MemcachedFS: sistema de ficheros para clusters basado en memoria distribuida

Francisco J. Rodrigo[1] , Javier García Blas[2] y Jesús Carretero[3]

Resumen—
Este trabajo presenta el diseño e implementación de un sistema de ficheros distribuido de alto rendimiento para clusters y clouds basado en memoria distribuida, llamadado MemcachedFS. Nuestra solución ofrece una capa de almacenamiento temporal para las aplicaciones basado en Memcached. Se ha evaluado el rendimiento de MemcachedFS comparando con otros sistemas distribuidos como NFS y PVFS. Los resultados arrojan que MemcachedFS obtiene el mejor rendimiento cuando las aplicaciones tienen accesos pequeños.

Palabras clave— **memoria distribuida, sistema de ficheros, clusters, cloud computing.**

I. Introducción

Actualmente, los dispositivos de almacenamiento son el principal talón de Aquiles en los sistemas de alto rendimiento que requieren la utilización de dispositivos de memoria secundaria. Los actuales avances se centran en complejas configuraciones multidisco o los dispositivos de estado sólido (SSD). Sin embargo, a la hora de distribuir los datos a través de una red, estos sistemas pueden resultar complejos o no cumplir las expectativas de rendimiento y flexibilidad.

Está ampliamente aceptado que los futuros sistemas a gran escala de gestión de almacenamiento tienen que ser jerárquicos [1]. Esta organización jerárquica tendrá que ser estructurada como sistemas autónomos, con el fin de reducir el impacto de los fallos en el comportamiento global del sistema. Los sistemas deben ofrecer la capacidad instantánea de ampliación y reducción de escala con el objetivo de responder con rapidez a los cambios en la carga, la disponibilidad y la eficiencia de los recursos (incluida la eficiencia energética). Los avances recientes en cloud computing y las tecnologías de virtualización ofrecen una gran oportunidad, pero su investigación y la explotación se encuentra todavía en la fase inicial de datos de aplicaciones de uso intensivo de colaboración y de almacenamiento de alto rendimiento.

Una de las soluciones más populares para ofrecer servicios escalables y eficientes de memoria distribuida es Memcached [2], [3]. El principal objetivo de Memcached es proporcionar un sistema de almacenamiento de caché para bases de datos y es utilizado actualmente en multitud de infraestructuras, algunas tan conocidas como YouTube, Facebook o Twitter [4]. Sin embargo, Memcached no se reduce a su utilización como caché de consultas a bases de datos o contenido dinámico, ya que, gracias a su diseño genérico, puede ser utilizado para almacenar cualquier tipo de datos en su caché distribuida, de modo que constantemente se buscan nuevos entornos en los que aplicar sus beneficios.

En este trabajo presentamos un sistema de almacenamiento distribuido basado en Memcached llamadado MemcachedFS. Este trabajo tiene las siguientes contribuciones. Primero detallará el diseño e implementación de MemcachedFS como sistema de ficheros basado en memoria distribuida. Se aborda también la implementación de una librería de entrada y salida estándar. Su diseño se asemejará lo máximo posible a la utilizada en POSIX, de modo que sea sencillo portar los programas ya existentes a la utilización de MemcachedFS. Por último, se presenta una evaluación de rendimiento y un estudio de viabilidad de la utilización de Memcached como sistema de ficheros distribuido.

El resto de este documento sigue la siguiente estructura. La Sección II presenta algunos trabajos relacionados y alternativas a Memcached. Las características de Memcached son presentadas en la Sección III. Describimos los detalles de diseño de MemcachedFS en la Sección IV. En la Sección V mostramos resultados experimentales obtenidos de la evaluación del sistema enfrentado a otros sistemas de ficheros distribuidos o paralelos. Finalmente, presentamos las conclusiones y futuras líneas de trabajo en la Sección VI.

II. Trabajos relacionados

Los sistemas de almacenamiento basados en cachés asíncronas, especialmente aquellos basados en las políticas de almacenamiento *write-back* y con prefetch han mostrado resultados prometedores ocultando la latencia de almacenamiento [5], [6]. En el estado del arte de los sistemas de almacenamiento a gran escala destaca Hadoop [7]. Hadoop distribuye al azar varias copias de bloques de datos. Este enfoque apunta a aumentar el rendimiento y disponibilidad, pero impide la creación de algoritmos de distribución

[1]Dept. de Informática, Univ. Carlos III de Madrid, e-mail: frodrigo@arcos.inf.uc3m.es
[2]Dept. de Informática, Univ. Carlos III de Madrid, e-mail: fjblas@arcos.inf.uc3m.es
[3]Dept. de Informática, Univ. Carlos III de Madrid, e-mail: jcarrete@arcos.inf.uc3m.es

personalizada de los datos para mejorar la localidad.

Actualmente existen multitud de alternativas que ofrecen similitudes con las características de Memcached. Un ejemplo de ello son los aceleradores/cacheadores de PHP como APC, eAccelerator o XCache [8], etc. Sin embargo, estos sistemas, como otras alternativas del estilo de *nmap memory*, solo son capaces de aprovechar la memoria inutilizada en el servidor en que son ejecutados, es decir, se pierde la principal ventaja de transparencia en la ubicación de los datos cacheados.

Por otro lado, los propios sistemas gestores de bases de datos constan de sistemas de caché de consultas, como "MySQL's query cache", este sistema de cacheo es completamente centralizado y utiliza sólo la RAM inutilizada en la base de datos. Además, en ningún momento se puede considerar como alternativa de cara al diseño de un sistema de ficheros distribuido, ya que, su funcionamiento está diseñado única y exclusivamente para servir de caché a las consultas de una base de datos. Otras de las alternativas más conocidas que pretenden hacerle frente, son bases de datos NoSQL. Algunos ejemplos son Cassandra [9], MongoDB o, incluso, Membase que se trata de un proyecto de los creadores de Memcached utilizando éste como base y dotándolo de persistencia y facilidades a la hora de expandir o contraer el número de nodos. Se ha optado por desarrollar sobre Memcached por ser mucho más utilizado y, en caso de ser necesario, se podría desplegar sobre Membase con sus ventajas.

Redis se presenta como otra alternativa similar a Memcached, pero aportando algunas funcionalidades extra como son la persistencia y replicación de los datos. Sin embargo, hay tres principales desventajas en esta solución. Escasa popularidad actual, rendimiento ligeramente peor a Memcached [10] y, sobre todo, falta de transparencia en la distribución de los datos que repercute directamente en la escalabilidad. Se trata, por tanto, de un sistema de tabla hash en memoria en red y no de un sistema de caché distribuido.

Otro trabajo que utiliza Memcached dotándolo de una funcionalidad para la que no fue concebido consiste en una implementación de Memcached como caché de un sistema de ficheros distribuido [11], que demuestra que puede ser utilizado para mejorar el rendimiento, sin embargo, se utiliza solo como caché para ficheros menores de 1 Mbyte, sin llegar a implementar un sistema de ficheros distribuido completo como en nuestro caso.

III. Memcached

Memcached es una solución que proporciona un sistema de memoria compartida y distribuida. Como se puede observar en la Figura 1 consiste en ofrecer memoria RAM disponible en varios servidores distintos como si de un único espacio de caché se tratara, mediante la implementación de una tabla hash distribuida, es decir, Memcached almacena objetos binarios de hasta 1 Mbyte y los asocia a una clave de hasta 250 bytes por la que podrán ser buscados.

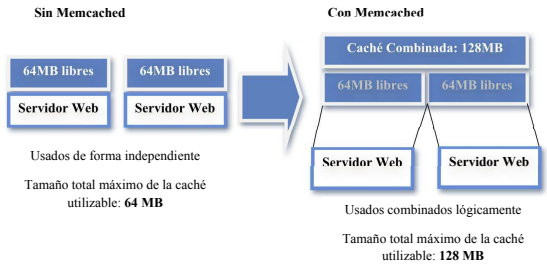

Fig. 1. Ejemplo esquemático de utilización de Memcached.

Este sistema de caché distribuida fue diseñado teniendo en mente la optimización de los accesos a bases de datos de páginas web dinámicas bajo grandes cargas de trabajo, por tanto, está diseñado para ofrecer datos en lectura de una forma mucho más rápida y ligera que el acceso a bases de datos, sin embargo, los datos deben ser modificados primero en la base de datos y después en caché. No existe ningún mecanismo de escritura retardada o diferida para devolver a memoria secundaria los datos expulsados de caché, y la caché sólo se mantiene en memoria, por lo que en caso de caerse cualquiera de los nodos o de que se expulse alguno de los valores por caducidad o por su política de reemplazo LRU, se perderá la información que contenían.

Los datos no están replicados y Memcached no tiene ningún mecanismo para administrar fallos, a diferencia de algunos de sus clientes, que sí administran la replicación/persistencia. Actualmente, existen algoritmos consistentes de hash, en los que cambiando alguno de los servidores de la lista, el resto de referencias siguen apuntando al mismo servidor previo a la modificación, como libketama [12].

IV. Diseño de MemcachedFS

La principal funcionalidad que ofrece el sistema desarrollado es la de sistema de ficheros distribuido. Para ello se utiliza la funcionalidad ya ofrecida por Memcached, en la que se parte de una tabla *hash* de pares clave-valor distribuida entre varios servidores de forma totalmente transparente al usuario. Gracias a ello, se puede usar una entrada de este mapa hash como si de un bloque de datos de un dispositivo convencional se tratara, con un tamaño máximo de 1 MByte. En dichos bloques de datos, se guardará cualquier tipo de información, ya sean metadatos o datos.

Como se puede ver en la Figura 2, la arquitectura seguida en el diseño de MemcachedFS es por capas, siendo la más baja la de interacción con el usuario (*memcachedfs*) y la intermedia (*utils*) la de

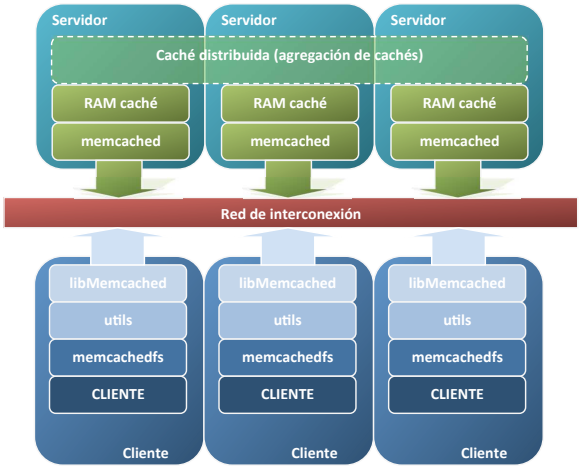

Fig. 2. Arquitectura por capas de MemcachedFS.

interacción con libMemcached que es la librería de C utilizada para la comunicación con Memcached: para hacer las conexiones a los servidores, introducir datos en la caché ("set"), obtener datos de la caché ("get"), etc. Se han aprovechado, siempre que ha sido posible, las características que ofrece a la hora de hacer extracciones simultáneas de varios valores de caché ("mget"), ya que, mejoran el rendimiento con respecto a las extracciones individuales. Mediante la utilización de esta arquitectura es posible cambiar Memcached por cualquier otro sistema de caché distribuida con tan solo cambiar la capa de interconexión con libMemcached. Para asemejarse al estándar de entrada y salida de POSIX, se han implementado sus funciones básicas con una sintaxis que mimetiza al máximo el estándar.

A. Gestor de almacenamiento

Lo único necesario para almacenar un bloque de datos o de metadatos en la caché distribuida, es asignarle un identificador como clave. Los identificadores que se utilicen como claves, deben ser fácilmente recalculables, ya que, serán necesarios para recuperar los bloques de datos/metadatos de la memoria caché distribuida en el futuro. El esquema de identificadores seguido es el siguiente:

- **Bloques de metadatos**. Para los bloques de metadatos, el identificador (clave) es el nombre que se le dé al fichero y podrá tener una longitud de hasta 249. De esta forma, cada vez que se abra el fichero, sólo necesitaremos su nombre para acceder tanto a sus metadatos, como a su contenido (a partir del identificador almacenado en sus metadatos como se verá más adelante). No pueden existir en el sistema dos ficheros con el mismo nombre, ya que no se ha implementado ningún sistema de directorios. Se descartó la implementación de un sistema de directorios, debido a que al tratarse de una librería experimental, suponía un aumento de la complejidad para implementar una funcionalidad muy concreta, sin aportar nada de cara a la evaluación de rendimiento.
- **Bloques de datos**. De cara a la generación de identificadores para los bloques de datos de un fichero, el esquema a seguir se basa en concatenar el identificador único del fichero con el índice del bloque de datos.

Para asegurar que la longitud del identificador único sea siempre la misma y para asegurar que el bloque de datos pertenece exactamente al fichero con el que se está trabajando se aplica SHA2-512 sobre la concatenación del nombre del fichero más el instante de creación del fichero. La elección de SHA-512 se basa en el tamaño de los hash que genera, debido a que se precisa de un mecanismo resistente a colisiones a la vez que consuma menos de 250 bytes (para poder insertar a continuación del identificador el número de bloque). En este caso, SHA-512 consume 128 bytes, algo más de la mitad de la longitud de la clave, dejando suficiente espacio (hasta 125 caracteres para direccionar bloques de datos) siendo además el más resistente a colisiones de todas las versiones de SHA2, gracias a su mayor longitud de resumen.

Para evitar tener que recalcular el identificador en cada acceso, el identificador es calculado en el momento de la creación del fichero y se guarda en su estructura de metadatos. De este modo, cuando se quiera acceder a uno de los bloques de un fichero, tan sólo habrá que dirigirse al bloque de metadatos de dicho fichero, recuperar su identificador y generar la clave del bloque a acceder.

Dado que la clave a la que están asociados cada uno de los bloques de datos no viene asignada por el dispositivo (como suele ocurrir en los dispositivos habituales de almacenamiento), podemos prescindir por completo de almacenar en los metadatos del fichero los punteros a los bloques que éste ocupa. Esto nos aporta dos beneficios: se ahorra espacio de almacenamiento, ya que no es necesario guardar el puntero a los datos (este beneficio se ve mitigado en parte por tener que almacenar siempre clave+valor) y, sobre todo, mejora el rendimiento. Mientras que en los sistemas de ficheros habituales, hay que recorrer el árbol de punteros hasta conseguir llegar al puntero que se busca (que dependerá de la posición que ocupe el bloque en el fichero) con las consiguientes búsquedas en muchos casos no secuenciales, y la pérdida de rendimiento sobre todo en discos mecánicos; en el caso del sistema de ficheros desarrollado, el puntero al bloque de datos (clave) se puede calcular rápidamente y tan sólo se requiere un acceso al bloque de metadatos. Además de ser más rápido, este tiempo es siempre constante. Por último, se evita por com-

pleto la fragmentación tanto interna como externa, ya que el direccionamiento será siempre a bloques de datos del tamaño justo, no quedando espacio vacío al final de los bloques ni entre ficheros.

El uso de este identificador único en lugar del nombre de fichero, nos permite evitar problemas con bloques obsoletos (el fichero se llamará igual, pero su identificador habrá cambiando y con él todos sus bloques de datos asociados) y facilita el borrado y renombrado de ficheros, ya que, se podrá hacer borrado lógico de los ficheros con solo borrar el bloque de metadatos y para cambiar el nombre del fichero bastará con cambiar la clave de dicho bloque de metadatos, siendo válidos aún todos los bloques de datos asociados a su identificador único.

Los metadatos que se guardarán del fichero están basados en la estructura "stat" de POSIX, convenientemente modificada para adaptarse a las características del sistema de ficheros distribuido.

B. Descripctores de fichero

Un descriptor de fichero es un identificador (normalmente un número entero) que se devuelve al realizar una operación de apertura y que identifica de forma unívoca una de las entradas de la tabla de descriptores. En dicha entrada se guardará información sobre el fichero que está abierto y su estado, de forma que cada vez que se realice una operación con ese fichero (lectura, escritura, posicionamiento del puntero, cerrado), sólo haya que aportar el descriptor de fichero que identifica esa entrada de la tabla. Se ha decidido guardar la tabla de descriptores de fichero como variable global de la librería, es decir, que cada cliente tenga en memoria principal su propia tabla de descriptores de fichero. Se podría haber guardado en caché distribuida, pero no aportaría ninguna ventaja sustancial, ya que, no está soportado ningún tipo de sincronización para que dos clientes accedan al mismo fichero de forma simultánea. Se puede hacer sólo administrando la sincronización y la coherencia de lecturas/escrituras por parte del cliente sin ningún apoyo de la librería. Sin embargo, tener la tabla de descriptores en memoria principal aporta una mejora de rendimiento sustancial, ya que, todas las acciones que se realizan sobre la tabla son mucho más rápidas que si hubiese que acceder a ella por red (caché distribuida). Además, la tabla ha sido implementada como un array de estructuras, de modo que no es necesario guardar su identificador, el descriptor de fichero será el índice del array en el que está almacenada la estructura.

C. Configuración flexible

La implementación actual permite desplegar MemcachedFS sobre tantos servidores como soporte Memcached, permitiendo configurar el tamaño de bloque (desde 1 Kbyte hasta 1023z Kbytes) y el modo en que se realizan las lecturas, simples o múltiples. Gracias a la primitiva *mget* se permite solicitar varios bloques de una sola vez a los servidores , con el objetivo de agregar peticiones pequeñas en una sola.

Se han diseñado dos modos adicionales de funcionamiento en función de la gestión de la caché: los modos caché y pre-caché. En el modo pre-caché, se puede abrir un fichero local del sistema, que será transferido al sistema de ficheros distribuido y se trabajará con él mediante MemcachedFS. En el momento del cierre, el fichero será transferido de nuevo al sistema de ficheros local. El modo caché es una optimización del anterior, en el que se abre un fichero local pero solo se llevan a caché distribuida aquellos bloques de datos que se utilizan. En el momento del cierre solo se sincronizarán aquellos bloques que hayan sido modificados desde la apertura (los que hayan sido simplemente leídos, estarán en caché distribuida pero no serán sincronizados, puesto que su contenido es el mismo que en el sistema de ficheros local).

V. Evaluación

Para las pruebas se ha utilizado un sistema compuesto por 24 nodos de cómputo con las siguientes características: Intel Xeon CPU E5405 @2.00GHz, memoria 4GB DDR3, tarjeta de red Gigabit Ethernet 100/1000 y el sistema operativo Ubuntu 10.10 Server Maverick.

Se ha evaluado el rendimiento de MemcachedFS, y ha sido comparado con los sistemas de ficheros distribuidos NFSv4 [13] y PVFS2 [14]. NFSv4 está configurado con 512 Kbyte de tamaño de bloque sobre ext4 con tamaño de bloque de 4 Kbyte, mientras que se ha utilizado PVFS2 en su versión 2.8.2, con tamaño de bloque de 64 Kbyte y cuatro servidores de E/S.

Para llevar a cabo esta evaluación de rendimiento, se han diseñado e implementado una serie de procedimientos, utilizando tanto la librería de E/S estándar como la de MemcachedFS.

A. Rendimiento

Esta evaluación consiste en abrir, operar (leer o escribir) y cerrar un fichero, aumentando el tamaño de la operación de forma progresiva. De este modo, no solo se mide el rendimiento puro del dispositivo, sino también la eficiencia del sistema de ficheros administrando los metadatos. MemcachedFS se ha desplegado sobre 4 servidores (aunque se han obtenido resultados similares variando el número de servidores) y se han utilizado tamaños de bloque de 8, 128 y 1023 Kbyte.

Lo primero que se observa es que, tanto MemcachedFS como PVFS2 mejoran su rendimiento a medida que aumenta el tamaño de la escritura, llegan-

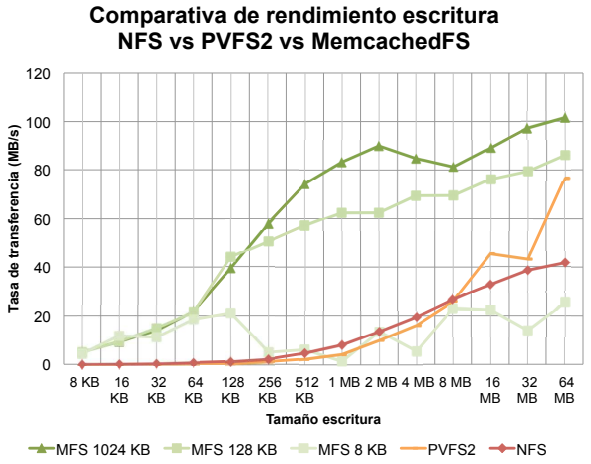

Fig. 3. Evaluación de rendimiento de MemcachedFS frente a NFS y PVFS en operaciones de escritura.

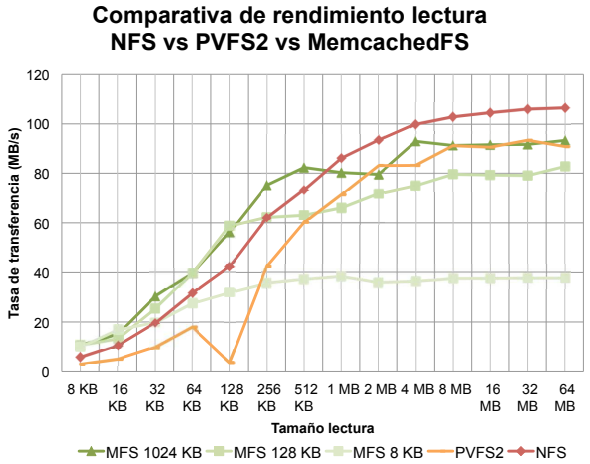

Fig. 4. Evaluación de rendimiento de MemcachedFS frente a NFS y PVFS en operaciones de lectura.

do ambos a rozar el límite impuesto por la red. Sin embargo, el caso más similar a la configuración utilizada de PVFS2, el de MemcachedFS con 128 Kbyte de tamaño de bloque y cuatro servidores, es sustancialmente más rápido con la mayoría de tamaños de fichero. Mientras que el rendimiento de PVFS empieza a despegar a partir de ficheros de 64 Kbyte (su tamaño de bloque) y el crecimiento de rendimiento es completamente exponencial (al igual que lo es el aumento del tamaño de fichero), MemcachedFS alcanza el 50 % de su rendimiento con tamaños de fichero inferiores al Mbyte y se mantiene siempre por encima de PVFS2 en el resto de casos.

En este primer caso, referido a las escrituras, se observa en la Figura 3 un rendimiento bastante discreto de NFSv4. Probablemente se deba en parte al rendimiento real y en parte a la metodología empleada para medir el rendimiento. Después de cada escritura, se cerraba el fichero y se utilizaba la función sync() para hacer permanentes los cambios realizados en los ficheros, llevándolos a disco y, con ello, perdiendo parte del rendimiento a fin de evitar interferencias producidas por el cacheo de datos. Como se observa, el rendimiento a partir de ficheros de 1 Mbyte es muy similar al ofrecido por MemcachedFS con tamaño de bloque 8 Kbyte. Para tamaños de bloque superiores es muy superior.

En el caso de las lecturas, como se puede ver en la Figura 4, el modelo que sigue el rendimiento de PVFS2 es muy parecido al que sigue MemcachedFS con tamaño de bloque de 1023 Kbyte y de nuevo se roza la capacidad de la red en los mejores casos. Comparado con MemcachedFS con tamaños de bloque de 128 Kbyte, el mejor caso dependerá en parte del uso que se quiera dar al sistema. En ficheros de menos de 1 Mbyte, MemcachedFS se comporta ligeramente mejor, mientras que a partir de dicho tamaño, es PVFS2 el que coge la delantera. En cualquier caso,

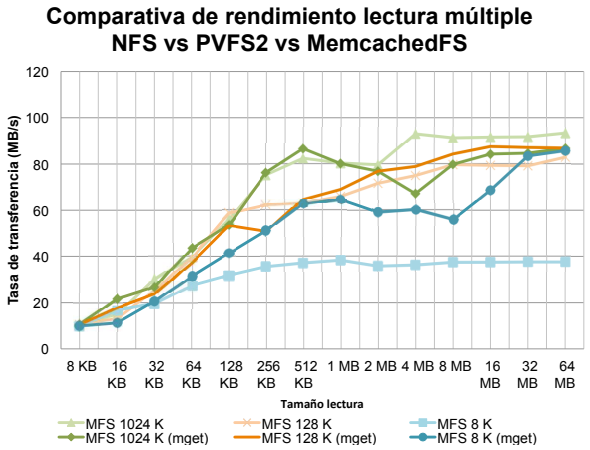

Fig. 5. Evaluación de rendimiento de MemcachedFS frente a NFS y PVFS en operaciones de lectura implementadas como lecturas múltiples (*mget*).

MemcachedFS consigue resultados bastante competitivos con tamaños de bloque grandes, mientras que su rendimiento es un poco más limitado con tamaños de bloque pequeños, algo lógico debido al aprovechamiento de la red. En este caso el dominio de NFS es bastante evidente gracias a su gran tamaño de bloque, aunque con los tamaños de bloque mayores de MemcachedFS, se consigue un resultado bastante competitivo, superando incluso a NFS en lecturas inferiores al Mbyte. La ventaja principal que ofrece NFS con respecto a MemcachedFS es que tiene un rendimiento mucho más estable. La gráfica que forma es casi perfecta, con muy pocas irregularidades.

B. *Accesos múltiples*

En la Figura 5 se compara el rendimiento de la versión de MemcachedFS anterior con una implementada aprovechando las lecturas múltiples, es decir, mediante el comando *mget* obtiene varios bloques en una sola petición. La nueva implementación empeora ligeramente su rendimiento en los mejores casos

a cambio de obtener unos valores mucho más estables con tamaños de bloque pequeños, permitiendo de este modo un rendimiento aceptable y competitivo incluso con tamaños de bloque muy pequeños para un sistema de ficheros distribuido, por lo que aporta flexibilidad en función de las necesidades.

VI. Conclusiones y trabajos futuros

Se ha presentado un diseño e implementación de un sistema de ficheros distribuido que utiliza Memcached para ofrecer un mecanismo de caché escalable. Se ha evaluado la implementación realizada comparándola con otros sistemas de ficheros distribuidos o paralelos, demostrando un rendimiento comparable en el peor de los casos a los sistemas de ficheros existentes.

Se ha conseguido un sistema ampliamente escalable gracias a Memcached y portable gracias a la utilización de una sintaxis muy similar a la de POSIX, además de permitir flexibilidad gracias al diseño por capas, es decir, se podrá cambiar de Memcached a otra plataforma similar tan solo modificando el código de una de las capas.

Con respecto a los trabajos futuros, en primer lugar se debe realizar una mejora de la persistencia mediante mecanismos *write-back* para que, de forma periódica, los datos almacenados en Memcached sean trasladados a un dispositivo persistente para evitar que se pierdan si son expulsados de la caché.

Como funcionalidad adicional se debe considerar la integración del funcionamiento del sistema de ficheros distribuido con el de un sistema de ficheros local, es decir, poder acceder a ficheros remotos y ficheros locales de forma completamente transparente, sin tener que hacer diferencias entre ellos. De cara a la mejora de la portabilidad, podría ser interesante adaptar el código a la interfaz FUSE, para poder reemplazar de forma sencilla a cualquier otro sistema de ficheros que implemente dicha interfaz. También se puede plantear como trabajo futuro, la utilización de MemcachedFS como caché de un sistema de ficheros distribuido.

Mediante la evaluación de rendimiento de lecturas y escrituras aleatorias, se ha observado que puede ser algo ineficiente actualizar el bloque de metadatos absolutamente en todas las operaciones. Como posible mejora de rendimiento, se debería mantener el bloque de metadatos en memoria del cliente desde la apertura hasta el cierre de un fichero y sincronizar el bloque de metadatos de forma retardada. Por último, se debería investigar en la utilización de los ficheros de forma concurrente, de cara a conseguir un sistema de ficheros distribuido más versátil y en la paralelización del sistema de ficheros.

Agradecimientos

Este trabajo ha sido parcialmente subvencionado por el proyecto "Técnicas Escalables de Entrada /Salida en Entornos Distribuidos y de Computación de Altas Prestaciones" del Ministerio de ciencia e innovación, TIN2010-16497.

Referencias

[1] Jack Dongarra, Pete Beckman, Terry Moore, and Aerts, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[2] Brad Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, aug 2004.

[3] Alan Harris and Alan Harris, "Distributed caching via memcached," in *Pro ASP.NET 4 CMS*, pp. 165–196. Apress, 2010.

[4] T. Finger and C. Maciel, "A Comparative Study On The Architecture Of Social Networks Facing Users Great Demand," *IEEE (Revista IEEE America Latina) Latin America Transactions*, vol. 10, no. 1, pp. 1208 –1214, jan. 2012.

[5] Florin Isaila, Javier Garcia Blas, Jesus Carretero, Robert Latham, Samuel Lang, and Robert Ross, "Latency hiding file i/o for blue gene systems," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2009, CCGRID '09, pp. 212–219, IEEE Computer Society.

[6] Zhe Zhang, Amit Kulkarni, Xiaosong Ma, and Yuanyuan Zhou, "Memory resource allocation for file system prefetching: from a supply chain management perspective," in *Proceedings of the 4th ACM European conference on Computer systems*, New York, NY, USA, 2009, EuroSys '09, pp. 75–88, ACM.

[7] Richard A. Brown, "Hadoop at home: large-scale computing at a small college," in *Proceedings of the 40th ACM technical symposium on Computer science education*, New York, NY, USA, 2009, SIGCSE '09, pp. 106–110, ACM.

[8] Li Chen, Elke A. Rundensteiner, and Song Wang, "Xcache: a semantic caching system for xml queries," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2002, SIGMOD '02, pp. 618–618, ACM.

[9] Avinash Lakshman and Prashant Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[10] Yandong Mao, Eddie Kohler, and Robert Tappan Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM european conference on Computer Systems*, New York, NY, USA, 2012, EuroSys '12, pp. 183–196, ACM.

[11] Chuncong Xu, Xiaomeng Huang, Nuo Wu, Pengzhi Xu, and Guangwen Yang, "Using memcached to promote read throughput in massive small-file storage system," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, nov. 2010, pp. 24 –29.

[12] "libketama Official Web Page," 2007.

[13] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Nfs version 4 protocol," 2000.

[14] Avery Ching, Alok Choudhary, Wei Keng Liao, Rob Ross, and William Gropp, "Noncontiguous I/O through PVFS," in *CLUSTER '02*, Washington, DC, USA, 2002, p. 405, IEEE Computer Society.