clients. Conversely, when files were created with batch, a client only helped itself in the stat test, reading mainly its i-nodes. The other clients had to read their i-nodes, stored in different disk areas, by themselves, and this caused larger head seeks. In the case of unlink, both read and write factors affected the test. There, batchops helped some configurations, but significantly downgraded performance in others. Reducing the time of the test by sending more operations to the servers allows us to reduce the number of writes, but we also need to consider the use of caches for reads in this test. Given all this, we could not always determine to what extent each factor affected.

Hence, we only show results with HDD-OSD+s in Section 6.5.2 for a single shared hugedir. For the remaining benchmarks, we only show results with SSD-OSD+ devices, as they always improved HDD-OSD+s' results, and, because the behaviour of batchops is more homogeneous with SSD-OSD+ devices. Moreover, results with SSD-OSD+s have an easier explanation given that there are less factors influencing the results (especially the head-seeks).

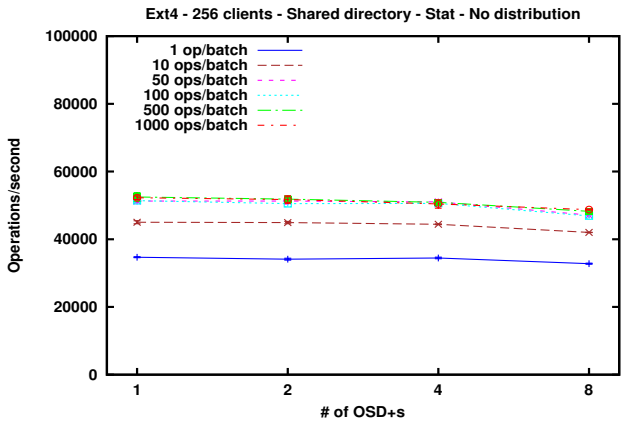
6.5.1. Batch operation's size

We start measuring the optimum number of operations embedded per batch operation. We perform the test where 256 clients create concurrently 400,000 files per OSD+ in a single directory. Figure 6.4 and 6.5 show the throughput in operations/s when not distributing and dynamically distributing the directory, respectively. The figures show the performance for 1 (equivalent to no batchops), 10, 50, 100, 500, and 1,000 operations per batchop. These tests use SSD-OSD+ devices, since the results for HDD-OSD+s are equivalent.

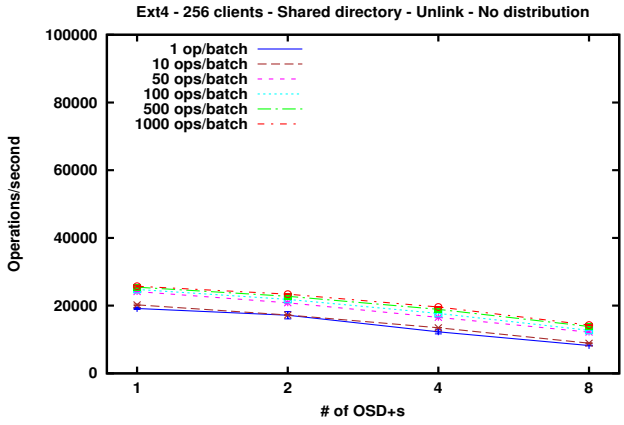
As we can see in Figure 6.4, when the shared directory is not distributed, with 1,000 ops/batch we increase the performance between 34% and 73% for Ext4 with respect to no-batch operations, and between 38% and 75% for ReiserFS, depending on the test and number of OSD+s. We also see that we already achieve almost the maximum possible improvement with only 50 operations per batchop, for both Ext4 and ReiserFS, and for any test. Note that in this configuration there is a single server receiving concurrent requests from 256 clients. Therefore, the server is saturated and more operations per batch cannot improve the performance further.



(a) create

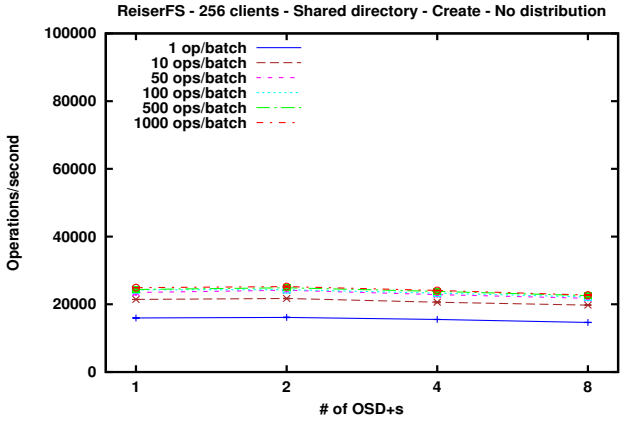


(b) stat

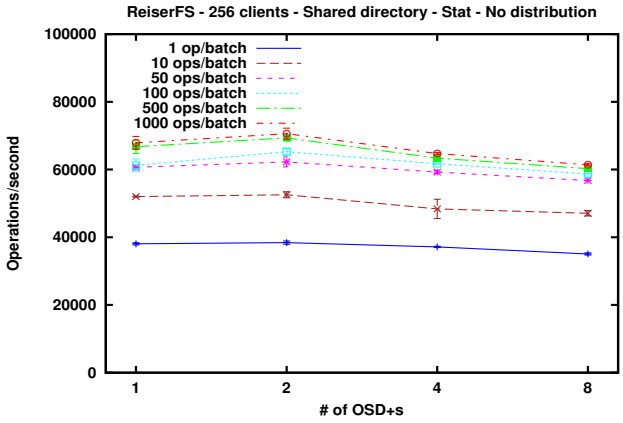


(c) unlink

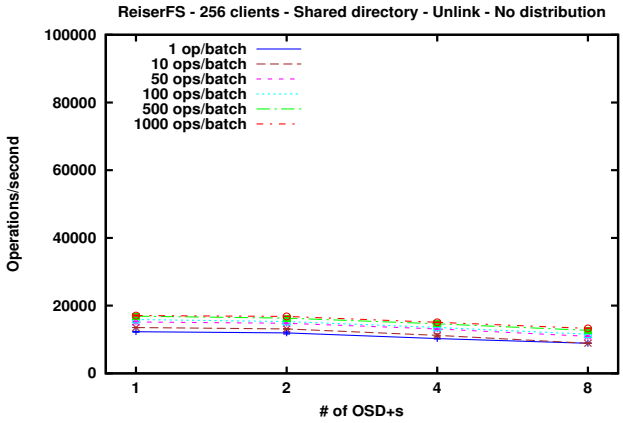
Figure 6.4: Operations per second obtained by FPFS with SSD-OSD+s and Ext4, when using one non-distributed shared directory and the number of operations per batch varies.



(d) create

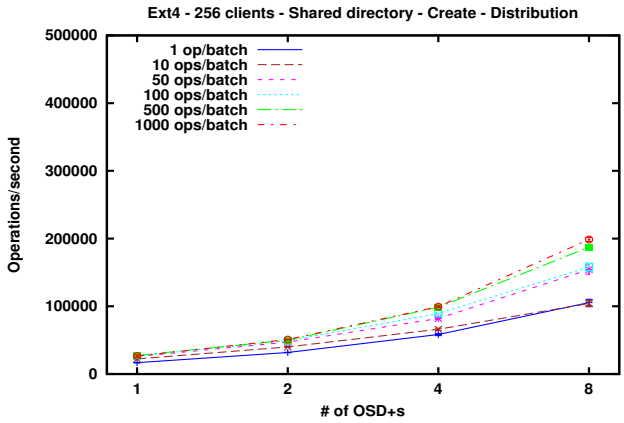


(e) stat

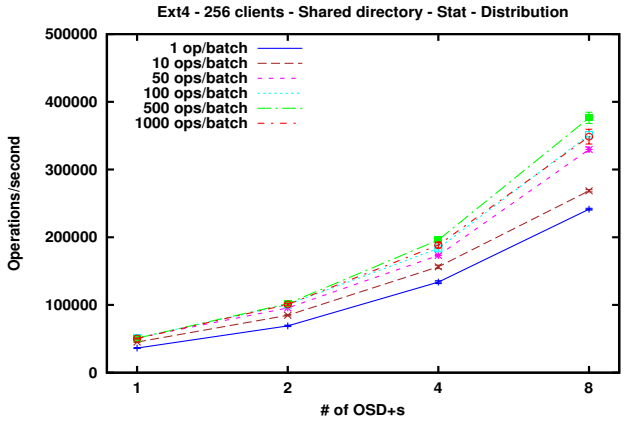


(f) unlink

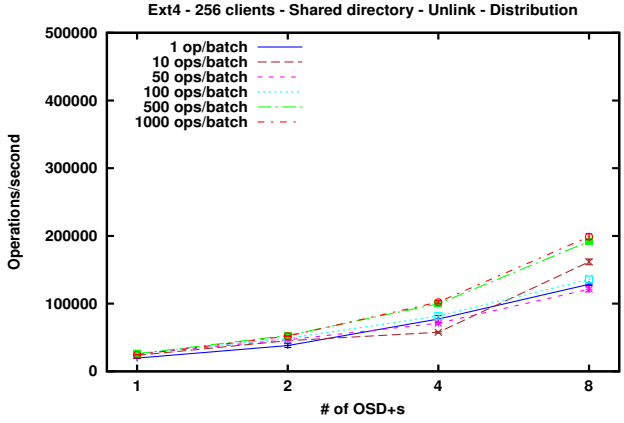
Figure 6.4: (Cont.) Operations per second obtained by FPFS with SSD-OSD+s and ReiserFS, when using one non-distributed shared directory and the number of operations per batch varies.



(a) create

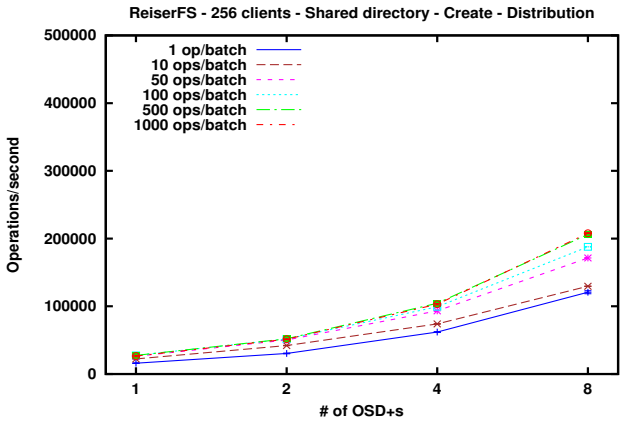


(b) stat

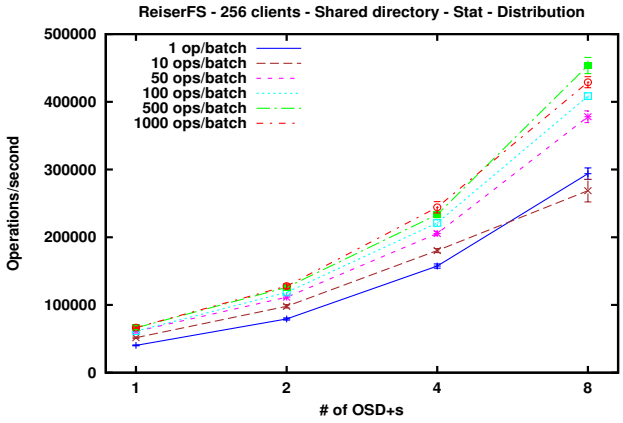


(c) unlink

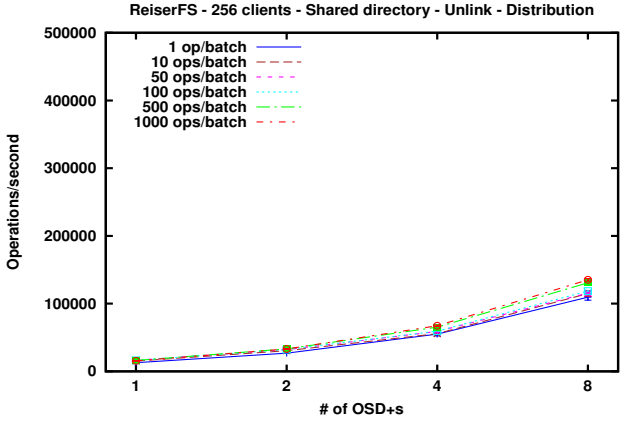
Figure 6.5: Operations per second obtained by FPFS with SSD-OSD+s and Ext4, when using one distributed shared directory and the number of operations per batch varies.



(d) create



(e) stat



(f) unlink

Figure 6.5: (Cont.) Operations per second obtained by FPFS with SSD-OSD+s and ReiserFS, when using one distributed shared directory and the number of operations per batch varies.

Regardless the number of ops/batch, the performance worsens when the number of OSD+s increases. A clear example is the *unlink* test on Ext4. This is due to the directory size, given that, the larger the number of OSD+s, the larger the directory (400,000 files * # OSD+s). As already explained in Section 5.4.2, Ext4 handles larger directories worse than ReiserFS, and, despite SSD-OSD+ devices help avoiding this problem, it is still noticeable in this test.

Figure 6.5 shows the results when dynamically distributing the directory. As in previous experiments, we use a dynamic distribution that shares out the directory when it exceeds 8,000 files. Once distributed, the directory object size becomes constant on each OSD+, resulting in a balanced workload. As we can see, the greater the number of operations per batch, the better. In general, the largest performance is obtained at 500 ops/batch.

Considering 1,000 ops/batch, the largest gains are obtained for *create* and *stat*. With Ext4, we obtain between 39% and 88% of improvement. With ReiserFS, between 46% and 72%. Note that, with both file systems and only 8 SSD-OSD+s, and thanks to batchops, we can produce more than 200,000 creates/s and 350,000 stats/s.

In the *unlink* test, there are also significant gains, but lower. With Ext4, we gain between 25% and 55%. With ReiserFS, we obtain improvements between 21% and 23%. In the case of Ext4, thanks to batchops, we reach 200,000 unlinks/s with just 8 SSD-OSD+ devices.

6.5.2. Single shared directory

This section compares the performance and scalability of batch and no-batch operations on a single distributed shared hugedir. Remember that, in this test, the hugedir is accessed by hundreds of processes at the same time to create, get the status and delete files. Also, we evaluate the effect of the directory size creating $F \times N$ files in the directory, where F is either 200,000, 400,000 or 800,000, and N is the number of OSD+s. Figures 6.6 and 6.8 show FPFS performance in operations/s obtained with HDD-OSD+ and SSD-OSD+ devices, respectively, when 256 clients access concurrently the hugedir. Figures 6.7 and 6.9 show the speedup achieved for the same devices.

HDD-OSD+

In this section, we show and analyze results for HDD-OSD+s and a single shared hugedir. As we advanced in the beginning of the section, HDD-OSD+ devices obtain a worse performance than SSD-OSD+s. Also, with HDD-OSD+s there are more factors involved in the results, and it was not always clear to what extent they affected the different configurations. Moreover, since the behaviour and performance observed here are repeated in the other tests carried out with HDD-OSD+, conclusions showed here can be extrapolated to a large extent.

For the create tests in Figures 6.6.(a) and 6.6.(d), we can see that batchops always perform better than no-batch. Namely, the improvement of batchops for Reiserfs is more than 40%, and with Ext4 the improvement achieves over 50% for 8 OSD+s. The configurations with no-batch suffer the network limitation in the create test, which we already showed in previous chapter. In Section 5.4.1, we detailed the four network packets this test generates in FPFS: two (request and reply) for an `open` or `creat` call, and another two for closing the returned file descriptor. This significantly increases the amount of network packets compared to the other tests, and, therefore, batchops are more effective. Moreover, this benchmark only produces

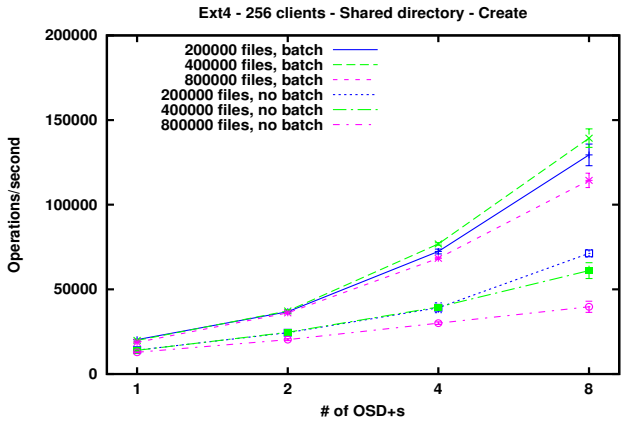
writes requests that go to cache, instead of directly accessing the disk. Therefore, batchops take a better advantage than no-batch, by sending more requests in each packet.

Yet, for stat and unlink, batchops are not always beneficial. As previously explained, performance of this test depends on the backend file system and device, and how files are created. With batchops, OSD+s receive large batchs of requests at the same time from each client, so every thread in the OSD+ will be able to create large amount of files close on disk on behalf of a client. With no-batch, files end up being created in an interleaved pattern. For instance, when the backend file system is Ext4, batchops cause i-nodes of the files created by a client to be stored together in a few disk blocks, which are not shared with the i-nodes of other clients' files; however, without batchops, disk blocks will be occupied by i-nodes of files created by different clients. The way files are created is key on how stat and unlink tests perform afterwards, mainly because of the head-seeks latencies, and the use of the disk cache, as we have already explained.

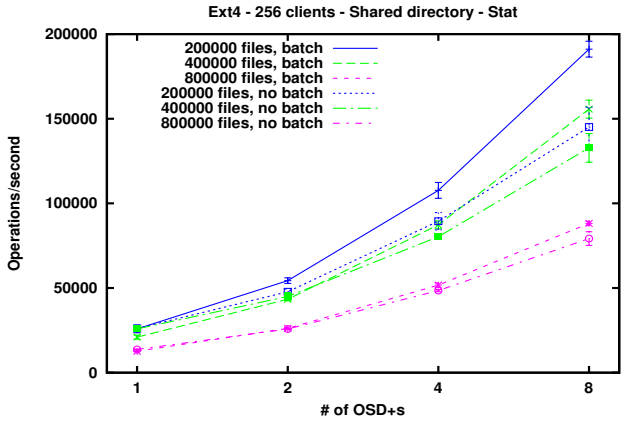
For stat and Ext4, batchops improve performance compared to no-batch for small objects (200,000 files) and for 4 and 8 OSD+s for larger objects, where the reduction of network traffic and the higher number ops/sec are more noticeable. For ReiserFS, batchops also improve for small objects, and for 400,000 with 4 and 8 OSD+s, but not for 800,000 files. In general, as directory objects are larger, batchops performance decreases mainly because of the increase in head seeks, and the poor use of caches compared with no-batch. To understand this, remember the way files are created and then read. When we use no batch, files are in an interleaved pattern. Each disk block read by a client will probably help other clients because it will probably contain some of their i-nodes. A side effect of this is that clients roughly proceed at the same pace, so disk heads usually move forwards and disk caches are more efficiently used too. When using batchops, each client only helps itself, so the other clients have to issue read requests that produce large head seeks forwards and, what is worse, backwards, incurring in large latencies. This behaviour is more noticeable as directory objects grow, and, specially, in ReiserFS where batchops get to perform 60% worse than without batchops.

In the unlink test, we have two different behaviours depending on the file system used (see Figure 6.6.(c) for Ext4 and Figure 6.6.(f) for ReiserFS). Ext4 benefits from the batchops from 4 and 8 OSD+s in any case, while ReiserFS only benefits for 4 and 8 OSD+s when there are 200,000 files and 400,000 file per OSD+. As before, performance downgrades as the size of the directory grows. Several factors are intervening here. On the one hand, the frequency of commit intervals mentioned above. On the other hand, the type of workload, that mixes reads (as those issued by stat) and writes. As we have just seen, both Ext4 and ReiserFS obtain downgrades with batchops for some configurations in stat, and this problem also affects reads in this test.

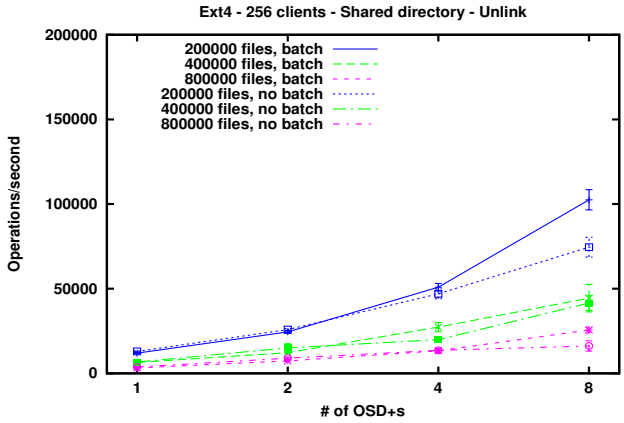
However, when it comes to writes, Ext4 benefits from batchops, since many disk blocks are entirely modified in a short time (e.g., blocks full of i-nodes of files of the same client), and are usually written to disk only once despite the frequent commits in Ext4. This reduces the duration of the test, which also reduces the chance of a block of being modified several times, and, therefore, it reduces (again) the amount of writes. Without batchops, a disk block (e.g., a block with i-nodes from different clients) can be modified at different moments, and written to disk several times. This increases the number of writes and head seeks, so the test takes longer.



(a) create

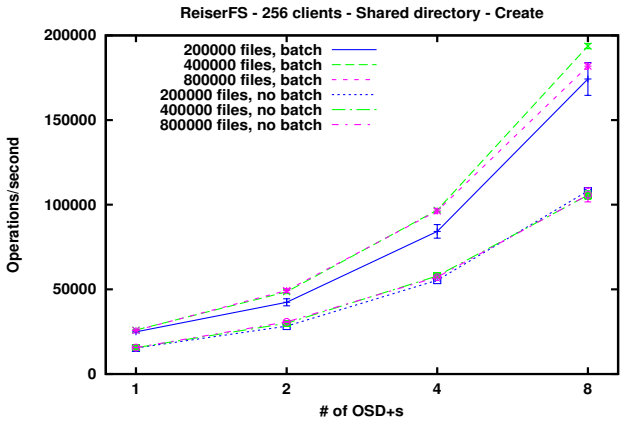


(b) stat

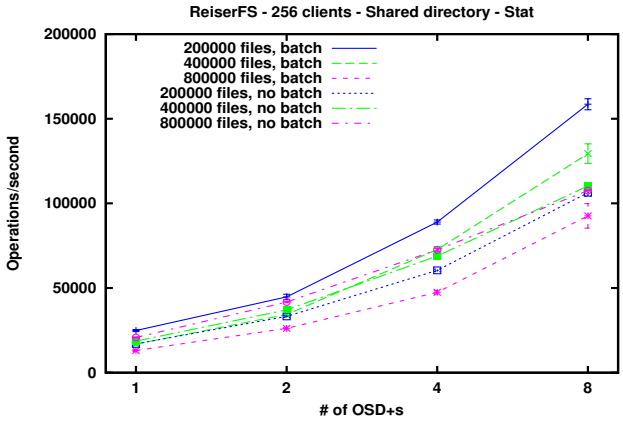


(c) unlink

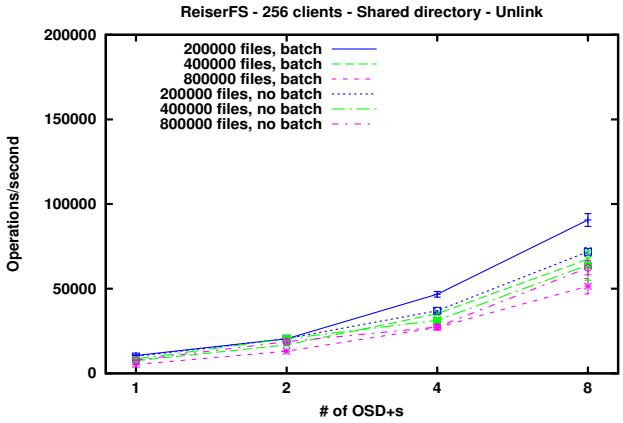
Figure 6.6: Operations per second obtained by FPFS with HDD-OSD+s and Ext4 when using one shared directory.



(d) create

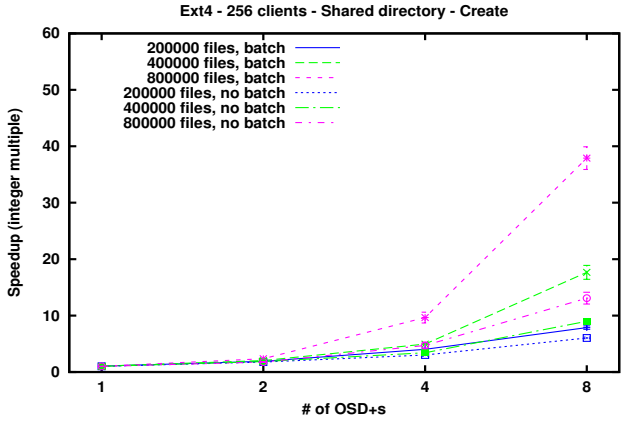


(e) stat

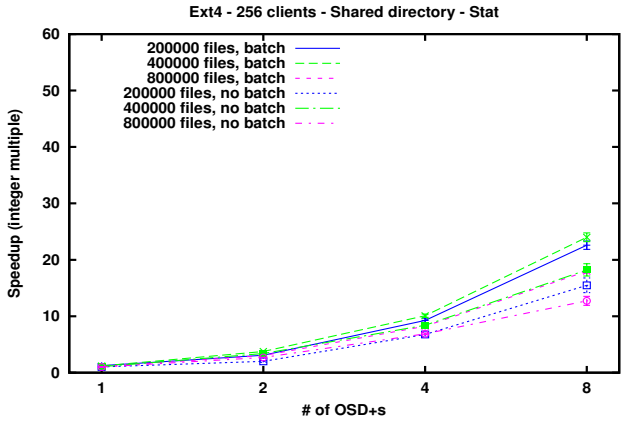


(f) unlink

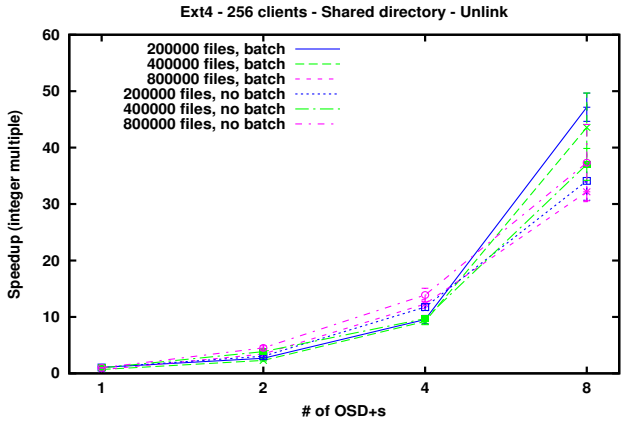
Figure 6.6: (Cont.) Operations per second obtained by FPFS with HDD-OSD+s and ReiserFS when using one shared directory.



(a) create

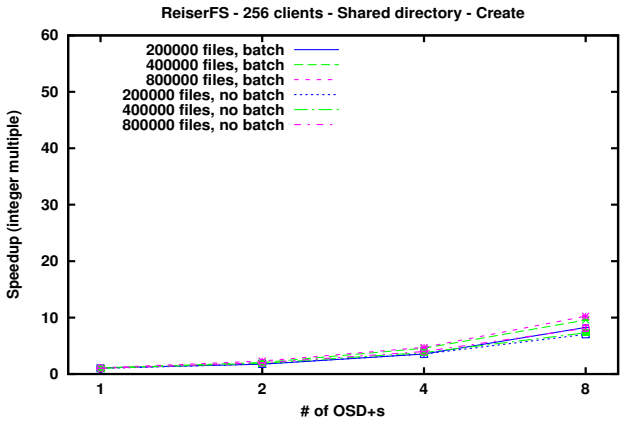


(b) stat



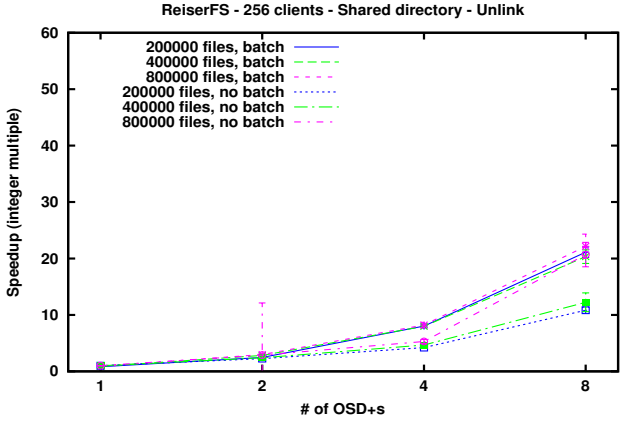
(c) unlink

Figure 6.7: Scalability obtained by FPFS with HDD-OSD+s, and Ext4 as file system when using one shared hugedir.



(d) create

(e) stat



(f) utime

Figure 6.7: (Cont.) Scalability obtained by FPFS with HDD-OSD+s, and ReiserFS as file system when using one shared hugedir.

ReiserFS also reduces the number of writes, but its performance is greatly determined by its behaviour in the stat test. Therefore, we obtain better results with batchops as directories are smaller (200,000 files per OSD+). Also, ReiserFS uses a B-tree+ to store the directory and, apparently, that tree produces a more random pattern to place files on disk, which later produces a worse use of caches.

For HDD-OSD+s, scalability is super-linear, and usually better that with no-batchops. Remember that the speed-up is calculated comparing the performance obtained when not distributing and when distributing the files in a single shared directory (we need to compare directories of equal size). As we mentioned before, performance is worse as directories grow, therefore for non-distributed configurations performance with batchops significantly downgrade. Scalability for ReiserFS is usually smaller than for Ext4, because ReiserFS is less sensitive to the directory size. For example, Ext4 achieves a scalability over than 30 for unlink, while ReiserFS slightly exceeds 20. Also, in the create test, Ext4 achieves the largest speed-up for the largest objects (800,000 files and 8 OSD+).

SSD-OSD+

Results show that batchops perform better than no-batchops for all the tests. As already said in the previous section, batchops are specially helpful for the *create* test, because they reduce the network traffic. Also, with batchop, we noticeably increase the number of requests/sec for each OSD+, by sending more requests to each server in each packet and, also, sending them in parallel to all the servers. Thanks to batchops, FPFS is always able to improve performance by 50% at least, doubling the number of operations per second in some cases.

Ext4 obtains a larger improvement with respect to the results obtained with HDD-OSD+s, due to the worse performance of Ext4 on HDD-OSD+s as directory sizes get larger. With SSD-OSD+ devices, we improve the number of operations per second by 30% for batchops and 400,000 files/OSD+ respect to HDD-OSD+ devices, while ReiserFS only improves by 5%.

In the *stat* test, both Ext4 and ReiserFS improve performance with batchops by 25% at least. The improvement is smaller than in the *create* test because the reduction in network traffic is less, since stat already produces half the network traffic than create.

Finally, in the *unlink* test, the backend file system determines results to a great extent (as we already explained in Section 5.4.2). The effect of the batchops also seems to depend on the backend file system, being Ext4 the file system that better leverages batchops. Specially for large directories, Ext4 performs a 60% better with than without batchops, while ReiserFS only achieves a 23% of improvement. We believe that this is because batchops cause a better use of the different caches when Ext4 is the local file system. Batchops allow the serving threads in the storage nodes to carry out a request immediately after the previous one, without waiting for a new request from a client after serving a request. This specially helps Ext4, which reads and writes more blocks than ReiserFS. For 800,000 files, Ext4 exceeds RAM capacity, and using batch helps reducing the number of written blocks. By writing less, we also improve the reads performance, since there is less competence for disk. Also, Ext4 commits to disk every 5 seconds, reducing the possibility of a block being re-written. In the case of ReiserFS, it does not exceed the maximum capacity of RAM for our tests, and its commits are each 30 seconds. Batchops still provide some benefits, but they are less noticeable.

Therefore, disk blocks in the buffer cache, fetched during the processing of a request, are likely to be used in the next request of the same thread before being evicted by requests of other threads. ReiserFS provides a smaller benefit. As mentioned in Section 5.4.2, ReiserFS produces a quite "random" access pattern from a cache's point of view [42]. This limits the improvement that can be obtained from the "aggregate disk cache".

According to Figure 6.9, batchops hardly affect scalability. For the create test, the most noticeable change is for 800,000 files per OSD+, and 8 OSD+s, where scalability is super-linear. In the stat test, however, batchops slightly reduce the scalability for ReiserFS, and for unlink it remains super-linear for both Ext4 and ReiserFS. With batchops, we significantly reduce the amount of network traffic, specially when the directory is not distributed. Therefore, when we distribute a directory with batchops, the network reduction is not as high as the one achieved with no-batchops.

These results diverge from the ones with HDD-OSD+s, where batchops significantly increased the scalability. While, with batchops and HDD-OSD+s, the directory size greatly affected several tests, with SSD drives, again, we remove all the head-seeks that provoked this increment.

6.5.3. Multiple Hugedirs

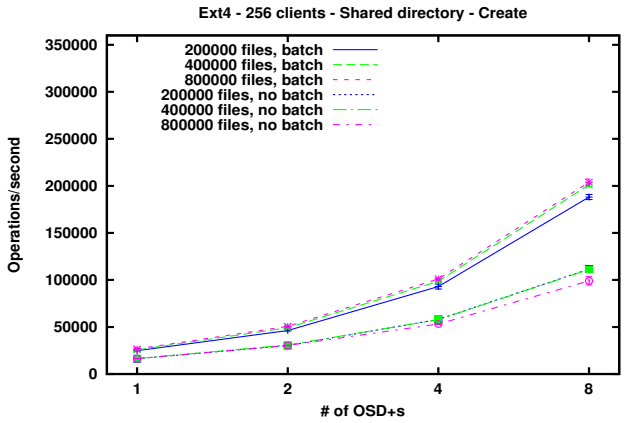
Table 6.1 shows, for each number of processes per directory, absolute application times when hugedirs are never distributed in the first column. Following columns show relative application-time variations, in percentage, with respect to the absolute times, when hugedirs are distributed dynamically (i.e., when a directory exceeds 8,000 files), and always (i.e., when threshold is 0). Confidence intervals (not showed) are smaller than 10% of the mean. Note that a positive/negative percentage means an increase/decrease in time (and, hence, a worse/better performance).

In this section, we show results for batchops. Therefore, to see the effect of batch against no-batch, we need to compare Table 6.1 to Table 5.4 in previous Chapter 5.

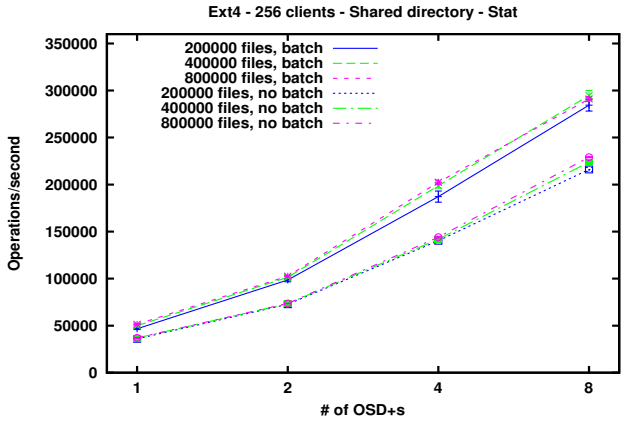
The first thing we can observe is that, when comparing absolute times in columns *never*, batchops improve performance in general (for both Ext4 and ReiserFS), specially when there is one client per directory. When directories are distributed, results obtained by batchops are more variable, as it already happens with non-batch operations, and they also depend on the number of OSD+s, number of processes per directory and backend file system. However, there are some noticeable differences now.

For Ext4 and *create*, distribution and batchops improve results with respect to *never* when there is 1 client per directory, but slightly worsen them when the number of clients per directory grows. However, absolute times are inferior now in any cases. For *stat*, results are comparable with those without batchops, except for 1 client per directory and 8 OSD+s, where the distribution with batchops increments the application time from 5% to 56%. For *unlink*, distribution with batchops behaves much better than without batchops, and now there is only a small increase in the application time. Moreover, with 8 OSD+s, batchops are able to significantly reduce the application time. Exception appears for 1 process per directory and 2 OSD+s, although, considering absolute times, batchops still reduce the application time considerably.

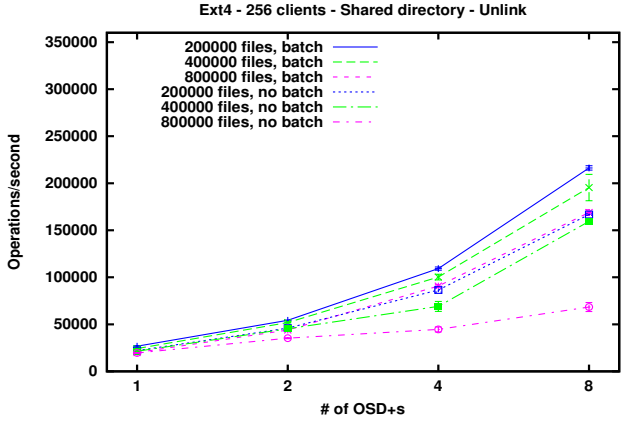
For ReiserFS as backend file system and *create*, the behaviour is similar to that of Ext4. For *stat* and 32 clients per directory, results are comparable to those we had without batchops.



(a) create

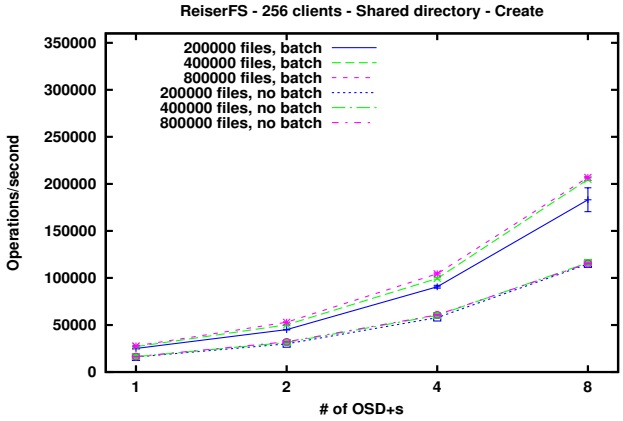


(b) stat

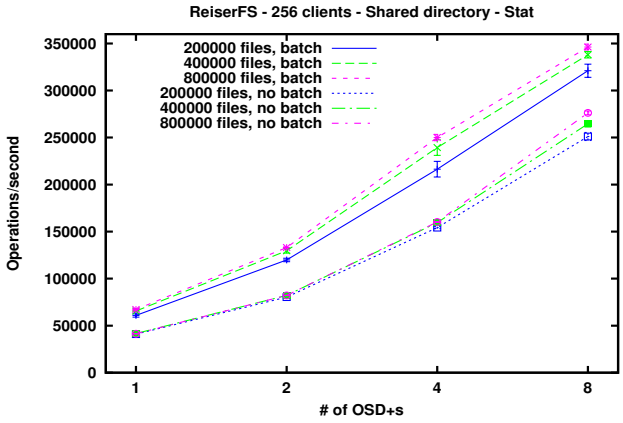


(c) unlink

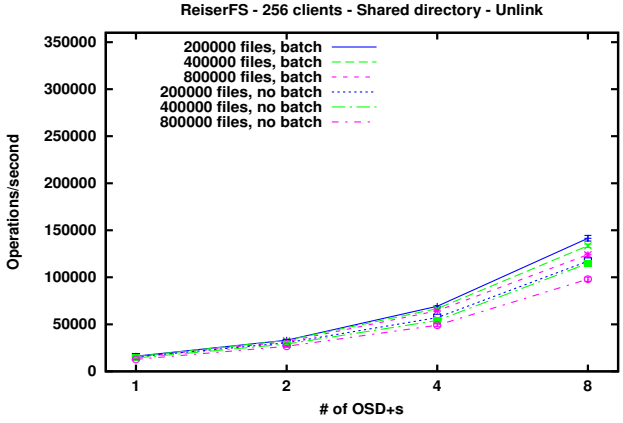
Figure 6.8: Operations per second obtained by FPFS with SSD-OSD+s and Ext4 when using one shared directory.



(d) create

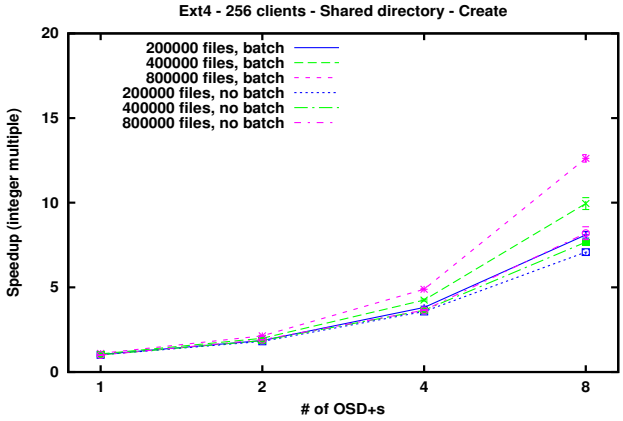


(e) stat

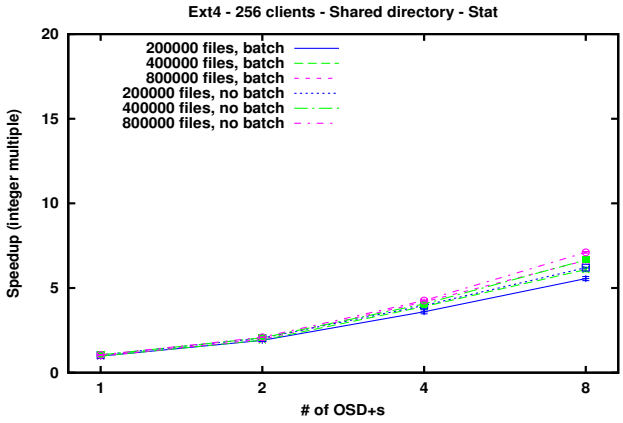


(f) unlink

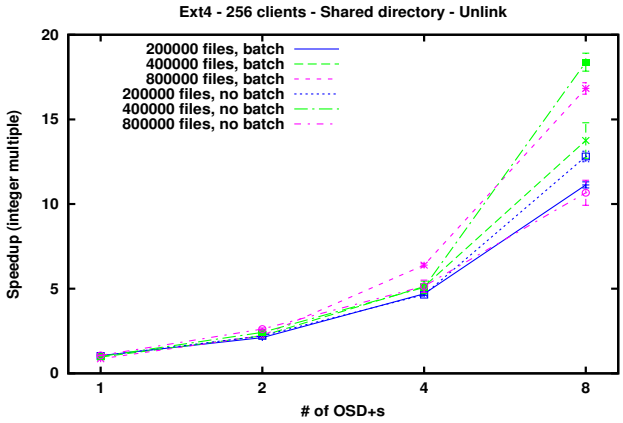
Figure 6.8: (Cont.) Operations per second obtained by FPFS with SSD-OSD+s and ReiserFS when using one shared directory.



(a) create

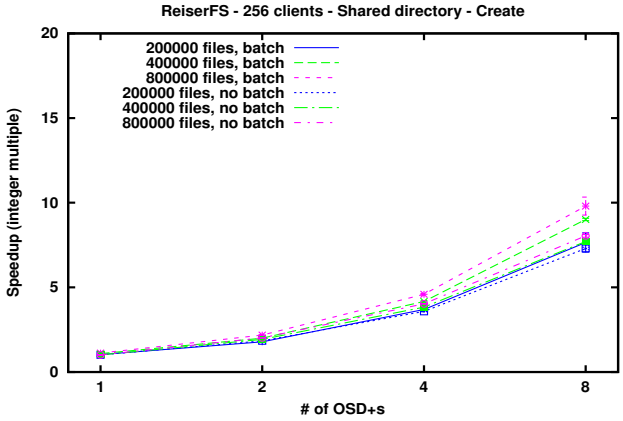


(b) stat



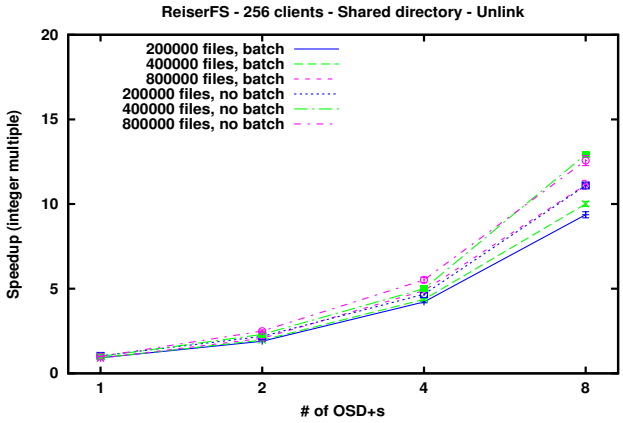
(c) unlink

Figure 6.9: Scalability obtained by FPFS with SSD-OSD+s, and Ext4 as file system when using one shared hugedir.



(d) create

(e) stat



(f) utime

Figure 6.9: (Cont.) Scalability obtained by FPFS with SSD-OSD+s, and ReiserFS as file system when using one shared hugedir.

Table 6.1: Performance obtained by FPFS on SSD-OSD+ devices with batchops, and Ext4 and ReiserFS, when 8 hugedirs are accessed concurrently.

(a) Ext4

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)
create	1	74.32	-2.14	-2.14	75.28	-2.40	-2.31	74.10	-4.34	-3.75
	2	37.44	6.31	1.78	37.27	15.06	11.63	37.15	17.52	33.45
	4	23.01	-4.55	-10.35	20.97	26.05	21.72	21.38	24.75	21.45
	8	16.27	-33.73	-34.73	13.52	13.78	9.12	13.63	18.96	10.97
stat	1	28.29	-0.24	-0.61	17.96	3.74	4.13	18.58	2.39	7.80
	2	24.74	-8.44	-9.03	10.77	-3.11	0.61	12.35	-7.71	-6.73
	4	22.04	6.37	7.43	9.23	-5.80	-1.53	9.68	-5.35	-9.98
	8	19.69	56.67	55.74	8.28	-5.28	-3.49	8.69	-2.83	-6.43
unlink	1	133.07	-1.89	-4.21	201.93	-12.36	-11.80	214.32	-9.46	-7.33
	2	33.84	56.10	46.47	53.04	6.13	8.27	58.63	15.72	8.39
	4	20.08	23.33	3.73	20.53	2.02	-1.09	21.27	19.41	13.85
	8	13.12	-26.83	-30.60	12.63	-26.01	-26.09	12.79	-28.34	-29.10

(b) ReiserFS

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)	Never(s)	Dynamic(%)	Always(%)
create	1	85.54	0.22	-0.33	94.57	-3.41	-2.35	100.71	-11.75	-11.41
	2	42.29	-0.71	-4.97	43.91	18.67	14.06	40.81	24.78	16.95
	4	24.68	-15.96	-19.14	20.59	35.66	32.90	20.01	14.55	25.46
	8	16.61	-32.26	-35.82	13.20	18.61	13.98	12.89	7.29	-6.86
stat	1	25.01	-1.03	0.45	23.83	-0.62	-0.64	23.08	1.86	2.00
	2	21.57	-4.80	-5.87	11.69	3.17	10.65	12.60	-1.41	2.68
	4	19.33	17.54	17.33	8.23	2.45	0.19	8.24	-1.18	5.72
	8	18.38	64.06	65.24	6.78	16.83	18.65	8.00	1.34	1.77
unlink	1	141.00	-1.98	-2.66	144.73	0.41	-1.48	150.40	-3.10	-3.38
	2	70.91	-5.65	-6.44	71.51	-1.62	-4.76	71.94	-3.65	-4.28
	4	32.16	2.90	2.97	34.64	0.60	0.22	35.11	-3.48	-4.31
	8	18.33	-8.13	-7.57	18.06	-2.87	-5.96	18.31	-5.76	-4.00

For 1 and 8 clients per directory and for 8 OSD+s (and, sometimes, 4 OSD+s), distribution increments times respect to *never* more than when we did not have batchops. For *unlink*, differently to what happened with no-batch, distribution and batchops reduce the application time with respect to *never*.

In summary, batchops allow in general to reduce the application times for distributed directories. We believe this is because the threads attending requests in the servers can process more requests in a shorter time. This improves caches' performance and reduces the overhead produced by disk contentions.

6.5.4. Mixed directories

Figure 6.10 depicts the throughput in operations/s achieved by FPFS with SSD-OSD+ devices for this test. The graphs show the performance with and without batchops of a distributed and a non-distributed directory, each storing 1,280,000 files. As we can see, batchops always improve the performance of both directories in all cases, and, as in a single shared directory, the reduction in network traffic and a better use of the caches explain the improvements.

In the *create* test, batchops achieves an improvement of more than 30% for both the non-distributed and distributed directory, and both Ext4 and ReiserFS, due to the reduction in network traffic.

Batchops obtain the best improvements in the *stat* test. For the non-distributed directory and Ext4, batchops improve the throughput by 34% at least, and by 44% with ReiserFS. In the case of the distributed directory and Ext4, batchops achieves a maximum improvement of 36%, and with ReiserFS the improvement reaches a 40%. Since this is a read-only test, which reads related directory entries and i-nodes, batchops allow servers to make a better use of caches and prefetching, because they process many requests in a row.

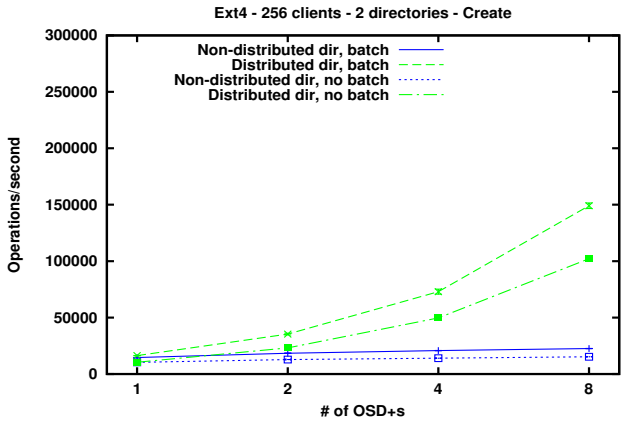
Finally, results in the *unlink* test are similar to those in Section 6.5.2 where Ext4 performs better than ReiserFS as the number of OSD+s increases. For the distributed directory, Ext4 achieves a 40% of improvement with 8 OSD+s, while ReiserFS gets 16%. For Ext4 and the non-distributed directory the improvement is around 30%, and, in the case of ReiserFS the improvement is around 25%.

6.6. Conclusions

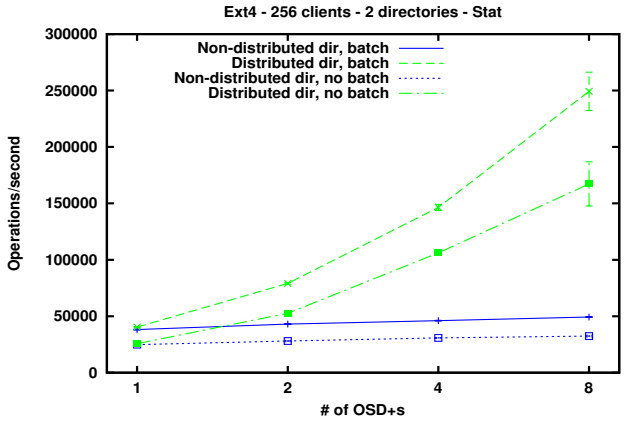
Workloads that perform the same operation on multiple files, such as the migration of a directory, the creation of a set of files in a directory, or the removal of all the files in a directory, usually incur in large amounts of network traffic to perform the same type of operations. In order to deal with these workloads in a more efficient way, this chapter presents the design and implementation of operations that embed hundreds or thousands of entries of the same type of request into a single packet. With these operations, that we call batchops, we greatly reduce the amount of network packets, and, therefore, network delays and round-trips. We also manage to reduce the overall network congestion, making a better use of the available I/O and processing resources.

We add the management of batchops to the FPFS library by including specific operations to create, stat and unlink files in a batch fashion. For each operation, we modify the packet format to include a list of entries within the same directory. Our batch operations include semantics to specify the behavior in case of failure of an operation in the batchop. The implementation also supports huge directories in a transparent way (clients do not need to differentiate between distributed and non-distributed directories when issuing batchops).

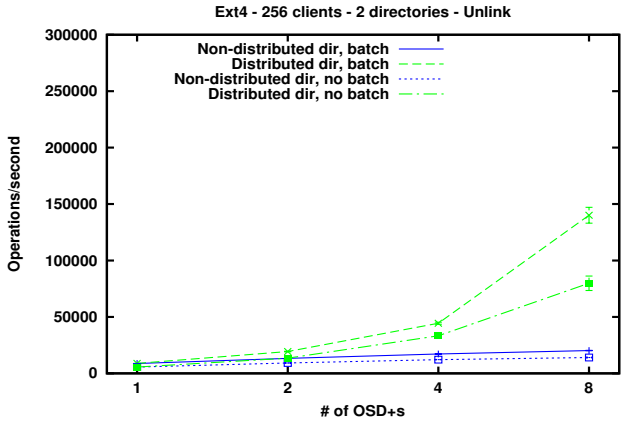
The experiments show that batchops help us to reduce the network overhead, and increment the number of operations/s in OSD+s, improving FPFS performance in the experiments evaluated in the previous chapters. Specifically, in tests that make a more intensive use of the network, such as the creation of a single shared directory, performance improves by a 50% at least, doubling the number of operations per seconds in some cases. In the case of stat, the



(a) create

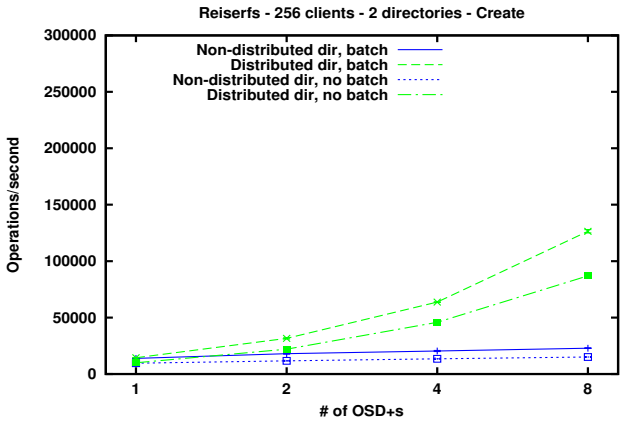


(b) stat

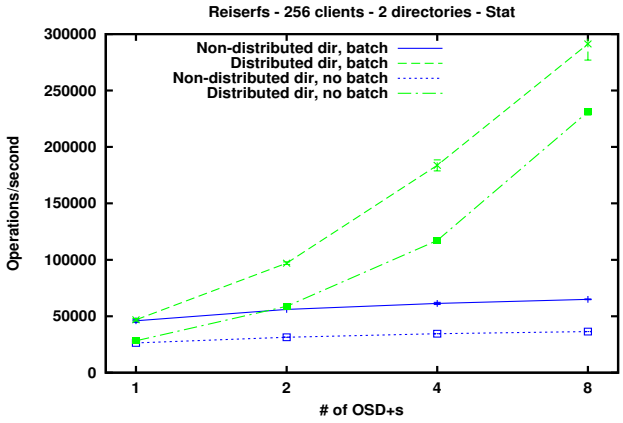


(c) unlink

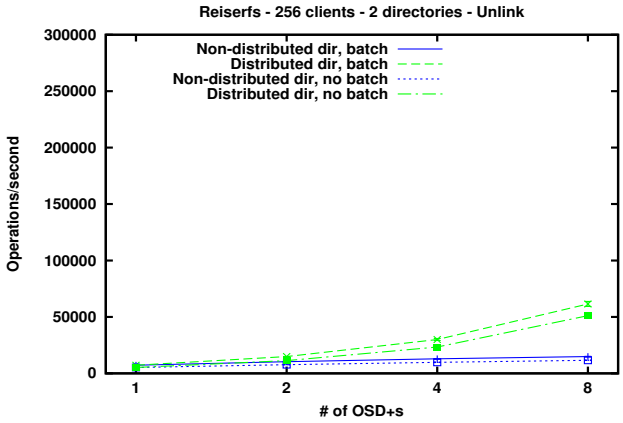
Figure 6.10: Operations per second obtained by FPFS with SSD-OSD+s and Ext4 when a distributed hugedir and a non-distributed hugedir are concurrently accessed.



(d) create



(e) stat



(f) unlink

Figure 6.10: (Cont.) Operations per second obtained by FPFS with SSD-OSD+s and ReiserFS when a distributed hugedir and a non-distributed hugedir are concurrently accessed.

improvement is always around 25%. Finally, for the unlink test, which issues both read and write requests, the backend file systems determines results to a great extent, being Ext4 the one that achieves a larger improvement of 60%, while ReiserFS obtains a 23% when using batchops.

Chapter 7

Conclusions and Future ways

Let us finish this thesis by summarizing the work presented and considering further research directions.

7.1. Conclusions

As the performance and scalability requirements of parallel file systems increase, we need simpler and more decentralized architectures. As we have seen, an efficient metadata management becomes a fundamental aspect of a system's storage architecture to prevent bottlenecks, and achieve the desired features of high performance and scalability. In order to provide such an efficient metadata management, we have proposed a new file system and related metadata service based on OSD+ devices. Following, we describe our mayor contributions.

FPFS

Firstly, we have designed the Fusion Parallel File System, a file system that merges the management and storage of data and metadata into the same type of server. In this way, clients can massively perform in parallel metadata and data requests by using all the nodes in the cluster.

This design simplifies the complexity of the storage architecture, merging the data cluster and the metadata cluster into the same nodes. Also, it makes a better use of the existing resources, since we do not need two different type of servers. Usually, metadata clusters are no larger than a dozen of nodes. With our architecture, we enlarge the capacity of the metadata cluster, which becomes as large as the data cluster, increasing the overall system capacity and scalability.

OSD+

Secondly, we have described a new OSD device, called OSD+, that supports the FPFS architecture. An OSD+ is an enhanced OSD that, apart from handling data objects, is able to store and manage metadata objects through what we call directory objects. We have extended the OSD interface in order to manage directory objects and its related operations. In addition, we have assured that directory operations are atomic.

OSD+ devices profit the existence of a local file system in the storage nodes, and directly map directory-object operations to directory operations in the underlying system. Hence, we export many features of the local file system to the cluster file system, achieving a great flexibility, simplicity and small overhead.

By using OSD+s, data and metadata of a parallel file system can be managed by all the OSD+s in a cluster, improving the overall throughput.

Metadata cluster

The third contribution has been the design and implementation of the FPFS metadata cluster. Having a metadata cluster as large as the data cluster increases the overall capacity and scalability of the system, but it also entails challenging issues, such as namespace distribution, workload balance, and object migrations.

We hash directory names and use a pseudo-random distribution function to have a balanced distribution of directory objects, and to minimize the migration of objects when cluster changes occur. Hashing approaches usually suffer from massive migrations in case of renames. However, we highly reduce the number of migrations by only hashing directories. Also, we manage the migrations in a lazy fashion that allows us to delay the movement of metadata, avoiding flooding the system. In a similar vein, we leverage the rename management to tackle with symbolic links, so no extra mechanism is needed.

Atomicity of metadata operations involving several OSD+s is guaranteed through a network-commit protocol, while the local file system in each OSD+ guarantees atomicity for operations on a single directory.

The implemented prototype improves Lustre's performance by a 60–80%, and shows that FPFS performance scaled with the number of OSD+s, being super-linear in some cases. Experiments also show that the underlying file system and formatting options can affect the system's performance. However, since FPFS can use any file system supporting extended attributes as backend file system, it is easy to set up the system to obtain the best performance.

Huge directories

The fourth contribution has been the management of huge directories that store millions of entries accessed by thousands of clients at the same time. We leverage directory objects to store huge directories among several servers while maintaining POSIX semantics. Directory objects supporting a huge directory work independently, achieving good performance and scalability.

We dynamically distribute a hugedir among several OSD+s when it surpasses a given number of files. The enhanced version of the directory objects allows to optimize the redistribution of existing directory entries. Also, it avoids massive metadata migrations when a huge directory is renamed, given that a rename only involves a change of roles between two nodes.

The evaluation shows that FPFS exceeds today's requirements of HPC applications regarding huge directories: a billion files per directory, more than 40,000 files created per second, etc. FPFS achieves a high throughput of more than 80,000 creates per second, 100,000 stats per second, and 70,000 unlinks per second for huge directories on a cluster with just 8 HDD-OSD+s, and a super-linear scalability as the number of OSD+s increases. Moreover, these numbers grow when we use SSD-OSD+ devices, which do not suffer the limiting factor of a small number of IOPS as current hard drives do. FPFS increases its throughput to more than 110,000 creates per second, more than 200,000 stats per second, and more than 100,000 unlinks per seconds with 8 SSD-OSD+ devices.

Experiments, however, have produced unexpected results too. While distribution is beneficial when a huge directory is accessed by many clients, it can also downgrade the performance

when a few clients concurrently access several huge directories. A consequence of the results is that, we need to find better ways to decide when to split directories.

Last, we compared FPFS performance with OrangeFS, which developed an experimental version that supported the static distribution of huge directories among several servers. OrangeFS is able to create and delete around 9,000 files per second, and stating 10,000 with SSD drives. FPFS results increase these rates at least by an order of magnitude.

Batch operations

Our last contribution is the design and implementation of batch operations, which embed several entries of the same type into a single packet. Procedures such as the migration of directories that move files from one server to another, deletion of all files in a directory, or creation of a group of files, are workloads that can be perform in batches. With this processing, we reduce the number of network packets, and save network overheads and delays, while we make a better use of existing resources by increasing the number of operations per second that servers receive.

Current batchops support creating, stating and unlinking files, and are included in the FPFS library. Our implementation also supports batchops with huge directories, in a completely transparent way for the clients, so the same operations work for all types of directories. Batchops include a semantic field to specify the behaviour in case of failure of an operation: whether to stop or continue with the remaining ones.

Results of our experiments show that batchops reduce the network traffic, significantly improving workloads with high network requirements, as our create tests, which improves performance by 50%. For the other workloads, the improvements go from 23% to 60%, depending on the type of workload and backend file system.

7.2. Future Work

The work presented in this dissertation can be completed and extended in different directions. We describe here those we currently find more interesting.

Although we usually run 256 clients in our experiments, several internal tests reveal that FPFS obtains the highest rate of operations per second for 32/64 clients. Our OSD+ implementation creates a thread for each new connection. As the number of clients grows, so does the number of threads, degrading at some point the OSD+ performance. Moreover, drives are only able to sustain a certain amount of parallel I/O activity before performance is degraded, due to the high number of seeks and threads waiting for I/O [87]. Therefore we plan to make the number of threads in the OSD+s independent of the number of clients, and evaluate the performance of FPFS when the number of threads per OSD+ varies.

On the design of the metadata cluster, we can currently access any directory object by hashing its full pathname. This fact, along with the use of pseudo-random deterministic distribution functions like CRUSH, allows us to minimize object migrations to a large extent, since only directories are affected by renames. As future work, our aim is to completely avoid migrations or minimize them even more, while still accessing any directory object directly. Note that, to achieve this goal, we could perform path resolutions component by component, which would allow us to map paths to fixed *ids*. This methods would not suffer the migration

problem, but would require clients to store path conversion tables, or to cache information that would need to be update on each rename. Our challenge is, however, to avoid this path resolution while still removing/minimizing object migrations.

As experimental results showed in the Huge Directories chapter, the splitting policy based on directory sizes does not adapt well to different workloads. Other factors such as the resource availability in the servers are more significant in the partition of directories. However, directory sizes and resource availability dynamically change, specially the latter, and continuously splitting/collapsing directories seems to be inefficient too. Finding new splitting patterns or heuristics that adapt to different types of workloads would improve the performance for huge directories.

Finally, along the OSD+ device description, we define a new object type to handle metadata, and the operations to manage that object. We plan to include the specification of directory objects and related operations following the OSD standard [53].

Bibliography

[1] "Batch requests." [Online]. Available: https://developers.facebook.com/docs/reference/ads-api/batch-requests

[2] "Heartbeat: Linux HA." [Online]. Available: http://www.linux-ha.org/Heartbeat

[3] "Making multiple api requests." [Online]. Available: https://developers.facebook.com/docs/graph-api/making-multiple-requests/

[4] "National supercomputer center in Tianjin," 2009. [Online]. Available: http://www.nscc-tj.gov.cn/en/

[5] "Blue gene," 2012. [Online]. Available: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/bluegene/

[6] "Google spreadsheet," November 2013. [Online]. Available: https://developers.google.com/chart/interactive/docs/spreadsheets

[7] "Google base," 2014. [Online]. Available: http://www.google.com/merchants/default

[8] "Google calendar," 2014. [Online]. Available: https://www.google.com/calendar

[9] "Google cloud storage: Sending batch requets," April 2014. [Online]. Available: https://developers.google.com/storage/docs/json_api/v1/how-tos/batch

[10] "Linux 3.14," March 2014. [Online]. Available: http://kernelnewbies.org/Linux_3.14

[11] "Using batch operations," April 2014. [Online]. Available: http://code.google.com/p/gdata-python-client/wiki/UsingBatchOperations

[12] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan., "An OSD-based approach to managing directory operations in parallel file systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2008, pp. 175–184.

[13] ——, "Revisting the metadata architecture of parallel file systems," The Ohio State University, Tech. Rep., 2008.

[14] E. Artiaga and T. Cortes, "Using filesystem virtualization to avoid metadata bottlenecks," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010.

[15] A. Avilés-González, J. Piernas, and P. González-Férez, "A metadata cluster based on OSD+ devices," in *Proceedings of the 23rd SBAC-PAD*, 2011.

[16] ——, "Scalable huge directories through OSD+ devices," in *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom*, 2013, pp. 1–8. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/PDP.2013.11

[17] ——, "Scalable metadata management through OSD+ devices," *International Journal of Parallel Programming*, vol. 42, no. 1, pp. 4–29, February 2014.

[18] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010. [Online]. Available: http://static.usenix.org/event/osdi10/tech/full_papers/Beaver.pdf

[19] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[20] P. J. Braams, "High-performance storage architecture and scalable cluster file system," 2008. [Online]. Available: http://wiki.lustre.org/index.php/Lustre_Publications

[21] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue., "Efficient metadata management in large distributed storage systems," in *Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies (MSST'03)*, 2003.

[22] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161029

[23] A. L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, "Performance and scalability of a replica location service," in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 182–191. [Online]. Available: http://dx.doi.org/10.1109/HPDC.2004.27

[24] P. F. Corbett and F. D. G., "The Vesta parallel file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, pp. 225–264, August 1996. [Online]. Available: http://doi.acm.org/10.1145/233557.233558

[25] S. Dayal, "Characterizing HEC storage systems at rest," Carnegie Mellon University, Tech. Rep. Technical Report CMU-PDL-08-109, July 2008.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of 21st symposium on Operating Systems Principles (SOSP'07)*, October 2007, pp. 205–220.

[27] P. development team, "Parallel virtual file system, version 2," PVFS2, Tech. Rep., September 2003.

[28] A. Devulappali, I. Murugandi, D. Xu, and P. Wyckoff, "Design of an intelligent object-based storage device," Ohio Supercomputer Center, Tech. Rep. Online, 2009. [Online]. Available: http://www.osc.edu/research/networkfile/projects/object/papers/istor-tr.pdf

[29] W. Di, "CMD code walk through," 2009. [Online]. Available: http://wiki.lustre.org/images/7/70/SC09-CMD-Code.pdf

[30] P. C. Dibble, M. L. Scott, and C. S. Ellis, "Bridge: A high-performance file system for parallel processors." in *Proceedings of the 8th international Conference on Distributed Computing Systems (ICDCS'88)*. IEEE Computer Society, 1988, pp. 154–161. [Online]. Available: http://dblp.uni-trier.de/db/conf/icdcs/icdcs88.html#DibbleSE88

[31] A. Dilger, "Lustre metadata scaling," in *IEEE Mass Storage Systems*, April 2012.

[32] M. Dunn and A. L. N. Reddy, "A new I/O scheduler for solid state devices," Department of Electrical and Computer Engineering Texas A&M University, Tech. Rep., April 2009. [Online]. Available: http://dropzone.tamu.edu/TechReports

[33] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing - a fast access method for dynamic files," *ACM Transactions on Database Systems*, vol. 4, pp.

315–344, September 1979.

[34] D. Feng, J. Wang, F. Wang, and P. Xia., "DOIDFH: an effective distributed metadata management scheme," in *Proceeding of the 5th international Conferences on Computational Science and Its Applications (ICCSA'07)*, October 2007.

[35] A. Fikes, "Storage architecture and challenges," in *Google Faculty Summit 2010*, June 2010. [Online]. Available: http://research.google.com/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf

[36] O. Foundation, "Archive auto extraction," 2014. [Online]. Available: http://docs.openstack.org/developer/swift/middleware.html#module-swift.common.middleware.bulk

[37] ——, "Bulk delete," 2014. [Online]. Available: http://docs.openstack.org/api/openstack-object-storage/1.0/content/bulk-delete.html

[38] C. S. Freedman, J. Burger, and D. J. DeWitt, "Spiffi-a scalable parallel file system for the intel paragon," *IEEE Transactions Parallel Distributed Systems*, vol. 7, no. 11, pp. 1185–1200, November 1996. [Online]. Available: http://dx.doi.org/10.1109/71.544358

[39] R. Freitas, J. Slember, W. Sawdon, and L. Chiu, "GPFS scans 10 billion files in 43 minutes," IBM Almaden Research Center, Tech. Rep. RJ10484, 2011. [Online]. Available: http://www.almaden.ibm.com/storagesystems/resources/GPFS-Violin-white-paper.pdf

[40] G. R. Ganger and M. F. Kaashoek., "Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files," in *Proceedings of USENIX Annual Technical Conference (ATC)*, January 1997, pp. 1–17.

[41] G. A. Gibson, D. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture." in *Proceedings of the international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, D. Bhandarkar and A. Agarwal, Eds. ACM Press, 1998, pp. 92–103.

[42] P. González-Férez, J. Piernas, and T. Cortés, "Evaluating the Effectiveness of REDCAP to Recover the Locality Missed by Today's Linux Systems," 2008.

[43] R. L. Haskin, "Tiger shark - a scalable file system for multimedia," *IBM Journal of Research and Development*, vol. 42, pp. 185–197, 1998.

[44] D. He, X. Zhang, D. Du, and G. Grider, "Coordinating parallel hierarchical storage management in object-base cluster file systems," in *Proceedings of the IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2006.

[45] R. Hedges, K. Fitzgerald, M. Gary, and D. M. Stearman, "Comparison of leading parallel NAS file systems on commodity hardware," in *Petascale Data Storage Workshop (poster)*, November 2010. [Online]. Available: http://www.pdsi-scidac.org/events/PDSW10/resources/posters/parallelNASFSs.pdf

[46] R. Henwood, "Remote directories solution architecture," January 2012. [Online]. Available: https://wiki.hpdd.intel.com/display/PUB/Remote+Directories+Solution+Architecture

[47] Hewlett–Packard, "Fstrace," http://tesla.hpl.hp.com/open-source/fstrace, 2002.

[48] High performance data division, "A new generation of Lustre software expands HPC into the commercial enerprise," White Paper, Intel, 2013.

[49] D. Hildebrand and P. Honeyman., "Exporting storage systems in a scalable manner with pNFS," in *Proceedings of the 22nd IEEE Conference on Massive Storage Systems and Technologies (MSST'05)*, 2005.

[50] R. J. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," in *Proceedings of the 18th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'04)*, April 2004.

[51] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, "Smartstore: A new metadata organization paradigm with semantic-awareness for next-generation file systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654070

[52] H. H. Huang, N. Zhang, W. Wang, G. Das, and A. S. Szalay, "Just-in-time analytics on large file systems," *IEEE Transactions on Computers*, vol. 61, no. 11, pp. 1651–1664, 2012.

[53] INCITS Technical Committee T10, "SCSI object-based storage device commands-3 (OSD–3). Project t10/2128–d. Working draft, revision 02," http://www.t10.org/drafts.htm#OSD_Family, July 2010.

[54] Intel. [Online]. Available: https://wiki.hpdd.intel.com/display/PUB/Lustre+2.4

[55] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk Schedulers for Solid State Drivers," in *Proceedings of the 7th ACM international conference on Embedded software*, 2009.

[56] A. Kumar, M. Cao, J. Santos, and A. Dilger, "Ext4 block and inode allocator improvements," in *Linux Symposium*, July 2008.

[57] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 3247–3259, 1998.

[58] R. Latham, N. Miller, R. Ross, and P. Carns., "A next-generation parallel file system for Linux clusters," *LinuxWorld*, pp. 56–59, January 2004.

[59] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proceeding of the 7st USENIX Conference on File and Storage Technologies (FAST'09)*, USENIX, Ed. USENIX, February 2009, pp. 153–166.

[60] W. Lin, Q. Wei, and B. Veeravalli, "WPAR: A weight-based metadata management strategy for petabyte-scale object storage systems," in *Proceedings of the 4th international Workshop on Storage Network Architecture and Parallel I/Os Workshop (SNAPI'07)*, September 2007.

[61] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04),*, 2004.

[62] A. MacDonald, "Nfsv4," *;login:*, vol. 37, no. 1, Febraury 2012.

[63] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Linux Symposium*, 2007. [Online]. Available: http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf

[64] M. K. McKusick and S. Quinlan, "Case Study GFS: Evolution on Fast-forward," 2009. [Online]. Available: http://queue.acm.org/detail.cfm?id=1594206

[65] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, August 2003.

[66] A. Miranda, S. Effert, Y. Kang, E. L. Miller, A. Brinkmann, and T. Cortes, "Reliable and randomized data distribution strategies for large scale storage systems," in *Proceedings of HiPC Conference*, December 2011, pp. 1–10.

[67] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith., "Andrew: A distributed personal computing enviroment," *Communications ACM*, vol. 29, pp. 184–201, March 1986. [Online]. Available: http://doi.acm.org/10.1145/5666.5671

[68] C. Morrone, B. Loewe, and T. McLarty, "mdtest HPC Benchmark," 2010. [Online]. Available: http://sourceforge.net/projects/mdtest

[69] D. Nagle, D. Serenyi, , and A. Matthews., "The panasas ActiveScale storage cluster -delivering scalable high bandwidth storage," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, November 2004.

[70] B. C. Neuman and T. Ts'o, "Kerberos: An authentication service for computer networks," *IEEE Communications Magazine*, vol. 32, pp. 33–38, September 1994. [Online]. Available: http://gost.isi.edu/publications/kerberos-neuman-tso.html

[71] H. Newman, "HPCS mission partner file I/O scenarios, revision 3," November 2008. [Online]. Available: http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf

[72] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proceeding of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, USENIX, Ed. USENIX, February 2011.

[73] E. Polyakov., "The Elliptics network," 2009. [Online]. Available: http://www.ioremap.net/projects/elliptics

[74] ——, "Parallel optimized host message exchange layered file system," 2009. [Online]. Available: http://www.ioremap.net/projects/pohmelfs

[75] M. Probet, "High performance computing - history of the supercomputer," Lecture notes on HPC, 2013.

[76] K. Ren, S. Patil, and G. Gibson, "A case for scaling HPC metadata performance through de-specialization," November 2012.

[77] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," in *Proceeding of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, USENIX, Ed. USENIX, January 2002, pp. 15–30.

[78] D. Roselli, J. Lorch, and T. Anderson., "A comparison of file system workloads," in *Proceedings of USENIX Annual Technical Conference*, June 2000, pp. 41–54.

[79] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, and E. H. Siegel., "Coda: A highly available file system for a distributed workstation enviroment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[80] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceeding of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, USENIX, Ed. USENIX, January 2002.

[81] Seagate, "Kinetic open storage," 2013. [Online]. Available: https://developers.seagate.com/display/KV/Kinetic+Open+Storage+Documentation+Wiki

[82] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 26th IEEE Conference on Massive Storage Systems and Technologies (MSST'10)*, 2010.

[83] K. V. Shvachko, "Apache Hadoop. The scalability update," *USENIX ;login Magazine*, vol. 36, pp. 7–13, June 2011.

[84] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger, "A transparently-scalable metadata service for the Ursa Minor storage system," in *Proceedings of USENIX Annual Technical Conference (ATC)*, June 2010.

[85] ——, "A transparently-scalable metadata service for the Ursa Minor storage system," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2010.

[86] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Transactions on Software Engineering*, vol. 9, pp. 219–228, May 1983. [Online]. Available: http://dx.doi.org/10.1109/TSE.1983.236608

[87] Sun-Oracle, "Lustre tunning," 2010. [Online]. Available: http://wiki.lustre.org/manual/LustreManual18_HTML/LustreTuning.html

[88] SwiftStack, "Kinetic motion with seagate and openstack swift," October 2013. [Online]. Available: https://swiftstack.com/blog/2013/10/22/kinetic-for-openstack-swift-with-seagate/

[89] I. systems and technology group, "An introduction to GPFS version 3.5," IBM Almaden Research Center, Tech. Rep., August 2012. [Online]. Available: http://www.almaden.ibm.com/storagesystems/resources/GPFS-Violin-white-paper.pdf

[90] S. Tweedie, "Journaling the Linux ext2fs Filesystem," in *LinuxExpo'98*, 1998.

[91] University Corporation for Atmospheric Research, "metarates," 2004. [Online]. Available: http://www.cisl.ucar.edu/css/software/metarates/

[92] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty, "File system workload analysis for large scale scientific computing applications," in *Proceedings of the 21st IEEE Conference on Massive Storage Systems and Technologies (MSST'04)*, 2004.

[93] J. Wang, D. Feng, F. Wang, , and C. Lu., "MHS: A distributed metadata management strategy," *The Journal of Systems and Software*, vol. 82, no. 12, pp. 2004–2011, July 2009.

[94] L. Weijia, X. Wei, J. Shu, and W. Zheng., "Dynamic hashing: Adaptive metadata management for petabyte-scale file systems," in *Proceedings of the 23rd IEEE Conference on Massive Storage Systems and Technologies (MSST'06)*, May 2006, pp. 159–164.

[95] S. Weil, "Scalable archival data and metadata management in object-based file systems," Storage Systems Research Center, Tech. Rep., June 2004.

[96] S. Weil., "Ceph: reliable, scalable, and high-performance distributed storage," Ph.D. dissertation, University of California, Santa Cruz, (CA), December 2007.

[97] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn., "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[98] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn., "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, November 2006.

[99] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn., "Rados: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2nd Parallel Data Storage Workshop*, November 2007.

[100] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller., "Dynamic matadata management for petabyte-scale file systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, November 2004.

[101] B. Welch, M. Unangst, Z. Abbasi, D. Gibson, J. Mueller, B. Small, J. Zelenka, and B. Zhou., "Scalable performance of the Panasas Parallel File System," in *Proceeding of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, USENIX, Ed. USENIX, February 2008.

[102] Whamcloud, "Lustre file system 2.x," November 2012. [Online]. Available: http://build.whamcloud.com/job/lustre-manual/lastSuccessfulBuild/artifact/lustre_manual.xhtml#idp613648

[103] R. Wheeler, "One billion files: Scalability limits in Linux file systems," in *LinuxCon'10*, August 2010. [Online]. Available: http://events.linuxfoundation.org/slides/2010/linuxcon2010_wheeler.pdf

[104] Y. Wu, "A study for scalable directory in parallel file systems," Master's thesis, Clemson University, Clemson, SC, USA, July 2009.

[105] S. Yang, W. B. L. III, and E. C. Quarles, "Scalable distributed directory implementation on Orange File System," in *Proceedings of 7th international Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'11)*, 2011.

[106] Y. Zhang, Z. Q. Qian, and W. M. Zheng, "Employing intelligence in object-based storage devices to provide attribute-based file access," in *Science China Information Sciences*, vol. 56, no. 1, March 2013, pp. 1–10.

[107] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical Bloom Filter Arrays (HBA): A novel, scalable metadata management system for large cluster-based storage," in *Proceedings of IEEE International Conference on Cluster Computing*, September 2004.