Escuela Politécnica Superior

Proyecto de Fin de Carrera

Diseño e implementación de un sistema de ficheros distribuido basado en Memcached

Autor: Francisco José Rodrigo Duro

Tutor: Francisco Javier García Blas

Leganés, Octubre de 2011

Autor: Francisco José Rodrigo Duro
Director: Francisco Javier García Blas

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 13 de octubre de 2011 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

AGRADECIMIENTOS

En primer lugar, mi más sincero agradecimiento va para mis padres. A pesar de sus orígenes humildes, su esfuerzo y sacrificio durante toda su vida, pero especialmente, a lo largo de estos últimos años son los que me han brindado la oportunidad de realizar unos Estudios Superiores. Sé que su orgullo en estos momentos es incluso superior al mío.

A mi hermana, que desde pequeño ha estado ahí para acompañarme y ayudarme a subsanar mis errores. Sé que si hubiese estado en sus manos, me hubiese ayudado incluso más a lo largo de la carrera.

A mis tíos y a mis abuelos, especialmente a aquellos que se quedaron en el camino, porque sé que una de las principales motivaciones que guiaban su vida era verme algún día graduado. Ojalá pudiera haberos dado esa alegría. Este año ha sido muy duro para toda la familia, esperemos que este sea el punto de inflexión que cambie el rumbo.

A mis amigos, por no desfallecer a pesar de las épocas duras en las que no había tiempo para vernos, siempre han seguido ahí, dispuestos a hacerme olvidar las preocupaciones.

Por supuesto, a mis compañeros, es imposible llegar a este momento sin unos buenos compañeros en los que apoyarse y con los que repartir el trabajo. Especialmente a Nieto y a Óscar, porque han sido muchas horas juntos en situaciones agotadoras, que lo hubiesen sido aún más sin su esfuerzo y su compañía.

Por último, no por ser menos merecedor de este agradecimiento, sino por ser el último eslabón agregado a la cadena, merece mi reconocimiento mi tutor Javi, por el interés mostrado más allá de los límites del Proyecto, incluso cuando estaba a cientos de kilómetros de distancia.

Es imposible aglutinar en unas decenas de palabras toda la gente que ha influido de manera positiva en un proceso que ha durado los últimos seis años, pero no me olvido de ninguno, todo aquel que ha mostrado su apoyo, su ayuda o simplemente su interés, sabe que es parte de este logro y que merece mi agradecimiento.

RESUMEN

En la actualidad, 18 de los 20 sitios web más importantes en lo que a número de usuarios se refiere, utilizan *Memcached* para mejorar su rendimiento y escalabilidad (por ejemplo, Google, YouTube, Facebook, Twitter, etc.). En este Proyecto de Fin de Carrera se hará una aproximación para estudiar la viabilidad de utilizar *Memcached* como base para el diseño e implementación de un sistema de ficheros distribuido de alto rendimiento para entornos clúster.

En este proyecto se pretende obtener una interfaz para trabajar con *Memcached* como si de un sistema de ficheros distribuido se tratase, permitiendo incluso, utilizar *Memcached* como caché para un sistema de ficheros local, siempre procurando que la interfaz diseñada e implementada sea lo más similar posible a la ofrecida por POSIX para las operaciones de entrada y salida estándar (*open*, *read*, *write*, *close*, etc.)

Además en este proyecto se presenta la evaluación de rendimiento de todas las funcionalidades implementadas, de modo que se pueda juzgar la viabilidad y utilidad de un sistema de ficheros distribuido basado en *Memcached*.

Palabras clave: memcached, sistema de ficheros distribuido, distribuido, caché distribuida, rendimiento

ABSTRACT

Nowadays, 18 of the top 20 sites in terms of user counts are powered by *Memcached* to improve their performance and scalability (e. g. Google, YouTube, Facebook, Twitter, etc.). In this end-of-degree dissertation an approximation to the study of the viability of using *Memcached* as the base of the design and implementation of a distributed file system will be done.

This work aims to obtain an interface (an API) for working with *Memcached* as a distributed file system and utilizing *Memcached* as a local file system's cache, always trying that the API designed and implemented is as similar as possible to the one used in POSIX for standard I/O operations (*open*, *read*, *write*, *close*, etc.)

We have evaluated the performance of each one of the implemented functionalities and with the data obtained, evaluated the viability and utility of a distributed file system based in *Memcached*.

Keywords: memcached, distributed file system, distributed cache, performance

ÍNDICE

Contenido

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

Este capítulo pretende dar una visión global tanto del proyecto, explicando los motivos que incitaron su realización y los objetivos que se persigue conseguir, así como del propio documento, detallando su estructura y otras herramientas para facilitar su lectura.

1.1 Motivación

Memcached se trata de una plataforma que está creciendo muy rápido actualmente. El objetivo principal de *Memcached* es proporcionar un sistema de almacenamiento de caché para bases de datos y es utilizado en multitud de páginas web, algunas tan conocidas como YouTube[1], Facebook[2], Twitter[3] o Tuenti[4]. De hecho, es utilizado por 18 de los 20 sitios web con más usuarios[12] actualmente. Pero las características de *Memcached* no se reducen a su utilización como caché de consultas a bases de datos o de páginas web dinámicas, ya que, su diseño es genérico y permite ser utilizado para almacenar los datos que se deseen en su caché distribuida, de modo que constantemente se buscan nuevos entornos en los que aplicar sus beneficios.

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. Se está investigando en este sentido de forma muy abundante y se está viviendo una pequeña revolución con la inclusión cada más frecuente de complejas configuraciones multi-disco o 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. Es por ello, que es interesante estudiar si *Memcached* puede ser capaz de, al igual que aumenta el rendimiento de sitios web, adaptar su sistema de almacenamiento temporal para un sistema de ficheros distribuido en entornos Linux con un buen rendimiento y escalabilidad.

Es por esto que la finalidad de este proyecto es doble:

- Viabilidad. Se pretende estudiar la viabilidad de la utilización de *Memcached* como sistema de ficheros distribuido. Si sus características se adecúan a las necesidades de un sistema de este tipo, plantear un diseño, implementarlo y evaluar su rendimiento en varios casos.

- Creación de una librería de entrada y salida estándar. Su estructura se asemejará lo máximo posible a la utilizada en POSIX, de modo que sea lo

más sencillo posible portar los programas ya existentes a la librería, sin olvidar que se trata de un sistema completamente experimental, por lo que factores como la evaluación del rendimiento primarán por encima de funcionalidades para usos más específicos o complejos.

A título personal, la motivación para la realización de este Proyecto Fin de Carrera es el conocimiento en profundidad de la plataforma *Memcached* y las librerías disponibles como *libMemcached*. Además, la realización de este proyecto me ofrecía la oportunidad de desarrollar un proyecto distribuido más complejo de lo habitual a lo largo de la carrera y utilizar un clúster avanzado para la puesta en funcionamiento del sistema y su evaluación.

1.2 Objetivos

Los objetivos de este Proyecto Fin de Carrera se definen de una forma abstracta, se trata de una serie de mínimos, unas guías a seguir a la hora de realizar el posterior análisis y diseño del sistema en profundidad.

- **Diseño de un sistema de ficheros distribuido basado en Memcached** para almacenar los datos en una arquitectura distribuida. Se aprovecharán al máximo las características de *Memcached* para tratar de optimizar el sistema de ficheros, así como se tendrán en cuenta las desventajas de utilizar un sistema de caché para intentar disminuir los efectos negativos que se deriven y, en caso de no poder ser evitados, contar con ellos de cara a su control y a la información al usuario.

- **Implementación de una librería** de entrada y salida portable y flexible para el nuevo sistema de ficheros. Esta librería cumplirá dos objetivos fundamentales:

    - Facilidad de portabilidad: tanto la interfaz de la librería como su funcionamiento deben ser lo más similares posibles a la que utiliza POSIX, de modo que se implementarán las siguientes llamadas: *open, read, write, lseek, unlink, rename, stat y close* y, tanto sus parámetros como sus valores de retorno se asimilarán en la medida de lo posible a los de POSIX.
    - Flexibilidad de configuración: se facilitará la modificación de parámetros (como el tamaño de bloque), la escalabilidad del sistema, etc.

- **Evaluación de rendimiento:** a pesar de que la evaluación de rendimiento no es un objetivo en sí mismo, se perseguirá durante todo el desarrollo del proyecto que la implementación de la librería facilite en lo posible el estudio del rendimiento que puede ofrecer *Memcached* como sistema de ficheros distribuido. Para ello, se explorarán todas las optimizaciones de rendimiento posibles y se implementarán de modo que puedan ser evaluadas y comparadas con opciones menos optimizadas.

1.3 Definiciones, Abreviaturas y Acrónimos

1.3.1 Definiciones

Array: también conocido como arreglo o vector. Se trata de una estructura de almacenamiento en la que se pueden introducir varios valores del mismo tipo de datos de forma consecutiva en memoria. Se puede acceder a cualquiera de las posiciones del array, tanto para lectura como para escritura, indicando su índice, que comenzará en cero. Desde un punto de vista lógico, se puede ver como una matriz de una sola dimensión (una fila o columna).

Biblioteca: ver librería.

Ethernet: Ethernet es un estándar de redes de área local para computadores con acceso al medio por contienda CSMA/CD. ("Acceso Múltiple por Detección de Portadora con Detección de Colisiones"), es una técnica usada en redes Ethernet para mejorar sus prestaciones. El nombre viene del concepto físico de *ether*. Ethernet define las características de cableado y señalización de nivel físico y los formatos de tramas de datos del nivel de enlace de datos del modelo OSI. La Ethernet se tomó como base para la redacción del estándar internacional IEEE 802.3. Usualmente se toman Ethernet e IEEE 802.3 como sinónimos. Ambas se diferencian en uno de los campos de la trama de datos. Las tramas Ethernet e IEEE 802.3 pueden coexistir en la misma red. (Fuente: *wikipedia*).

FastEthernet: versión de 100 Mbps del protocolo Ethernet (ver definición de Ethernet).

Firewall (o *cortafuegos*): sistema (hardware, software o híbrido) encargado de filtrar las comunicaciones en un segmento de red. Se encarga de limitar, bloquear, cifrar o descifrar el tráfico entre diferentes ámbitos de la red.

GigabitEthernet: versión de 1 Gbps del protocolo Ethernet (ver definición de Ethernet).

Hash: es utilizado tanto para referirse a algoritmos de resumen como para referirse al resultado de la aplicación de un algoritmo de este tipo. Los algoritmos de resumen tienen como entrada un conjunto de bytes de cualquier tamaño y devuelven una secuencia de bytes de tamaño fijo que permiten identificar unívocamente (salvo casos de colisión) el conjunto de bytes de origen. Además, la función debe ser de un solo sentido, es decir, debe ser imposible conocer ningún dato acerca del texto de origen a partir de su resumen y el resumen de un conjunto de bytes debe ser siempre el mismo.

Librería: conjunto de subprogramas diseñados e implementados para facilitar la tarea de desarrollar otros programas más complejos. Pueden contener tanto código como datos, que serán incluidos en otros programas y pasarán a formar parte de ellos. También se conoce como biblioteca, término mejor traducido del inglés "library" pero menos utilizado en España.

MemcachedFS: se trata del nombre que se ha seleccionado para referirse al sistema de ficheros distribuido basado en *Memcached* desarrollado en este Proyecto Fin de Carrera.

Multicast: envío de un solo mensaje a un grupo de nodos de destino de forma simultánea en una sola transmisión desde el nodo origen. En el caso de redes de ordenadores, se crean copias de forma automática cuando se requiere por la topología de la red, por ejemplo, en los routers.

Renderización: proceso de generación de una imagen desde un modelo. Dicho modelo puede ser desde las características de una imagen 3D (polígonos, texturas, shaders, etc.) hasta un documento HTML que describa una página web y que el navegador debe convertir a una imagen para el usuario.

Router (*enrutador* o *encaminador*): dispositivo hardware de red cuya función es la de interconectar dos o más redes de datos. Es el encargado de dirigir los paquetes de datos hacia el segmento de red correcto para que alcancen su destino. Trabaja en el nivel de red (tercera capa de OSI).

SHA-512: se trata de un algoritmo de *hash* para funciones criptográficas, una de las evoluciones de SHA-1 (diseñado por la NSA y publicado en 1991 por el NIST) como SHA-224, SHA-256 y SHA-384 (llamados comúnmente SHA2).

Script: se les conoce también como archivo de procesamiento por lotes (*batch*). Se trata de un programa sencillo, usualmente almacenado en un fichero de texto plano. Consisten en una serie de órdenes utilizadas para automatizar funciones: combinar tareas, interactuar con el sistema operativo o con el usuario. Suelen ser interpretados por la *Shell* (intérprete de comandos).

1.3.2 Acrónimos y siglas

Acrónimo	Significado
API	Application Programming Interface (interfaz de programación de aplicaciones)
DDR	Double Data Rate (doble tasa de transferencia de datos)
fd	File descriptor (descriptor de fichero)
HDD	Hard Disk Drive (Unidad de disco duro)
OSI	Open System Interconnection (Sistema abierto de interconexión)
RAM	Random Access Memory (Memoria de acceso aleatorio)
RPC	Remote Procedure Call (Llamada a procedimiento remoto)
SFD	Sistema de Ficheros Distribuido
SHA	Secure Hash Algorithm (Algoritmo de hash seguro)
SSD	Solid State Drive (Unidad de estado sólido)
Ssh	Secure SHell
XDR	eXternal Data Representation

Tabla 1: Tabla de acrónimos

1.4 Estructura del Documento

En este apartado se detallan, a modo de guía, cuáles son los capítulos en que está dividido el documento y qué es lo que el lector puede esperar encontrar en cada uno de ellos.

En el capítulo 1, INTRODUCCIÓN, se detalla tanto la motivación que dio lugar al desarrollo de este proyecto como los objetivos que se persiguen. También se ofrece información de utilidad para ayudar a la lectura y comprensión del documento, con definiciones de términos, desglose de acrónimos y siglas, y esta guía sobre la estructura del documento.

En el capítulo 2, ESTADO DEL ARTE, se introducen una serie de conceptos imprescindibles para la comprensión del resto del documento. Se ha desarrollado partiendo de los conceptos más básicos como "sistema de ficheros", "caché" o "sistema distribuido", hacia más complejos como cachés en sistemas de ficheros distribuidos. También se detalla el estado actual de la tecnología referente a los conceptos introducidos o relacionados con el proyecto.

En el capítulo 3, MEMCACHED, se detalla todo lo referente al sistema de caché distribuido utilizado como base del sistema de ficheros distribuido. Cómo funciona, por qué se eligió, su competencia, etc.

En el capítulo 4, ANÁLISIS, se ofrece una aproximación a la funcionalidad que se espera del sistema y sus límites, culminando con la exposición de un pliego de requisitos a cumplir durante el desarrollo.

En el capítulo 5, DISEÑO, se explica de forma más detallada cómo se pretenden llevar a cabo las funcionalidades acotadas en el capítulo de análisis.

En el capítulo 6, IMPLEMENTACIÓN, se detalla de forma técnica cómo han sido llevadas a cabo en el código del sistema todas aquellas decisiones tomadas en el diseño y algunas otras tomadas durante la propia implementación. Además, servirá de referencia sobre el funcionamiento de la librería para los usuarios interesados en utilizarla, de forma más explicativa y menos técnica que la documentación del código.

En el capítulo 7, VERIFICACIÓN DEL SISTEMA, se detalla el plan de pruebas llevado a cabo para verificar el correcto funcionamiento del sistema desarrollado.

En el capítulo 8, EVALUACIÓN, se evalúa el rendimiento del sistema, mediante procedimientos habituales de utilización de la librería y mediciones de tiempo. Se detallan los resultados mediante gráficas para facilitar su lectura e interpretación.

En el capítulo 9, CONCLUSIONES Y PRESUPUESTO, se exponen las conclusiones que se extraen, tanto del desarrollo del proyecto como de su evaluación. Además, se comentan brevemente una serie de puntos que no han sido desarrollados en el proyecto pero que podrían resultar de interés de cara a su ampliación. Por último, se detalla el coste que ha tenido el desarrollo del sistema mediante un presupuesto.

En el capítulo 10, BIBLIOGRAFÍA Y REFERENCIAS, se incluyen detalles sobre las fuentes de información utilizadas para el desarrollo del documento, ya sean referencias a literatura o enlaces a otros documentos o contenidos electrónicos.

En el ANEXO I: MANUAL DE INSTALACIÓN, se incluye un manual de instalación para que los usuarios aprendan a utilizar el sistema desarrollado para sus propias aplicaciones.

En el ANEXO II: CREACIÓN DE SCRIPTS PARA CLÚSTER, se detalla cómo crear scripts para ejecutar clientes que utilicen la librería en un entorno distribuido, por ejemplo, un clúster con sistema de colas.

2 ESTADO DEL ARTE

Dada la complejidad del sistema propuesto, son muchos los conceptos que es necesario conocer de cara al pleno entendimiento de su funcionamiento. Además, muchos de estos conceptos son evoluciones de otros más sencillos, con lo que es interesante, en este apartado, comenzar por los más básicos para acabar haciendo una exposición de las técnicas más avanzadas.

2.1 Sistema de ficheros

La función de un sistema de ficheros es organizar la información almacenada en los dispositivos de almacenamiento, usualmente discos duros. La mayoría de los sistemas de ficheros actualmente dividen la información en distintos bloques de datos para facilitar su accesibilidad, siendo ésta aleatoria casi en la totalidad de dispositivos de almacenamiento, es decir, no es necesario buscar desde el inicio del dispositivo de almacenamiento para acceder a un bloque, sino que puede accederse directamente a cualquier bloque.

Con respecto a los bloques son varias las características que nos interesarán de cara al diseño más óptimo posible del sistema de ficheros:

- **Tiempo de acceso:** tiempo que tarda en accederse a un bloque desde que es solicitado al dispositivo de almacenamiento. Lógicamente, se intentará minimizar este valor. Los tiempos de acceso son casi instantáneos y uniformes (se tarda lo mismo en acceder a todos los bloques, el mínimo) tanto en memoria RAM como en SSD. En discos duros, sin embargo, el tiempo de acceso a cada bloque es variable, dependiendo de la zona del disco en que se encuentre y de la posición actual de la aguja lectora.
- **Tamaño de bloque:** a mayor tamaño de bloque, más rápido será el funcionamiento del dispositivo de almacenamiento, porque tendrá que hacer menos búsquedas para una lectura de igual tamaño. Sin embargo, un tamaño de bloque grande afecta negativamente a la fragmentación, ya que, en ficheros pequeños o en el último bloque de todos los ficheros, queda mucho espacio desaprovechado.

La otra parte del sistema de ficheros, es la que hace inteligible el formato de dichos bloques de datos distribuidos por el dispositivo, estructurándolos en ficheros y directorios. Para dotar de lógica a dichos bloques de datos, los sistemas de ficheros utilizan una serie de bloques que contienen información sobre los bloques de datos. Esta información es conocida como metadatos. Para facilitar la comprensión del

funcionamiento de los metadatos se pondrá como ejemplo el concepto de *inodo* utilizado en sistemas POSIX. El *inodo* consiste en uno o varios bloques de información sobre cada uno de los ficheros que se encuentran almacenados en el dispositivo. La información que contiene acerca de los ficheros es la siguiente:

- Identificador del dispositivo que alberga el sistema de ficheros
- Identificador del inodo.
- Tamaño del fichero en bytes.
- Tamaño de bloque del sistema de ficheros.
- Número de bloques.
- Identificadores de propiedad y permisos de lectura/escritura de cada propietario (usuario, grupo y otros).
- Marcas de tiempo: última modificación (*mtime*), acceso (*atime*) y de alteración del propio inodo (*ctime*).
- Número de enlaces físicos.

Por último, el inodo contiene el identificador de todos los bloques de datos que ocupa el fichero en disco perfectamente ordenados, para poder acceder a cualquier parte del fichero. Habitualmente, estos identificadores se guardan en forma de árbol, ya que, no suelen caber todos en un solo bloque. De este modo, según el nivel del árbol en que se encuentre el identificador del bloque al que se quiere acceder, el tiempo de acceso variará: por ejemplo, si se quiere acceder a un bloque cuyo identificador está en el tercer nivel del árbol de identificadores, el tiempo de acceso al bloque será cuatro veces el tiempo de acceso aleatorio del dispositivo, ya que, se tendrán que consultar los tres niveles del árbol además del bloque de datos (salvo que alguno de los bloques esté en caché o en memoria principal).

Esta sería la arquitectura básica de un sistema de ficheros, lo mínimo que debemos implementar para que pueda ser reconocido y usado como tal. Existen multitud de aproximaciones a este enfoque como FAT32 y NTFS (Windows), ext2 (UNIX) o HFS Plus (Mac OS X). Sin embargo, existen cada vez aproximaciones más complejas destinadas a mejorar el rendimiento o la fiabilidad de los sistemas de ficheros, como el *journaling* implementado en ext3, ext4 o ReiserFS.

La otra funcionalidad mínima que debe aportar un sistema de ficheros, son los servicios necesarios para leer, modificar y eliminar ficheros. De nuevo la referencia es POSIX (entorno al que irá destinado el sistema de ficheros desarrollado y al que se intentará asemejar lo máximo posible). Los principales servicios que ofrece POSIX para tal fin son los siguientes[5]:

- *open*: abre un fichero para ser utilizado y devuelve un descriptor de fichero con el que manipularlo.
- *read*: lee el número de bytes introducidos por parámetro.
- *write*: escribe en el fichero el *buffer* de datos introducido por parámetro.
- *lseek*: desplaza el cursor que marca la posición en la que se realizará la próxima operación de lectura/escritura.

- *close*: cierra el descriptor de fichero que se introduzca por parámetro. No se podrá seguir manipulando el fichero asociado a dicho descriptor.
- *unlink*: borra del sistema de ficheros el fichero cuya ruta coincida con la introducida por parámetro.
- *rename*: renombra el fichero.
- *stat*: devuelve en una estructura los metadatos del fichero.

Todas las llamadas son transparentes al usuario y, por tanto, funcionan con bytes, independientemente de las características de la estructura que tenga el sistema de ficheros, como por ejemplo, su tamaño de bloque. Otros sistemas de ficheros, como el de Win32 (Windows) tienen diferentes nomenclaturas pero la funcionalidad de las llamadas debe ser, en esencia, la misma.

Por último, los sistemas de ficheros suelen organizar los ficheros de forma jerárquica, habitualmente en forma de árbol, para que sea más sencilla la navegación. Esto se hace por medio de directorios, que pueden contener en su interior un número indefinido de ficheros, o bien, contener a su vez otros directorios, dando lugar a la estructura arbórea.

2.2 Caché

Históricamente, el concepto de memoria caché estaba intrínsecamente relacionado con el hardware. La memoria caché se dispone en uno de los niveles superiores en la jerarquía de memoria de las computadores actuales, con mayor velocidad que la memoria RAM y menor tamaño, y por debajo de los registros del procesador, a su vez más rápidos y limitados en tamaño que la memoria caché.

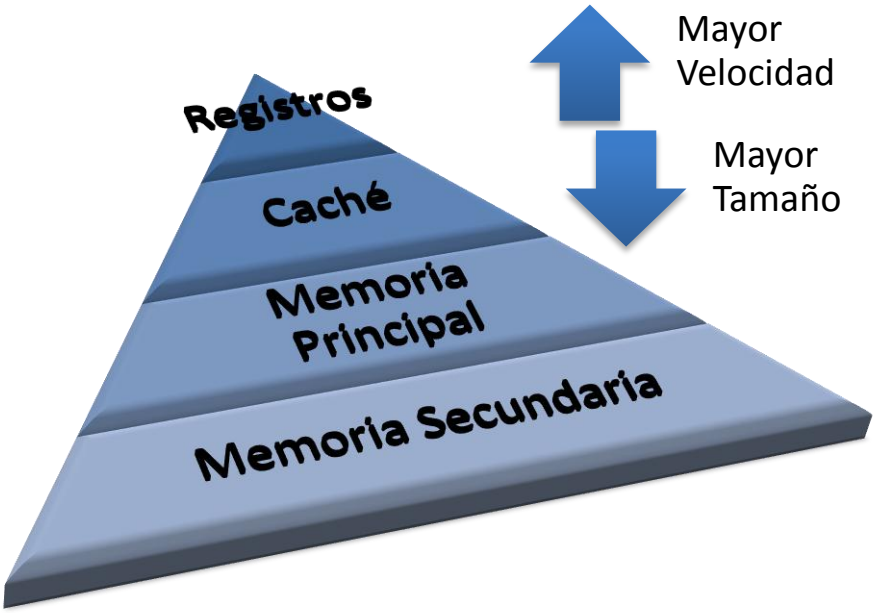

Figura 1: Estructura jerárquica de los tipos de memoria en un computador

Sin embargo, es importante conocer cuál es su funcionamiento de cara a conocer cómo se ha aprovechado este concepto en ámbitos software como caché de sistemas operativos, cachés distribuidas, etc.

La memoria caché se basa en dos principios:

- Principio de localidad temporal: si un dato de cualquiera de los niveles inferiores de memoria (memoria principal o memoria secundaria) es utilizado en un instante de tiempo, hay muchas posibilidades de que sea utilizado de nuevo en un instante posterior cercano.
- Principio de localidad espacial: si un dato de cualquiera de los niveles inferiores de memoria (memoria principal o memoria secundaria) es utilizado hay muchas posibilidades de que en un instante de tiempo posterior cercano, se utilice un dato contiguo (en el mismo bloque/página).

Tomando como base estos principios, el funcionamiento de la memoria caché consiste en tomar datos que presumiblemente serán utilizados en un futuro cercano y almacenarlos en caché. De este modo, cuando se busque cualquier dato en memoria principal o secundaria, primero se consultará en la caché, en caso de encontrarse se producirá un "acierto" caché y el dato se obtendrá mucho más rápido que accediendo a los niveles más bajos de memoria y, en caso contrario, se producirá un "fallo" caché, por lo que habrá que acceder al dato de la forma habitual, habiendo perdido un tiempo mínimo buscando en caché.

El principal problema que surge a la hora de usar la caché es que tiene un tamaño muy reducido, ya que, se opta por diseños de cachés pequeñas debido a que la velocidad de la caché es inversamente proporcional a su tamaño. Siendo tan pequeño el espacio de caché, se debe elegir muy bien los datos que deben permanecer en caché y los que deben ser eliminados para reservar espacio a nuevos datos. Para afrontar este problema aparecen las políticas de reemplazo, cuyos algoritmos más habituales son los siguientes:

- *FIFO* (First In First Out o Primero Entra Primero Sale): tal y como su nombre indica, el primer bloque que ha sido escrito en la caché, será el primero en ser expulsado cuando se necesite espacio para un nuevo dato y la caché se encuentre llena. Se trata de la misma política que siguen las colas de datos, en las que se van introduciendo datos por la cola y eliminando aquellos que están en la cabeza.
- *LRU* (Least Recently Used o Menos Recientemente Usado): probablemente la política a priori más lógica y la más utilizada, ya que, expulsa de la caché el dato que hace más tiempo que no ha sido accedido.
- *MRU* (Most Recently Used o Usado Más Recientemente): en este caso, se considera que los datos sólo suelen ser usados una vez y se elimina de memoria aquel que ha sido utilizado más recientemente.

- *LFU* (Least Frequently Used o Menos Frecuentemente Usado): se expulsa de la caché el bloque que ha experimentado menos referencias, es decir, aquel que, desde que entró en caché, ha sido solicitado menos veces.
- *Aleatoria*: se selecciona de forma aleatoria un bloque de entre todos los almacenados en caché y se expulsa para hacerle hueco al nuevo dato. A priori podría parecer el menos conveniente con diferencia, sin embargo, dada la complejidad de algunos sistemas, a penas se penaliza el rendimiento de su caché utilizándolos.

A pesar, como se ha indicado anteriormente, de tratarse de un concepto muy relacionado con el hardware, dado que existe una jerarquía de memorias en la que también existe mucha diferencia entre acceder a un dato en memoria primaria y acceder a él en memoria secundaria, cada vez es más frecuente utilizar la memoria RAM como caché para los dispositivos de almacenamiento secundario o, incluso, en sistemas más complejos como cachés para bases de datos, para páginas web, etc.

En estos casos, además de utilizarse ampliamente los principios de localidad espacial y temporal para cargar en memoria datos que se van a utilizar en el futuro, se pueden, incluso, utilizar estadísticas de uso para mantener en memoria principal, mientras sea posible, aquellas páginas web o tablas que, estadísticamente, más se utilizan, para optimizar el rendimiento del sistema.

El caso que más nos interesa dada la temática del proyecto es la caché en sistemas de ficheros. En los sistemas de ficheros existen dos tipos principales de caché, la primera está implementada mediante hardware en el propio dispositivo, ya que, la mayoría de los discos duros actuales contienen una pequeña caché (entre 8 y 64 MB) que funciona de forma completamente aislada del sistema operativo, pero que influye en su rendimiento. El otro tipo de caché que puede aplicarse a los sistemas de ficheros consiste en reservar una zona de la memoria para almacenar datos útiles para recuperar más rápidamente los bloques de la memoria secundaria:

- Caché de nombres: almacena nombres de ficheros y punteros a sus bloques de datos para ahorrar búsquedas entre los metadatos.
- Caché de bloques: incluye bloques de datos completos y datos para identificarlos unívocamente, de modo que cuando se intente acceder a un determinado bloque de memoria secundaria, primero se buscará en memoria principal a modo de caché (dando como resultado un acierto o un fallo).

La caché de un sistema de ficheros tiene un punto crítico: la coherencia entre los datos que existen en caché y los que están en disco. Si utilizamos una caché y, el equipo falla antes de que podamos volcar su contenido al dispositivo de almacenamiento persistente, se producirá una pérdida irreversible de dichos datos e, incluso, se podría corromper el sistema de ficheros. Para atenuar estos problemas, debe sincronizarse de forma habitual la caché del sistema de ficheros y los datos que se encuentran en el dispositivo de almacenamiento persistente. Para ello existen varias técnicas utilizadas habitualmente, que pueden ser incluso utilizadas de forma simultánea:

- *Escritura inmediata (write-through)*: las escrituras se hacen directamente sobre el soporte de memoria secundaria, reduciendo enormemente el rendimiento pero asegurando en todo momento la coherencia de los datos.
- *Escritura diferida (write-back):* los datos se escriben en disco en el momento en que son expulsados de la caché.
- *Escritura retrasada (delayed-write):* el sistema operativo vuelca a disco de forma frecuente los datos modificados en caché, por ejemplo, UNIX lo hace cada 30 segundos.
- *Escritura al cierre (write-on-close):* se sincronizan los bloques modificados de un fichero en el momento en que éste se cierra. Suele ser complementaria del resto y realizarse en *background* (cuando el sistema tiene poca carga).

En definitiva, dado que el sistema de ficheros a desarrollar se alza sobre la base de *Memcached* (un sistema de caché distribuida) es fundamental conocer cómo funciona una caché, no sólo a nivel hardware, sino también a nivel software y, sobre todo, en el entorno de los sistemas de ficheros.

2.3 Sistemas distribuidos

Existen dos aproximaciones para definir lo que es un sistema distribuido. Desde un punto de vista físico, se trata de un conjunto de computadoras, interconectadas a través de una red de comunicaciones y que no comparten memoria ni reloj. La otra aproximación se centra en la visión lógica del sistema, en el que varios procesos ejecutados en diferentes computadoras, se comunican entre sí mediante el uso de mensajes. De ambas visiones se puede extraer un denominador común, la posibilidad de ejecutar varios procesos y que se comuniquen entre sí con independencia de la máquina en la que se ejecuten y el lugar en el que ésta se encuentre.

Las principales características positivas que puede ofrecer un sistema distribuido son las siguientes:

- **Compartir recursos:** ya sea hardware (impresoras, sistemas de almacenamiento…), software o datos. Además, la tendencia actual es que los procesos puedan acceder a recursos remotos de forma completamente transparente, tal y como si fuesen recursos locales.

- **Escalabilidad:** en caso de necesitarse más capacidad de cómputo, los sistemas distribuidos pueden ampliarse, de forma relativamente sencilla, mediante la adición de máquinas a la red.

- **Relación precio/rendimiento:** interconectar varias máquinas comunes a una red, suele ser mucho más provechoso en cuanto a su relación precio/rendimiento, que la utilización de hardware específico avanzado de

potencia equiparable. Esto permite sustituir sistemas complejos (mainframes, servidores, supercomputadores…) que realicen funciones para ofrecer servicio concurrente a varios usuarios o aquellos diseñados para la computación paralela, por sistemas distribuidos menos costosos.

- **Tolerancia a fallos:** un sistema distribuido bien construido puede aportar grandes beneficios en la fiabilidad y disponibilidad de los servicios que ofrece, de modo que, al constar de varios nodos, puede seguir funcionando si uno de dichos nodos se cae sin perder funcionalidad (quizá perdiendo algo de rendimiento, pero lejos del problema que supondría la caída de un nodo en un entorno centralizado).

Obviamente, todas estas ventajas son a expensas de ciertos inconvenientes de los que adolecen los sistemas distribuidos:

- **Red de interconexión:** este es el principal problema con el que se encuentran los sistemas distribuidos. A pesar de los avances realizados en las redes de interconexión que se producen día a día, éstas siguen acarreando una serie de problemas.
  - *Coste:* las redes de interconexión son caras, y su precio asciende a medida que mejora la capacidad de transferencia.
  - *Saturación:* el ancho de banda de la red no es ilimitado y, usualmente, suele ser muy inferior al que ofrece, por ejemplo, una conexión directa entre el disco duro y la placa base. Penalizando el rendimiento general del sistema. Además, tal y como se ha comentado en el punto anterior, el ancho de banda de la red está directamente relacionado con su coste.
  - *Fiabilidad:* la inmensa mayoría de las redes de interconexión aún no están completamente libres de errores de comunicación y, la pérdida de un mensaje entre máquinas, puede ser crucial para el funcionamiento del sistema. Además, toda la funcionalidad del sistema depende del sistema de comunicaciones, por tanto, si la red se cae, el sistema queda completamente inutilizable.

- **Software más complejo:** el diseño y desarrollo de aplicaciones distribuidas es mucho más complejo que aquellas pensadas para ser ejecutadas en una sola máquina. El principal problema consiste en la gestión de todos los procesos que se deben ejecutar de forma concurrente en todas las máquinas del sistema y su sincronización.

- **Seguridad y confidencialidad:** para asegurar la confidencialidad de los datos que recorren la red, en la mayoría de los casos es necesario la utilización de protocolos de cifrado, que suponen un nuevo escollo en el rendimiento.

- **Potencia asimétrica de los nodos:** no todos los nodos tienen la misma potencia y, por tanto, se puede dar el caso de que si uno de los nodos es más lento que el resto, éste penalice el rendimiento del sistema completo.

2.3.1 Clúster

"A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone/complete computers cooperatively working together as a single, integrated computing resource." [Buyya98]

Esta definición de clúster podría traducirse como: un clúster es un tipo de sistema de procesamiento distribuido o paralelo, que consiste en una colección de ordenadores completos/autónomos trabajando de forma conjunta como un único recurso de computación integrado.

En definitiva, se trata de un conjunto de máquinas distribuidas que son presentadas al usuario como un solo sistema a través de un *middleware*, de manera que trabaja con él de forma transparente como si se encontrase ante una sola máquina. De este punto se encarga el SSI (Single System Image), parte fundamental en un clúster.

La principal característica que distingue a los clústeres de otros sistemas distribuidos es que los nodos que lo forman no están diseñados de forma específica para formar parte de dicho clúster. Habitualmente, las máquinas que forman parte de un clúster son computadores convencionales o domésticos, de modo que su coste es bastante ajustado. Incluso, en un mismo clúster pueden coexistir diferentes tipos de arquitecturas y sistemas operativos (x86, x64, SPARC, etc. / Windows, Linux, Solaris, etc.)

Dado que la red es un recurso fundamental en el caso de los clústeres, ésta sí que suele ser más avanzada que las redes domésticas, ya que, aunque puede utilizarse Ethernet (Fast Ethernet o Gigabit Ethernet) no es extraño ver arquitecturas más complejas y caras como Myrinet o FiberChannel, trabajando con protocolos más ligeros que el típico TCP/IP como Active Messages (U. Berkeley), VMMC, U-net (U. Cornell) o XTP (U. Virginia).

Evoluciones de los clústeres:

- **Grid:** nace como símil de la distribución de la energía en una red eléctrica. La principal diferencia con otros sistemas de computación distribuida de alto rendimiento como los clústeres, es que suelen ser débilmente acoplados, con máquinas más heterogéneas y más disperso geográficamente, por

ejemplo, los clústeres de diferentes universidades pueden ser configurados para trabajar como un Grid, de modo que la potencia de cálculo desplegada dependa de la tarea a ejecutar. El middleware más utilizado actualmente para Grid es Globus Toolkit (http://www.globus.org/).

- **Cloud computing:** considerado por muchos la alternativa comercial/empresarial a los Grid que suelen ser más propios de los entornos educativos o de investigación. De forma completamente transparente al usuario, se le ofrecen una serie de recursos (potencia de computación, servicios de almacenamiento o, incluso, aplicaciones completas) sin que éste necesite conocer en ningún momento la arquitectura en la que están corriendo, y adaptándose a sus necesidades de forma flexible. Además, cada vez está más relacionado con los servicios ubicuos, en los que el usuario acceda a la aplicación siempre con su configuración personal (preferencias, datos, etc.) sin importar el lugar desde el que se conecte. Un buen ejemplo de esto es Google Apps (GMail, Google Docs, Picasa, Gtalk, etc.) o Spotify.

- **Software como servicio (SaaS):** basado en cloud computing, la idea del software como servicio, surge de vender a las empresas la funcionalidad exacta del software que necesitan en lugar de vender paquetes o licencias. De este modo, se puede cobrar a las empresas por el número de horas que utilizan el software, los datos almacenados, los datos transferidos, etc. Además, dada la facilidad de escalar los sistemas distribuidos, se puede adaptar el servicio a las necesidades de la empresa en cada momento. Ejemplos de SaaS son Amazon EC2 (alquiler de sistemas de computación distribuida), Amazon S3 (alquiler de almacenamiento según uso), etc.

2.4 Sistemas de ficheros distribuidos

Los sistemas de ficheros distribuidos, nacen de la necesidad de integrar de forma transparente los ficheros de un sistema distribuido y, de este modo, permitir compartir datos a los usuarios de dicho sistema. Los servicios que debe ofrecer un sistema de ficheros distribuido son los mismos que los de un sistema de ficheros convencional: servicio de directorio y servicio de ficheros, sin embargo, las peculiaridades del sistema hacen que el objetivo que persiguen sea ligeramente distinto.

2.4.1 Servicio de directorio

También conocido como sistema de nombrado o gestión de nombres. El objetivo principal de este servicio es ofrecer a todos los usuarios una visión única del sistema de

ficheros, es decir, todos los clientes deben "ver" un mismo árbol de directorios y se debe ofrecer un espacio de nombres global, cumpliendo las siguientes condiciones:

- **Transparencia en la posición:** el nombre del objeto no permite obtener directamente el lugar en el que está almacenado.
- **Transparencia de la posición:** el nombre no necesita ser cambiado cuando el objeto cambia el lugar en que es almacenado.

Como objetivos secundarios, se perseguirán siempre las siguientes optimizaciones:

- **Facilidad de crecimiento:** la escalabilidad es uno de los puntos fuertes de los sistemas distribuidos y, por tanto, sería un lastre muy perjudicial que el sistema de ficheros distribuidos utilizado no permita dicha escalabilidad de forma sencilla.
- **Nombres orientados a los usuarios:** los nombres de ficheros, aunque se encuentren distribuidos por las máquinas y haya que recurrir a un servidor de nombres para descifrar su posición, deben poder ser manejados por los usuarios.
- **Replicación:** otra característica heredada de los sistemas distribuidos es la tolerancia a fallos, que también se puede ver lastrada si quedan ficheros inaccesibles en el momento en que se cae un servidor, por tanto, en la medida de lo posible los datos estarán replicados en varios servidores. Además, se evitan los cuellos de botella que puedan darse en caso de que algunos ficheros sean muy accedidos y colapsen el servidor en que se encuentran.

Existen dos aproximaciones para la implementación de este servicio, por un lado se puede implementar como un servidor centralizado al que irán todas las peticiones, con los inconvenientes que conlleva: dicho servidor puede ser un cuello de botella y si se cae se inutiliza el sistema. Por otro lado, existe la posibilidad de distribuir el servicio, de modo que cada servidor se encarga del nombrado de los ficheros que almacena. El problema que plantea este caso, como casi siempre en los sistemas distribuidos, es su complejidad, cuyo punto complicado es aquí el conocer qué conjunto de ficheros maneja cada servidor.

2.4.2 Servicio de ficheros

Su función es la gestión de los ficheros y del acceso a los datos. Para ello, existen tres modelos de acceso remoto a los datos en un sistema de ficheros distribuido:

- **Modelo carga/descarga:** cada vez que el cliente accede a un fichero, éste es transmitido por completo del servidor en el que se encuentre al cliente. Una vez en el cliente, se accede a sus datos como si se tratara de un fichero en local. La ventaja que ofrece este modelo es que el rendimiento, una vez

transmitido al cliente, es muy similar al que se tendría con un fichero local. Sin embargo, a pesar de que las transferencias son muy eficientes (todo el fichero se transmite de una vez hacia cliente, y se devuelve del mismo modo al servidor) existe una gran latencia en las llamadas para abrir el dichero y cerrarlo. Además, presenta una complicación sustancial, ya que, si varios clientes abren el fichero a la vez, existirán varias copias del mismo en local, presentándose un problema de coherencia a la hora de devolver los datos al servidor.

- **Modelo de servicios remotos:** todos los servicios de acceso a los ficheros son ofrecidos por el servidor siguiendo un modelo cliente-servidor. Normalmente el acceso a este tipo de modelos se realiza por bloques. Su principal problema es el rendimiento, ya que, todas las operaciones que se realicen sobre los ficheros, deben ser transmitidas por la red.

- **Caché en el cliente:** se trata de un modelo intermedio. Se almacenan en el cliente los bloques más recientemente accedidos o que tienen más posibilidades de ser utilizados en el futuro, de modo que el cliente siempre busca en la caché local antes de recurrir al servidor.

Además de estos modelos, otra de las características que se deben decidir en el diseño del servicio de ficheros es si tendrá estado o no. Si tienen estado, almacenan información de cómo utilizan los clientes los ficheros, como por ejemplo, el puntero de la posición de la siguiente lectura/escritura. De este modo las operaciones son más eficientes (no tienen que ser autocontenidas, por tanto, se ahorra información en cada comunicación), sin embargo, al tener que guardar información sobre cada cliente, se penaliza el número de clientes simultáneos que se pueden tener y, además, se tiene mayor dependencia entre el cliente y el servidor, penalizando también la tolerancia a fallos.

2.4.3 NFS

Las siglas NFS se corresponden con Network File System, o lo que es lo mismo, Sistema de Ficheros en Red, se trata de una especificación y su correspondiente implementación de un servicio de acceso a ficheros remotos, y sirve como ejemplo para ilustrar el funcionamiento de un sistema de ficheros distribuido, ya que, se trata de uno de los más utilizados en la actualidad.

Está diseñado para trabajar en entornos heterogéneos, es decir, las máquinas que forman parte del sistema distribuido pueden tener distinta arquitectura, diferente sistema operativo, etc. Para conseguir esto, se utilizan RPC (llamadas a procedimientos remotos) construidas sobre el protocolo XDR, consiguiendo que el formato de las llamadas sea completamente independiente de la plataforma. Las llamadas que se pueden realizar son las siguientes:

- Búsqueda de un fichero en un directorio.
- Lectura de entradas del directorio.
- Manipulación de enlaces y directorios.
- Acceso a los atributos de un fichero.
- Lectura y escritura de ficheros.

Es importante destacar que los servidores de NFS no almacenan estado, es decir, cada llamada debe ser auto contenida, contener toda la información necesaria para ser realizada. Para ello, se utilizan manejadores de ficheros. En caso de que el cliente esté gestionando un fichero local, el *vnode* (metadatos de NFS) apuntará a un *inodo* del sistema de ficheros, mientras que si se trata de un fichero remoto, el *vnode* incluirá este manejador de fichero. Son, por tanto, los clientes los encargados de guardar la posición (*offset*) y cualquier otra información necesaria para mantener la coherencia entre llamadas. Además, el protocolo no ofrece mecanismos de control de concurrencia para asegurar una semántica UNIX, por tanto, hay que prestar especial atención a la coherencia de los datos si se accede desde dos clientes a un mismo fichero remoto.

Mientras que en la versión 2 de NFS solo se permitía el uso de *write-thorugh* en las escrituras, a partir de la versión 3 se incluye un modo *delayed-write* en el que solo se escribe en memoria caché hasta que se hace un *commit* desde el cliente, normalmente al cerrar el fichero. Para solucionar el problema de coherencia en la caché, se actualizan periódicamente los datos en el servidor, caducando aquellos datos que estén en caché y sean más antiguos que los del servidor.

Por último, cabe destacar, que la imagen del sistema de ficheros no es constante en todos los clientes, ya que, cada uno puede montar el sistema de ficheros distribuido en cualquier punto de su sistema de ficheros local.

2.5 Caché en sistemas de ficheros distribuidos

El funcionamiento de una caché en un sistema de ficheros distribuido es muy similar al de cualquier otra caché en la jerarquía de memoria (ver sección 2.2). En este caso, en lugar de utilizarse una memoria más rápida que la secundaria (como RAM o caché), los bloques se copian del servidor al cliente, para trabajar con ellos de forma local. Las mejoras que puede ofrecer este tipo de caché son las siguientes:

- **Proximidad referencial:** como en la caché habitual, se explotan los principios de proximidad temporal y espacial.
- **Lectura adelantada (*prefetching*):** se realizan lecturas adelantadas de bloques antes de que sean solicitados. De este modo, se puede optimizar el rendimiento solapando las acciones de E/S (traer los datos del servidor) mientras la aplicación procesa los datos que ya tiene.

- **Escritura diferida o retardada:** en lugar de tener que enviar al servidor los bloques cada vez que se hace una modificación, esta trasferencia se retrasa, de modo que solo se tenga que transferir el bloque una vez. Las técnicas para hacerlo son las mismas que se vieron en el apartado 2.2: *write-through*, *delayed-write*, *write-on-close* y *write-back* (esta última es difícil que se dé, a menos que el disco duro de los clientes sea muy pequeño o carezcan de él).

Además de los problemas habituales, el diseño de este tipo de caché, requiere enfrentarse a ciertas características especiales.

- **Localización de la caché:** existen dos posibilidades, memoria principal o memoria secundaria. La memoria principal es mucho más rápida, sin embargo como puntos negativos están que su tamaño es mucho más limitado y que, dada su volatilidad, si el cliente se cae, se pierden los datos almacenados en caché.
- **Granularidad:** tamaño de los bloques de la caché. Puede oscilar desde partes del fichero hasta ficheros completos. Cuanto mayor es el tamaño de bloque, más posibilidades de acierto hay y, por tanto, hay que recurrir de forma menos frecuente a la red. Sin embargo, aumenta la latencia de las operaciones sobre los ficheros, ya que, cuando hay un fallo caché, la transferencia es mayor y ocupa más espacio en el sistema local.
- **Coherencia:** cada cliente tiene su propia caché, de modo que puede haber diferentes copias del mismo fichero, dando lugar a problemas de coherencia.

2.6 Memoria principal vs memoria secundaria

Históricamente se ha entendido siempre la memoria principal como memoria RAM y la memoria secundaria como discos duros. Las diferencias entre ambos tipos de memoria eran abismales, sin embargo, actualmente cada vez es más habitual encontrarnos con sistemas que disponen de unidades de estado sólido (SSDs) que reducen las distancias entre ambos tipos de memoria. Sin entrar a realizar una comparación exhaustiva entre los sistemas, es interesante ver las diferentes características que ofrecen.

2.6.1 HDD vs SSD

La principal diferencia entre estos dispositivos es su coste y su rendimiento (en el que se centra el siguiente punto). El coste por gigabyte de los discos duros se encuentra actualmente entre los 3 y los 10 céntimos de euro (0.03€ y 0.10€) mientras que en el caso de los SSD de ámbito doméstico, el precio por gigabyte asciende hasta la cifra de entre 1.20€ y 2.00€ entre 20 y 40 veces el precio de los discos duros. Además,

se suelen comercializar en tamaños mucho más pequeños (debido, en parte también, a su elevado coste).

Otras diferencias que cabe destacar, esta vez a favor de los SSD es que son absolutamente silenciosos, más ligeros, más pequeños, consumen menos energía y no tienen ninguna parte móvil, por lo que son más tolerantes a fallos en sistemas portables. Como contrapartida, sus celdas sufren de cierto grado de desgaste por cada escritura/borrado que reciben, dando lugar con el tiempo a sectores inutilizables y pérdidas de rendimiento. En este ámbito, las partes móviles de los discos duros también pueden sufrir fallos con el tiempo, dando lugar a la inutilización completa del disco.

2.6.2 Rendimiento

Son muchos los factores de rendimiento que se pueden medir, sin embargo, vamos a centrar la comparativa en dos de ellos: tiempo de acceso y velocidad de transferencia.

El **tiempo de acceso** de la memoria es el tiempo desde que se solicita un dato hasta que se encuentra en el disco y está listo para empezar a ser transferido. Tanto en los SSD como en la memoria RAM, este tiempo es constante y siempre el mínimo que permite el dispositivo. Sin embargo, en los discos duros es variable y depende de la posición que ocupe el dato en el disco y de la posición de la aguja lectora en el momento de la petición. Dado que no se busca una comparativa exhaustiva sino informativa acerca del rendimiento general de estos dispositivos, los siguientes valores[7][8] servirán como una aproximación válida para entender las diferencias.

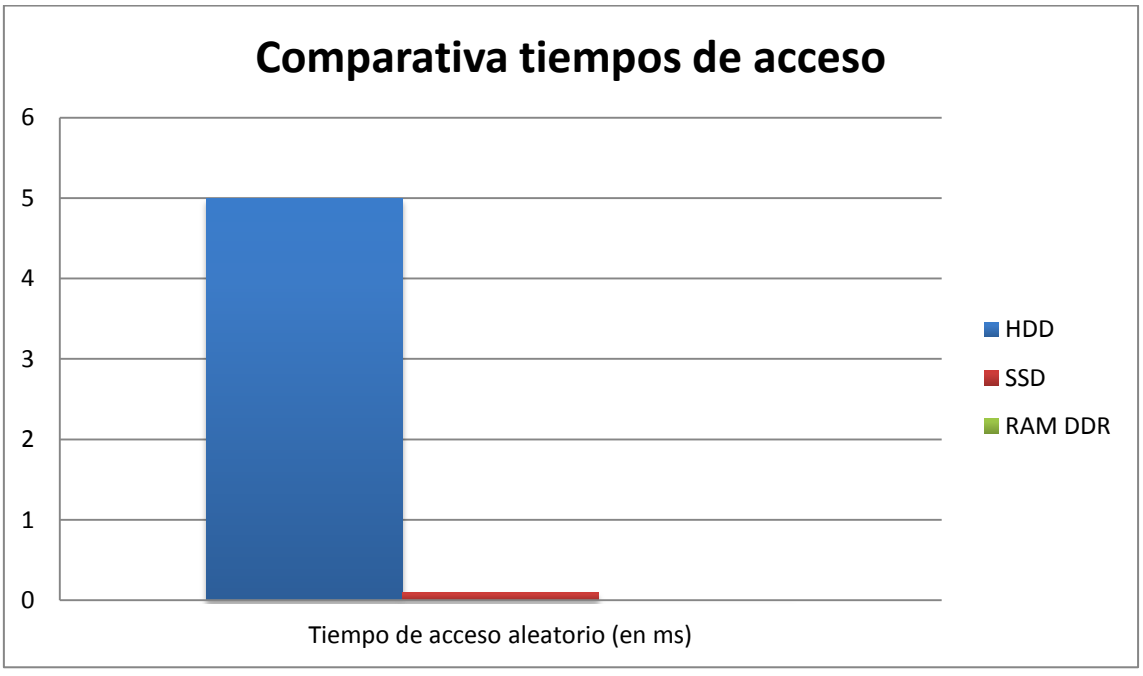

Figura 2: Gráfico comparativo de tiempos de acceso HDD vs SSD vs RAM

Como se puede observar en el gráfico, los tiempos de acceso de cada dispositivo están en órdenes de magnitud distintos. Los discos duros tradicionales tienen un tiempo de acceso de en torno a 5-10 ms (se ha puesto el menor de los valores para mejorar la visibilidad) mientras que los tiempos de acceso de los SSD están en torno a los 0.1 ms y la memoria RAM de la familia DDR (la más habitual actualmente en su versión DDR3) sería de unos 10 ns, es decir, $10x10^{-6}$ ms, varios órdenes de magnitud por debajo de cualquier tipo de dispositivo de memoria secundaria. En el caso de la RAM, se puede tomar como medida de sus tiempos de acceso sus valores de latencia.

La siguiente métrica de comparación a utilizar es la referente a velocidades de transferencia. De nuevo los discos duros sufren de variaciones en su rendimiento de transferencia en función de la posición de los datos mientras que los SSD y la RAM no, dependiendo de si todos los datos a leer están alineados o no (favoreciéndose las lecturas/escrituras secuenciales sobre las aleatorias). Sin embargo, de nuevo los datos[7][8] son meramente orientativos para entender las diferencias de rendimiento. A la comparativa se añade la velocidad de transferencia de las redes Ethernet más habituales (FastEthernet – 100Mbps teóricos y GigabitEthernet 1000Gbps teóricos).

Nota: en todos los casos, las velocidades de lectura/escritura se resienten en caso de bloques pequeños, por tanto, se tomará como referencia escrituras de varios Megabytes.

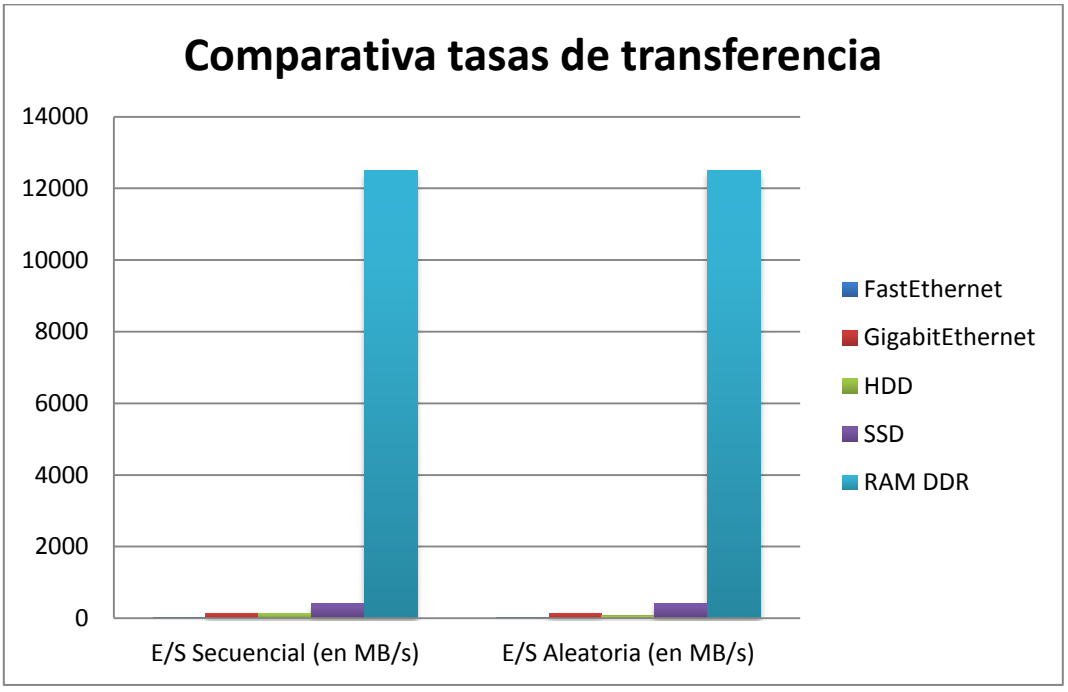

Figura 3: Gráfico comparativo de tasas de transferencia Ethernet vs HDD vs SSD vs RAM

En esta primera gráfica se observa como la velocidad de transferencia de la memoria RAM DDR está varios órdenes de magnitud por encima del resto, en torno a 12.5 GB/s de velocidad de transferencia para la memoria RAM por velocidades entre 12.5 y 400 MB/s para el resto.

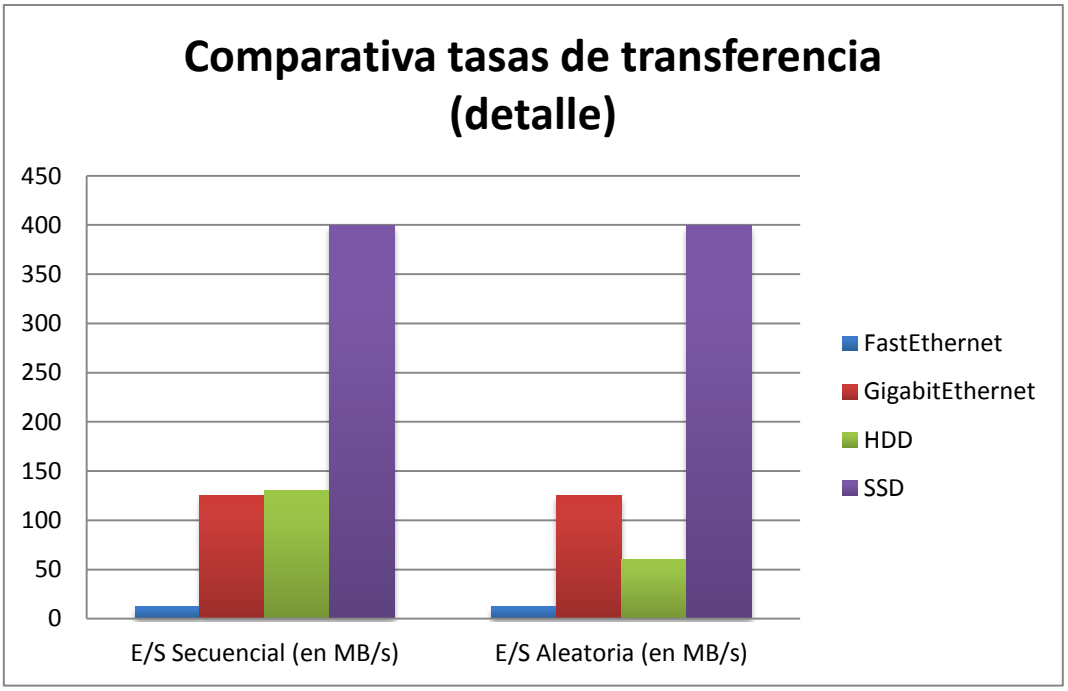

Figura 4: Gráfico comparativo tasas de transferencia Ethernet vs HDD vs SSD

Para facilitar el análisis, en esta segunda gráfica se elimina la RAM, para ver mejor las diferencias entre el resto. Lo que se observa es que la velocidad de los SSD es bastante superior al resto y no varía entre lecturas/escrituras secuenciales y aleatorias, manteniéndose en torno a 400 MB/s, mientras que los HDD consiguen en torno a 125 MB/s en modo secuencial, bajando hasta los 65 MB/s cuando las lecturas/escrituras son aleatorias. Como información adicional, se han añadido a la gráfica las velocidades teóricas de FastEthernet y GigabitEthernet, aunque en la práctica, incluso en condiciones óptimas, estas velocidades no se alcanzarán nunca pero de nuevo, sirven como indicativo aproximado de su rendimiento.

Las conclusiones son las esperadas. La memoria principal se encuentra varios órdenes de magnitud por encima en cuanto a rendimiento en cualquier contexto, mientras que las diferencias entre SSDs y HDDs, se presentan principalmente en cuanto a tiempos de acceso y lecturas/escrituras aleatorias, éstas últimas en un margen razonable. Por último, se observa que el rendimiento teórico de una red GigabitEthernet se podría acercar en tasas de transferencia a las de un disco duro tradicional, un dato muy interesante de cara a los sistemas de ficheros distribuidos.

3 MEMCACHED

Memcached merece un capítulo en exclusiva dentro de la descripción de este proyecto, ya que, se trata de una de las piezas clave del sistema de ficheros distribuido propuesto, y su creciente popularidad, es una de las principales motivaciones del proyecto.

3.1 ¿Qué es Memcached?[9]

Sistema de cacheo de objetos en memoria distribuida, de alto rendimiento, libre y de código abierto. Genérico en su naturaleza, pero ideado para su uso en la aceleración de aplicaciones web dinámicas aliviando la carga en los accesos a bases de datos relacionales.

Memcached es un sistema de almacenamiento en memoria de pares clave-valor para pequeños fragmentos de datos (cadenas de caracteres, objetos) procedentes de resultados de llamadas a la base de datos, llamadas a un API o renderizado de páginas.

Memcached es sencillo pero también potente. Su sencillo diseño facilita despliegues rápidos, desarrollos fáciles y resuelve muchos problemas referentes a cachés de datos grandes. Sus APIs están disponibles para la mayoría de los lenguajes de programación más populares. Fue desarrollado inicialmente por Danga Interactive para el popular servicio de *blogging* LiveJournal pero actualmente es libre y es utilizado por múltiples sitios web como: YouTube[1], Facebook[2], Twitter[3] o Tuenti[4].

3.2 Funcionamiento de Memcached

Memcached permite aprovechar zonas de la memoria RAM que están infrautilizadas para darle otro uso. En el siguiente gráfico se puede ver con más claridad dicho funcionamiento mediante un ejemplo. Se supone el caso de que dos servidores web tengan 64 MB de RAM no utilizados, lo máximo que pueden hacer con ellos es dos cachés independientes de 64 MB que, en caso de albergar el mismo servicio web, probablemente tengan un contenido muy similar, con datos duplicados en ambas cachés.

Sin embargo, gracias a *Memcached*, las dos zonas de 64MB se pueden ver como un único espacio caché y, de este modo, compartir datos cacheados por ambos servidores, duplicando el espacio útil dedicado a caché utilizando exactamente la misma memoria que antes estaba inutilizada.

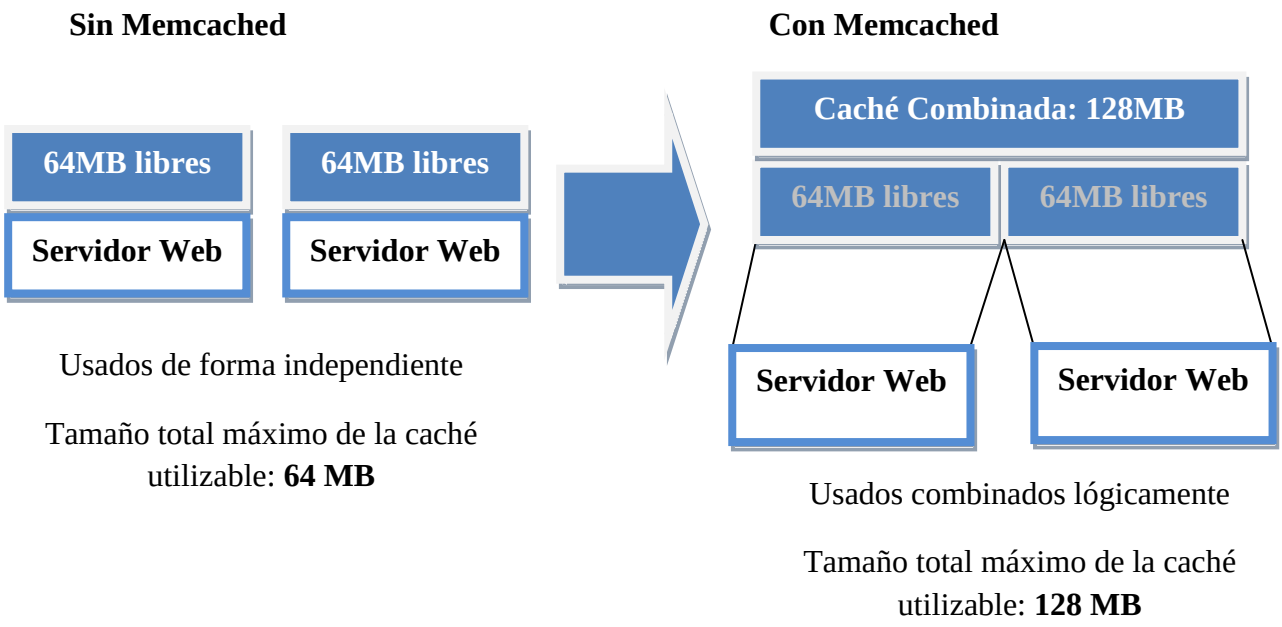

Figura 5: Ejemplo esquemático de utilización de Memcached. Esta solución proporciona un sistema de memoria compartida y distribuida

Además, de este modo, todos los servidores web de un sistema utilizarán un único pool de datos, de modo que todos los servidores del clúster buscarán los datos en la misma localización.

En el caso de los servidores web, es habitual que a medida que crecen los requisitos de computación de los nodos (se necesitan más servidores para hacer frente a una mayor cantidad de peticiones) también crece el tamaño de los datos accedidos regularmente. De este modo, tiene sentido introducir nuevos servidores en el clúster, dejando más espacios de memoria libres que pueden ser utilizados por *Memcached*.

Por supuesto, este es sólo un ejemplo de configuración de *Memcached*, también es muy habitual otra aproximación: tener varios servidores dedicados en exclusiva para ser usados como caché distribuida.

3.3 Puesta en marcha del sistema

En primer lugar, los clientes deben establecer conexión con todos los servidores que van a ser utilizados, es decir, los servidores deben estar corriendo *Memcached* y los clientes deben conocer sus direcciones IP y los puertos en los que están escuchando las peticiones.

Una vez establecida la conexión, los clientes pueden utilizar el total de memoria dedicada para caché distribuida que tengan todos los servidores como si fuese una única tabla *hash*. Es decir, los clientes podrán introducir ('set') todos los pares clave-valor que deseen y recuperarlos ('get') con total transparencia, sin necesidad de conocer en ningún momento en qué servidor están almacenados los datos.

Las claves tendrán una longitud máxima de 250 bytes, mientras que los datos asociados a la clave deben ser siempre inferiores a 1 Megabyte. En el caso de usar prefijos en las claves para delimitar los espacios de nombres, éstos deben incluirse en los 250 bytes. Cuanto más pequeñas sean las claves, menos espacio ocuparán en la caché (más espacio para datos) y más livianas serán las transferencias por red de las llamadas a *Memcached*.

Una de las principales ventajas que ofrece *Memcached* es que el cliente que recupere una clave en el sistema, no tiene porqué ser aquél que la introdujo, tan sólo debe conocer el identificador con el que fue introducida. Es aquí donde reside la principal "magia" de *Memcached*. Su *hash* de dos etapas permite que funcione como una gran tabla hash. Cuando se hace una búsqueda de clave, lo primero que hace el cliente es un hash entre la clave y la lista completa de servidores. De este modo, conoce en qué servidor se encuentra alojado el dato y envía una petición a dicho servidor, que realiza la segunda etapa, en la que el servidor hace una búsqueda interna para encontrar el ítem. Por ejemplo:

Se tienen los clientes 1, 2 y 3 y los servidores A, B y C. El cliente 1 intenta introducir la clave "key" con el valor asociado "value". Realiza el *hash* con una librería de *Memcached* entre la clave y la lista completa de servidores (A,B,C) y, por ejemplo, elige el servidor B para alojar los datos y establece una conexión directa con dicho servidor para llevar a cabo la operación. Poco después, el cliente 2 quiere recuperar el valor de la clave "key" y, para ello, ejecuta la misma librería que el cliente 1 entre la clave y la lista completa de servidores, obteniendo el mismo resultado, que el cliente 1: el dato está alojado en el servidor B. Solo resta establecer una conexión pidiendo la clave "key" al servidor B para que éste le devuelva el valor "value".

De este modo, tanto los clientes como los servidores son completamente independientes entre sí, no necesitan en absoluto comunicarse para conocer en qué servidor se encuentra cada valor. No existen intercomunicaciones que lastren el rendimiento del sistema o complejos protocolos *multicast* para mantener actualizadas las referencias, reduciendo así las interdependencias y consiguiendo una mejor tolerancia a fallos ante la caída de cualquier cliente o servidor y una escalabilidad más sencilla (los servidores no dependen unos de otros, sólo hay que agregar un nuevo servidor para ampliar el espacio de caché).

Con respecto a su comportamiento como caché, cabe destacar su política de reemplazo y su política de caducidad. Como primer punto, su política de reemplazo es LRU (Least Recently Used), es decir, como se ha indicado en apartados anteriores, cuando la caché se llena y se necesita espacio para un nuevo elemento, se expulsa de la

caché aquel que fue utilizado hace más tiempo. La política de caducidad lo que hace es eliminar los elementos de la caché pasado un tiempo, este tiempo no viene dado por el servidor, sino que es el cliente el que indica el tiempo que quiere que el dato se mantenga en memoria (se puede esquivar la política de caducidad introduciendo 0 como tiempo de caducidad). La ventaja que ofrece esta política es simple: en algunos sistemas existen valores muy costosos de calcular que sólo son válidos durante un periodo de tiempo determinado, la función que cumple este método es meter el dato en la caché con caducidad y a la hora de necesitarlo mirar si está en caché, si da acierto caché la información es válida y se mostrará directamente, si se produce un fallo caché, la información debe volver a ser calculada antes de ser mostrada. Para evitar sobrecargas, el sistema que invalida las referencias por caducidad tiene precisión de un segundo, por tanto, el dato puede ser invalidado desde casi un segundo antes de lo indicado por el cliente hasta casi un segundo después.

Por último, es interesante comentar las características de *Memcached* con respecto a la persistencia de los datos. 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, tal y como se especifica en el FAQ[11] oficial, los datos deben ser modificados primero en base de datos y después en caché. Es por esto, que la persistencia está dejada de lado. 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, se perderá la información que contenían.

La información no está replicada de ninguna manera y *Memcached* no tiene absolutamente ningún mecanismo para administrar las caídas. Las aplicaciones deben ser diseñadas considerando esta restricción, si se cae un servidor, todas las claves que están almacenadas en él desaparecen, pero además, dado el funcionamiento del algoritmo de hash, habrá una serie de claves que no podrán ser almacenadas en ningún otro servidor hasta que se vuelva a poner en funcionamiento el que cayó (o uno en su misma IP) o se modifique la lista de servidores, aunque esta última opción es poco recomendable, ya que, sería como *resetear* toda la caché, la mayoría de las calves tendrían como destino un servidor distinto del actual. Actualmente, existen algunos algoritmos y librerías para tener algoritmos consistentes de *hash*, en los que cambiando alguno de los servidores de la lista, el resto de referencias siguen apuntando donde apuntaban antes de la modificación, como *libketama*[1].

[1] http://www.last.fm/user/RJ/journal/2007/04/10/392555 "libketama" a consistent algorithm for memcache clients

3.4 Alternativas a Memcached

La elección de *Memcached* no es arbitraria. Un sistema tan utilizado tiene bastantes competidores, sin embargo, la mayoría de ellos no son genéricos como *Memcached*, que aunque fue implementado siempre teniendo en mente la optimización de consultas a bases de datos de páginas web dinámicas, también se mantuvo siempre como base de diseño que fuese genérico.

De este modo, existen multitud de alternativas que tratan de hacerle competencia en sectores y aplicaciones concretas. Un ejemplo de ello son los aceleradores/cacheadores de PHP[10] como *APC* (Alternative PHP Caché), *eAccelerator* o *XCache*. Además, 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 a diseño de un sistema de ficheros distribuidos, 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, es decir, en lugar de utilizar *Memcached* como sistema de cacheo de una base de datos SQL como pueda ser MySQL, las alternativas proponen cambiar el enfoque completo de base de datos para mejorar el rendimiento y la escalabilidad en los entornos en que se suele utilizar *Memcached*. Algunos ejemplos son *Cassandra*, *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. En esencia puede trabajar como servidor de *Memcached* y es 100% compatible con los clientes desarrollados para *Memcached*, por tanto, de cara a pruebas de rendimiento como es una de las pates principales de este proyecto, no hay mucha diferencia con respecto al uso de *Memcached* y, en caso de obtener resultados satisfactorios, el usuario final podría decidir entre desplegarlo sobre *Memcached* o pagar la licencia y utilizar *Membase* y las ventajas que aporta.

Por último, existe otra alternativa, en este caso de funcionamiento muy similar a *Memcached*, pero aportando algunas funcionalidades extra como persistencia y replicado de los datos, llamado *Redis*. Sin embargo, hay tres motivos principales por los que no se ha elegido. El primer motivo es su escasa popularidad. *Memcached* es utilizado actualmente por la mayoría de las webs punteras y, por tanto, especialmente interesante utilizarlo de cara a comprender su funcionamiento. El segundo motivo es

que, existen algunos análisis de rendimiento[13][14] en comparación con *Memcached* que lo ponen ligeramente por debajo, por lo tanto, no es adecuado para analizar el rendimiento que podría tener un sistema de ficheros distribuido utilizando como base un sistema de caché distribuida. El tercero y más importante de todos es que el enfoque distribuido de *Redis* no es para nada transparente, cada vez que se introduce un dato en memoria, se debe decir a qué servidor concreto se quiere enviar, rompiendo la transparencia acerca de la localización que tiene *Memcached* y perdiendo también muchos puntos de cara a su escalabilidad. Se trata, por tanto, de un sistema de tabla *hash* en memoria **en red** y no de un sistema de caché **distribuido**.

3.5 Memcached vs memcache

Existe cierta confusión en torno a si *Memcached* y memcache son o no son lo mismo. Desde la web de Membase (de los creadores de *Memcached*) se deja claro:

"Una pregunta habitual es "¿Cuál es la diferencia entre memcache vs. Memcached?"

La respuesta corta es: ninguna. La respuesta larga es que, debido a que ***memcache*** *es ejecutado en "background" en los sistemas Linux (y esto es considerado un "demonio"), el archivo de programa utilizado para arrancar el software es llamado memcached para seguir las convenciones de nombrado de demonios. Así que, técnicamente, memcache se refiere al software y Memcached al nombre del archivo de programa. Pero la mayoría de la gente simplemente usa ambos en este momento."* (Traducción)

Hay algunos motivos que han contribuido a que esta confusión sea frecuente. La primera es que, a pesar de que el software es conocido como *memcache*, el proyecto actualmente se llama *Memcached*, haciendo parecer que son implementaciones distintas. El segundo motivo es que existen dos librerías de *Memcached* para PHP desarrolladas de forma completamente independiente y que ofrecen distinta funcionalidad, una llamada *memcache* y otra *memcached*, de modo que existen bastante comparativas entre ellas, pudiendo llevar a confusión.

3.6 ¿Por qué utilizar Memcached como base de un sistema de ficheros distribuido?

Para justificar la decisión de uso de *Memcached* como base de un sistema de ficheros distribuido como el que se quiere diseñar, se recurrirá a las características

planteadas anteriormente que indican las características, tanto imprescindibles como opcionales, que debe cumplir un sistema de ficheros distribuido.

Los sistemas distribuidos deben constar de dos servicios fundamentales: el servicio de directorio y el de ficheros. Las características imprescindibles que debe cumplir un servicio de directorio son: transparencia en la posición y transparencia de la posición. El sistema de hash en dos etapas que tiene *Memcached* nos ayudará enormemente para conseguirlo, ya que, el nombre del fichero no estará restringido por la posición que ocupará y, únicamente con el nombre de fichero, será imposible determinar en qué servidor está almacenado. Como características opcionales, se tienen las siguientes:

- **Facilidad de crecimiento:** *Memcached* es un sistema muy fácilmente escalable. Añadir un nuevo servidor es tan sencillo como levantarlo, arrancar *Memcached* y configurar los clientes para que sepan dónde se encuentra. El principal problema es que un cambio en la lista de servidores hace que la mayoría de las claves cambien de posición en el sistema y, por tanto, también los ficheros. Afortunadamente, se puede solucionar mediante el uso de algoritmos de *hash* consistentes.
- **Nombres orientados a los usuarios:** los nombres de los ficheros serán completamente libres, el usuario podrá usar la nomenclatura que desee y será el sistema de hash el encargado de distribuir los datos por los servidores de forma transparente.
- **Replicación:** esta es la primera desventaja de *Memcached* para su uso en un sistema de ficheros distribuidos, ya que, no permite replicación. La solución pasaría por lanzar dos servicios de *Memcached* en puertos distintos y guardar los datos en ambos. Sin embargo, esta solución es inútil por varios motivos: no se garantiza que el sistema de *hash* envíe los bloques a un servidor distinto en todos los casos cambiando el puerto y, el principal problema, dada la nula orientación de *Memcached* a la persistencia, la caída de un servidor es fatal, por lo que la replicación no aportará mucha seguridad.

El otro servicio imprescindible es el servicio de ficheros. *Memcached* permite utilizar sus llamadas como un modelo de servicios remotos en el que cada escritura de bloque sea un “set” y cada lectura un “get”. Además, al tratarse de un sistema localizado enteramente en memoria principal, su rendimiento debería ser bastante bueno (al menos en algunos tipos de uso), mitigando en parte la necesidad de utilizar cachés locales.

3.7 libMemcached

Existen multitud de clientes para facilitar la comunicación con *Memcached*. De modo que no haya que utilizar la interfaz binaria o indagar en su protocolo de

comunicación. Dentro de los clientes existentes, los requisitos del proyecto exigían que éste fuese utilizable en C y Linux.

La elección ha sido libMemcached, la librería para *Memcached* en C más conocida, y utilizada por sitios como Twitter o Yahoo. Además, también es utilizada como base para la creación de muchos clientes en otros lenguajes de programación, entre ellos el cliente "memcached" para PHP, considerado por muchos como el mejor cliente para PHP, uno de los ámbitos de utilización más frecuente de *Memcached*.

libMemcached[15] es un conjunto de herramientas y una librería cliente de C/C++ y de código abierto para servidores *Memcached*. Ha sido diseñada para ser ligera en uso de memoria, con seguridad en hilos y proporciona acceso completo a los métodos del lado del servidor. Ha sido desarrollada por Brian Aker bajo licencia BSD con el objetivo de proporcionar el mayor número de opciones para utilizar *Memcached*, algunas de sus principales características son:

- Soporte de transporte tanto síncrono como asíncrono.
- *Hashing* y distribución consistente.
- Algoritmo de *hashing* tuneable para la comparación de claves.
- Soporte para el acceso a objetos grandes.
- Replicación local.
- Guía completa de referencias y documentación de la API.
- Herramientas para administrar las redes *Memcached*.

Las características más interesantes de cara a su utilización en un sistema de ficheros distribuido son la ligereza en uso de memoria, que permitiría usarse en clientes ligeros con almacenamiento exclusivamente distribuido y el *hashing* consistente (que se puede activar de forma opcional), que permite añadir en cualquier momento que se requiera más espacio, nuevos servidores sin que la distribución de las claves cambie con el cambio en la lista de servidores (como ocurre en la implementación predeterminada de *Memcached*). En menor medida, también es útil la funcionalidad de replicación que aporta, aunque siga siendo muy difícil asegurar la persistencia de datos en un sistema de este tipo.

4 ANÁLISIS

El análisis de un sistema consiste en buscar y especificar cuál es la funcionalidad que se espera desarrollar. Debe centrarse especialmente en identificar absolutamente todas las funcionalidades que se deben implementar y buscar sus límites, de modo que sirva como objetivos mínimos a cumplir en el momento en que se comience con el diseño.

En este caso, se realizará primero una descripción general del sistema a desarrollar y, como segundo punto, se detallarán una serie de requisitos tanto de capacidad como de restricción.

4.1 Descripción general

El objetivo del sistema será el desarrollo de un sistema de ficheros distribuido que utilice como base para la distribución de los datos el sistema de caché distribuida *Memcached*, al que se llamará MemcachedFS.

Para ello, se implementará una librería en lenguaje C para Linux que permita utilizar *Memcached* como si de un sistema de ficheros local se tratase, es decir, la librería contará con una interfaz lo más parecida posible a la ofrecida por POSIX para disponer de un sistema de entrada y salida estándar.

4.1.1 Sistema de ficheros distribuido

La principal funcionalidad que ofrecerá el sistema será la de sistema de ficheros distribuido. Para ello se utilizará la funcionalidad ya ofrecida por *Memcached*, en la que se tiene una tabla *hash* de pares clave-valor distribuida entre varios servidores de forma totalmente transparente al usuario. Gracias a ello, se podrá usar una entrada de este mapa *hash* como si de un bloque de datos (de hasta un megabyte) se tratara. En dichos bloques de datos, al igual que sucede con los bloques de un dispositivos de memoria secundaria (discos duros o SSDs) se guardará cualquier tipo de datos, ya sean metadatos de los ficheros o sus bloques de datos.

Existen multitud de librerías que ofrecen una interfaz de comunicación con *Memcached*. Se ha elegido *libMemcached* por ser la librería más potente de las que existen para C, de hecho, muchos de los clientes implementados para otras plataformas están realizados utilizando esta librería para comunicarse con *Memcached*. Ésta será la capa externa con la que la librería que se implemente se comunicará con *Memcached* para hacer las conexiones a los servidores, introducir datos en la caché (“set”), obtener

datos de la caché (“get”), etc. Se deben aprovechar, siempre que sea 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.

Para asemejarse al estándar de entrada y salida de POSIX, se implementarán, al menos, las funciones básicas de dicho estándar con una sintaxis que sea lo más semejante posible a ella. Para ello, se implementarán las funciones de *open, read, write, lseek, unlink, rename, stat y close*.

Es importante desatacar que, de cara a mejorar y simplificar la evaluación de rendimiento que tendría un sistema de ficheros distribuido basado en *Memcached*, se procurará facilitar la modificación de parámetros que afecten al rendimiento como el tamaño de bloque del sistema de ficheros. Esta condición experimental del desarrollo significa también que los requisitos no son absolutamente inalterables como ocurriría en un proyecto software destinado a la implementación de un sistema que cubra la necesidad de un cliente, sino que se podrán ir adaptando a medida que se avance el desarrollo del sistema para adaptarse a una evaluación de rendimiento más precisa y/o sencilla.

4.1.2 Caché de sistemas de ficheros locales

Una vez diseñado el sistema de ficheros y todas las funciones de gestión de ficheros (creación, lectura, escritura y borrado de ficheros) se puede reutilizar gran parte de ese diseño e implementación para utilizar *Memcached* como sistema de caché distribuida para un sistema de ficheros local. Para ello existen dos aproximaciones que se han denominado modo pre-caché y modo caché.

El primer modo, pre-caché, es muy sencillo: en el momento en que se abre el fichero local, se copia todo su contenido al sistema de ficheros distribuido basado en *Memcached*. A partir de ese momento, todas las funciones de acceso al archivo funcionan exactamente como si de un sistema de ficheros distribuido se tratase, pero con las posibles mejoras de rendimiento que puede aportar trabajar sobre memoria principal distribuida en lugar de sobre un dispositivo de memoria secundaria. En el momento en que se termina de trabajar con el fichero, el cliente realizará un *close* y todos los datos serán volcados de la caché distribuida al dispositivo de almacenamiento situado de forma local en el cliente, lo que en el mundo de las cachés se conoce como *write-on-close*.

El segundo modo de funcionamiento, es mucho más similar al funcionamiento de una caché. En lugar de copiar al sistema de ficheros distribuido todo el contenido del fichero en la apertura, se hace tal y como se haría en una caché. Cada vez que cualquiera de los datos de uno de los bloques que ocuparía el fichero en caché distribuida, es tocado (leído o escrito) todo el bloque es llevado a caché y se trabaja con él siempre desde allí a partir de ese momento. De nuevo, el método de sincronización es

write-on-close, cuando el cliente hace la llamada para cerrar el fichero, se devuelven a su dispositivo de almacenamiento local aquellos bloques (y solo aquellos) que han sido modificados en la caché distribuida. De este modo, se mejora la latencia de la apertura y se minimizan las comunicaciones por la red al mínimo posible. La política de reemplazo vendrá dada por *Memcached* y será LRU (least recently used).

Con respecto a la seguridad en la persistencia de los datos, *Memcached* dificulta mucho las cosas en este aspecto: no permite replicación, no hay forma de saber cuál es el próximo bloque que va a expulsar de caché para devolverlo a disco antes de que se pierda, su tolerancia a fallos en uno de los servidores es nula, etc. Dado que este es un sistema experimental, se ha decidido no prestar especial atención a este apartado, de modo que para que el sistema funcione, una vez inicializado no puede caerse ninguno de los servidores que forman parte de MemcachedFS y todos los ficheros que estén abiertos deben tener espacio suficiente en la caché distribuida configurada, de lo contrario, en el momento en que uno de los bloques fuese expulsado de caché, se perdería su información de forma irrecuperable.

4.2 Definición de requisitos

Todo análisis es culminado mediante la redacción de un pliego de requisitos en el que se especifiquen las funcionalidades que cubrirá el sistema y las restricciones con las que se debe contar a la hora de implementarlo, todo esto, sin entrar a explicar cómo se conseguirá dicha funcionalidad.

Para facilitar la lectura y trazabilidad de los requisitos, éstos serán plasmados en el documento rellenando una tabla con los siguientes campos:

- **Identificador:** identificará de forma unívoca cada uno de los requisitos que se redacten. La sintaxis a utilizar a la hora de seleccionar el identificador del requisito será la siguiente:

  RU<tipo>-<número>

  - RU: hace referencia a la abreviatura “Requisito de usuario”, dada la procedencia habitual de los requisitos en la fase de análisis.
  - <tipo>: tipo de requisito, será “C” para los requisitos de capacidad (aquellos que hacen referencia a funcionalidades del sistema) y “R” para los requisitos de restricción.
  - <número>: será un número de tres cifras que empezará por 001 y crecerá en una unidad por cada requisito.

- Ejemplos: los requisitos de capacidad se identificarán como RUC-001, RUC-002, etc., mientras que los de restricción serán RUR-001, RUR-002, etc.

- **Nombre:** nombre corto que servirá para identificar y referirse a un requisito de una forma más intuitiva y fácil de recordar que el identificador y más corta que su descripción.

- **Descripción:** breve comentario textual que define el requisito. Debe ser claro y conciso.

- **Necesidad:** determina si el requisito es imprescindible a la hora de desarrollar el sistema o no lo es. Sus valores serán "Esencial", "Deseable" u "Opcional".

- **Estabilidad:** indica si el requisito tiene posibilidades de ser alterado a lo largo del diseño y el desarrollo del sistema, puede ser a petición del cliente o por otras circunstancias que así lo requieran, o por el contrario va a ser mantenido íntegro con total seguridad a lo largo del proyecto. Los valores que tomará son "Estable" o "No estable".

- **Prioridad:** nivel de preferencia que se le dará a la implementación del requisito con respecto a los demás durante el desarrollo del sistema. Sus valores serán "Alta", "Media" o "Baja".

- **Fuente:** indica el origen del que procede el requisito. En el caso de este proyecto, los valores que podrá tomar son "Tutor" o "Alumno".

4.2.1 Requisitos de capacidad

Identificador: RUC-001

Nombre	Desarrollo de librería
Descripción	El resultado final del desarrollo será una librería de programación orientada a desarrolladores.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 2: Requisito de capacidad RUC-001

Identificador: RUC-002

Nombre	Sistema de ficheros distribuido
Descripción	La librería permitirá gestionar un sistema de ficheros en una caché distribuida.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 3: Requisito de capacidad RUC-002

Identificador: RUC-003

Nombre	Datos en caché distribuida
Descripción	Todos los datos del sistema de ficheros distribuido, ya sean metadatos de los ficheros o bloques de datos, se almacenarán en la memoria caché distribuida. Ésta a su vez, deberá estar almacenada por completo en memoria principal. Será éste el único método utilizado para distribuir los datos entre los servidores.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 4: Requisito de capacidad RUC-003

Identificador: RUC-004

Nombre	Servicios del SFD
Descripción	El Sistema de Ficheros Distribuido ofrecerá los dos servicios imprescindibles: el servicio de directorio y el servicio de ficheros.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 5: Requisito de capacidad RUC-004

Identificador: RUC-005

Nombre	Transparencia del servicio de directorio
Descripción	El servicio de directorio ofrecerá tanto transparencia en la posición como transparencia de la posición.
Necesidad	Deseable
Estabilidad	Estable
Prioridad	Media
Fuente	Tutor

Tabla 6: Requisito de capacidad RUC-005

Identificador: RUC-006

Nombre	Sin tolerancia a fallos
Descripción	El sistema no proporcionará ninguna tolerancia a fallos. Sólo funcionará mientras no falle ninguno de los servidores con los que fue inicializado y mientras haya espacio en la caché distribuida para evitar las expulsiones.
Necesidad	Esencial
Estabilidad	No estable
Prioridad	Alta
Fuente	Tutor

Tabla 7: Requisito de capacidad RUC-006

Identificador: RUC-007

Nombre	Modelo de servicios remotos
Descripción	El servicio de ficheros utilizará el modelo de servicios remotos para su implementación.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 8: Requisito de capacidad RUC-007

Identificador: RUC-008

Nombre	Funciones del servicio de ficheros
Descripción	El servicio de ficheros permitirá abrir, cerrar, renombrar y borrar ficheros, mover el puntero de posición, leer y escribir con el fichero abierto. También permitirá acceder al estado del fichero (metadatos).
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 9: Requisito de capacidad RUC-008

Identificador: RUC-009

Nombre	Interfaz semejante a POSIX
Descripción	Las funciones del servicio de ficheros tendrán una interfaz lo más parecida posible a la ofrecida por POSIX, así como también, un funcionamiento lo más parecido a POSIX.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 10: Requisito de capacidad RUC-009

Identificador: RUC-010

Nombre	Modificación tamaño de bloque
Descripción	Se permitirá la modificación del tamaño de bloque en la librería por usuarios expertos.
Necesidad	Deseable
Estabilidad	No estable
Prioridad	Media
Fuente	Alumno

Tabla 11: Requisito de capacidad RUC-010

Identificador: RUC-011

Nombre	Aprovechamiento de las extracciones múltiples
Descripción	El sistema aprovechará las extracciones múltiples simultáneas (“mget”) que ofrece la librería libMemcached.
Necesidad	Deseable
Estabilidad	No estable
Prioridad	Baja
Fuente	Alumno

Tabla 12: Requisito de capacidad RUC-011

Identificador: RUC-012

Nombre	Extracciones múltiples y simples
Descripción	El sistema permitirá utilizar los principales métodos de lectura tanto con extracciones múltiples simultáneas (“mget”) como con las extracciones simples (“get”) equivalentes.
Necesidad	Opcional
Estabilidad	No estable
Prioridad	Baja
Fuente	Alumno

Tabla 13: Requisito de capacidad RUC-012

Identificador: RUC-013

Nombre	SFD como caché
Descripción	El sistema de ficheros distribuido podrá ser utilizado como caché distribuida para un sistema de ficheros local.
Necesidad	Deseable
Estabilidad	Estable
Prioridad	Media
Fuente	Tutor

Tabla 14: Requisito de capacidad RUC-013

Identificador: RUC-014

Nombre	Tres modos de apertura
Descripción	El sistema permitirá tres modos de apertura del fichero: normal (SFD), pre-caché y caché.
Necesidad	Opcional
Estabilidad	No estable
Prioridad	Baja
Fuente	Alumno

Tabla 15: Requisito de capacidad RUC-014

Identificador: RUC-015

Nombre	Modo pre-caché
Descripción	En el momento de abrir un fichero local, todo su contenido será volcado a la caché distribuida. A partir de ese momento, se utilizará como cualquier otro fichero del sistema de ficheros distribuido hasta su cierre.
Necesidad	Deseable
Estabilidad	No estable
Prioridad	Media
Fuente	Tutor

Tabla 16: Requisito de capacidad RUC-015

Identificador: RUC-016

Nombre	Modo Caché
Descripción	Permitirá abrir ficheros locales y, cada vez que sea necesario, volcar a la caché distribuida el contenido del bloque con el que se está trabajando. Las operaciones de lectura y escritura funcionarán como el modo normal del SFD, de forma totalmente transparente al usuario hasta su cierre.
Necesidad	Opcional
Estabilidad	No estable
Prioridad	Baja
Fuente	Alumno

Tabla 17: Requisito de capacidad RUC-016

Identificador: RUC-017

Nombre	Write-on-close
Descripción	En cualquiera de los dos modos de caché (pre-caché y caché) se actualizará el estado del fichero del dispositivo de almacenamiento local con las modificaciones que hayan sucedido durante el tiempo que ha estado abierto en el momento de su cierre de forma transparente al usuario.
Necesidad	Deseable
Estabilidad	No estable
Prioridad	Media
Fuente	Tutor

Tabla 18: Requisito de capacidad RUC-017

Identificador: RUC-018

Nombre	Inicialización del sistema
Descripción	El usuario será el encargado de inicializar el sistema, indicando la IP y puerto de los servidores de *Memcached* que se utilizarán.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 19: Requisito de capacidad RUC-018

4.2.2 Requisitos de restricción

Identificador: RUR-001

Nombre	Memcached
Descripción	Se utilizarán servidores de *Memcached* como sistema de caché distribuida.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 20: Requisito de restricción RUR-001

Identificador: RUR-002

Nombre	Lenguaje C
Descripción	La librería se desarrollará en el lenguaje de programación C y podrá ser utilizada por otros clientes en C.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 21: Requisito de restricción RUR-002

Identificador: RUR-003

Nombre	Linux
Descripción	La librería se desarrollará para ser utilizada en sistemas Linux.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 22: Requisito de restricción RUR-003

Identificador: RUR-004

Nombre	libMemcached
Descripción	La librería de comunicación con los servidores de *Memcached* será libMemcached.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 23: Requisito de restricción RUR-004

Identificador: RUR-005

Nombre	Sistema distribuido
Descripción	El sistema podrá correr en un número indeterminado de servidores (siempre igual o superior a uno) en los que se esté ejecutando *Memcached*.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 24: Requisito de restricción RUR-005

Identificador: RUR-006

Nombre	Tamaño de bloque
Descripción	El tamaño de bloque del sistema de ficheros distribuido estará comprendido entre 1 y 1023 Kbyte.
Necesidad	Esencial
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 25: Requisito de restricción RUR-006

Identificador: RUR-007

Nombre	Tamaño de clave
Descripción	El tamaño de clave será como máximo de 250 bytes.
Necesidad	Alta
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 26: Requisito de restricción RUR-007

Identificador: RUR-008

Nombre	Longitud máxima nombre de fichero
Descripción	La longitud máxima del nombre de fichero en el sistema de ficheros distribuido será de 249 caracteres (la restricción viene impuesta por el RUR-007).
Necesidad	Alta
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 27: Requisito de restricción RUR-008

Identificador: RUR-009

Nombre	Documentación
Descripción	Tanto el código fuente como la librería estarán documentados detalladamente. Toda la documentación y comentarios del código fuente se escribirán en inglés.
Necesidad	Alta
Estabilidad	Estable
Prioridad	Alta
Fuente	Tutor

Tabla 28: Requisito de restricción RUR-009

Identificador: RUR-010

Nombre	Doxygen
Descripción	A partir de los comentarios del código, debidamente formateados para ser compatibles con Doxygen, se generará documentación detallada del código en varios formatos (al menos, en HTML) mediante dicha herramienta.
Necesidad	Media
Estabilidad	Estable
Prioridad	Media
Fuente	Alumno

Tabla 29: Requisito de restricción RUR-010

5 DISEÑO

5.1 Arquitectura

La librería se ha dividido en dos ficheros de código fuente (memcachedfs.c y utils.c) con sus respectivos ficheros de cabecera (memcachedfs.h y utils.h). En el primer fichero, "memcachedfs" que significa Memcached-file-system, se incluyen todas las funciones que el cliente necesitará para gestionar los ficheros del sistema de ficheros distribuido, es decir, lo que se conoce como el servicio de ficheros, así como otras funciones que modifican variables globales como la tabla de descriptores de fichero para evitar problemas al externalizarlas. En el otro fichero, "utils", se incluyen todas las funciones auxiliares de la librería.

Además, se ha procurado mantener en todo momento un aislamiento entre libMemcached y la librería distribuida, de modo que si fuese necesario cambiar esta librería, se pudiese hacer modificando la menor parte posible de código. Para ello, se ha estructurado la librería en capas, siendo la primera capa (memcachedfs) la encargada de implementar el sistema de ficheros sin absolutamente ninguna referencia ni llamada a libMemcached, mientras que la encargada de esto ha sido la de "utils". Mediante esta capa, se ha conseguido abstraer el funcionamiento, haciendo que tenga un funcionamiento similar al de un emulador de un driver para libMemcached.

Desde el sistema se llama a las funciones de utilidades y son éstas las que convierten los datos al formato que deben tener para ser utilizados con libMemcached y realizan la llamada a las funciones de libMemcached. Básicamente, su función es traducir llamadas genéricas a un sistema de caché distribuida en llamadas a libMemcached. Podría ser incluso posible cambiar *Memcached* por otro servidor de similares características tan solo con cambiar esta capa, y que el sistema de ficheros siguiese funcionando, al menos en la teoría, ya que, a pesar de que no se usa absolutamente ninguna llamada a libMemcached, es posible que algunas de las decisiones lógicas tomadas estén fuertemente relacionadas con las características que ofrece *Memcached*.

El resto de funciones que incluye la capa "utils", son funciones auxiliares de MemcachedFS, aquellas funciones que no interfieren directamente con las variables globales del sistema de ficheros y que no interesan en absoluto al cliente a la hora de utilizarlo: generadores de identificadores, parseadores de nombres de ficheros, etc.

Para comprender mejor el sistema de capas, veamos el siguiente gráfico:

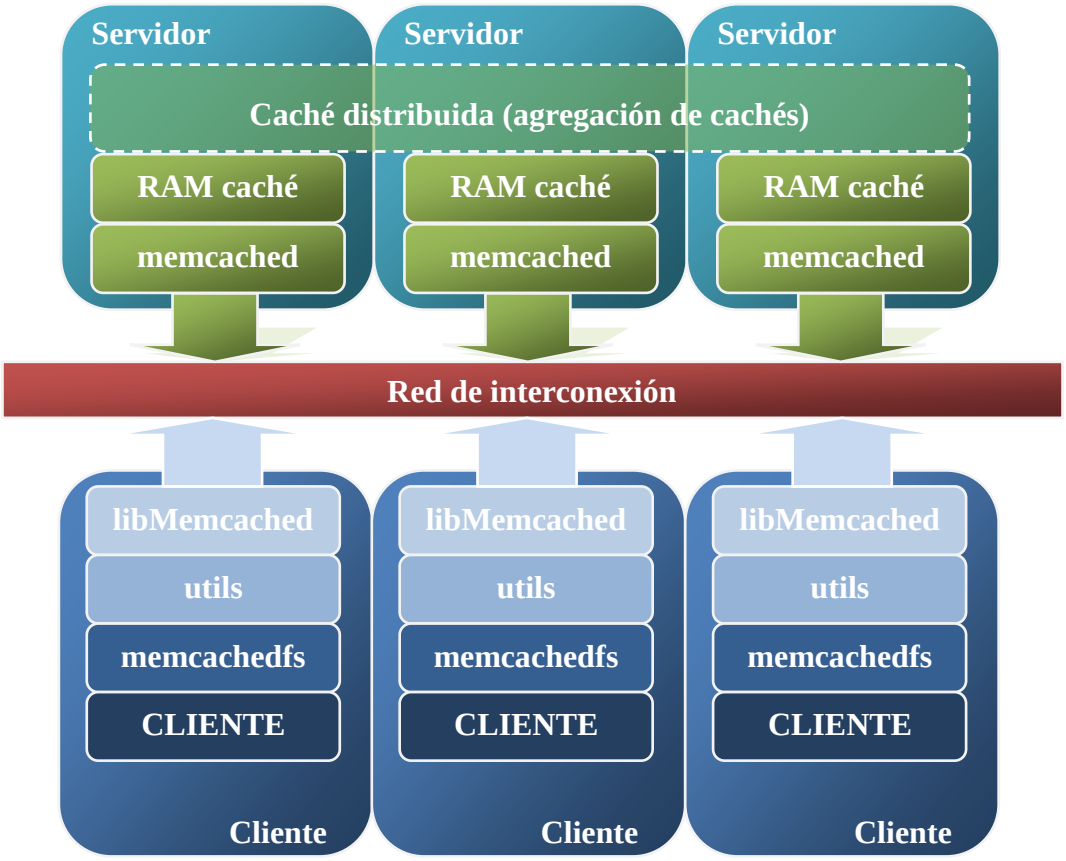

Figura 6: Esquema de funcionamiento (capas) del sistema

Nota: el número de clientes y servidores es completamente arbitrario. El sistema no tiene porqué estar balanceado entre clientes y servidores.

En el gráfico se muestra la arquitectura de capas del sistema. El "CLIENTE" sería el software implementado por el usuario, que sólo haría uso de la librería "memcachedfs". A partir de ahí, ésta utiliza "utils" para utilizar los métodos auxiliares y para *traducir* las llamadas de lectura y escritura de datos al formato comprensible por libMemcached. A partir de aquí, el funcionamiento no está desarrollado por este proyecto. libMemcached hace el *hash* de la clave, encuentra el servidor al que le pertenece y se comunica por red con él para leer/escribir el valor.

Otra característica a destacar es el bajo nivel de acople entre sí tanto de los clientes como de los servidores. Por un lado, no es necesario que los clientes se comuniquen entre sí en ningún caso, pueden acceder a cualquier clave sin que el cliente que la introdujo en la caché distribuida tenga que informarle de en qué servidor se encuentra. Del mismo modo, los servidores funcionan de forma completamente independiente unos de otros, no es necesario que conozcan ningún dato sobre el resto de servidores (pueden, incluso, añadirse o eliminarse servidores de *Memcached* sin afectar a los que están activos) ni sobre ninguno de los clientes. El servidor sólo se levanta con las características requeridas y queda a la espera de peticiones.

El único acople existente es en el sentido cliente-servidor, ya que, es necesario que todos los clientes conozcan la IP y el puerto en que están escuchando todos los servidores y que la red esté configurada para que puedan comunicarse con ellos (*routers*, *firewalls*, etc.)

Un ejemplo de arquitectura hardware para utilizar el sistema sería el siguiente:

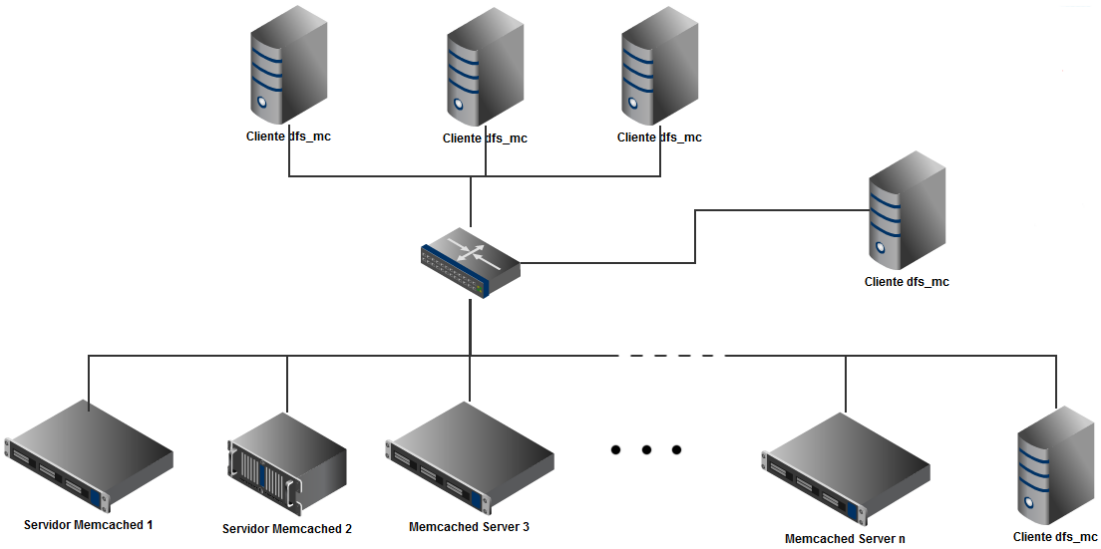

Figura 7: Ejemplo de arquitectura del sistema

A destacar de este ejemplo:

- **Servidores heterogéneos:** los servidores de *Memcached* no tienen porqué ser exactamente iguales entre sí. Lo único que deben compartir es la versión de *Memcached* que utilicen para evitar problemas de compatibilidad. A parte de esto, su arquitectura, cantidad de memoria, etc. puede ser completamente distinta y será aprovechada al máximo por el sistema. Es recomendable que tengan un tamaño similar de memoria RAM para optimizar la distribución de claves, pero no es obligatorio.

- **Número de servidores:** podrá haber tantos servidores de *Memcached* como necesite el sistema, con un mínimo de uno. Deben mantenerse constantes a lo largo del tiempo, de lo contrario, a pesar de ser una reconfiguración muy sencilla, se requeriría reiniciar MemcachedFS (todas las claves se deberían invalidar).

- **Situación de los clientes:** los clientes no tienen porqué estar en la misma red que los servidores, simplemente es necesario que sean capaces de acceder a absolutamente todos los servidores de *Memcached* que formen parte del sistema de ficheros distribuido. Lógicamente, cuanto mejor sea la

red de interconexión entre los clientes y los servidores, mejor será el rendimiento del sistema.

- **Número de clientes:** podrá haber tantos clientes como sean necesarios y situarse en cualquier punto de la red, siempre que se mantenga lo explicado en el caso anterior, es decir, que todos sean capaces de acceder a todos los servidores. No es necesario en ningún caso que los clientes sean capaces de comunicarse entre sí salvo para funcionalidades externas a MemcachedFS como podría ser la gestión de la concurrencia en el acceso a un mismo fichero por dos o más clientes.

5.2 Diseño del sistema MemcachedFS

Los datos se distribuirán por los distintos servidores mediante la propia funcionalidad que ofrece *Memcached*, y por tanto, se utilizará la caché distribuida como si de una tabla/mapa *hash* se tratase. Lo único necesario para almacenar un bloque de datos o de metadatos en la caché distribuida, es asignarle un identificador como clave. A parte de este hecho, se podrá utilizar la caché distribuida como cualquier otro dispositivo de almacenamiento.

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.

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. Éste debe tener 249 caracteres o menos. De esta forma, cada vez que se abra el archivo, sólo necesitaremos su nombre de fichero para acceder a su contenido, tanto datos como metadatos. Como se puede suponer, 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 aportar una funcionalidad muy concreta, sin aportar nada de cara al estudio de viabilidad. La única manera de trabajar con directorios es de forma explícita, es decir, si el usuario nombra sus ficheros con rutas de hasta 249 caracteres y se refiere a ellos siempre mediante la ruta absoluta, puede tener ficheros con nombres como: “/myfiles/file” y a la vez “/myfiles/important_files/file”. La única restricción que existe para los nombres de fichero, además de la longitud de 249 caracteres, es que no pueden incluir el símbolo ‘>’.

Bloques de datos: de cara a la generación de identificadores para los bloques de datos de un fichero, el esquema a seguir ha sido el siguiente:

```
<identificador_fichero> + '>' + <num. bloque>
```

La motivación de usar el carácter '>' es que no pueda existir ningún fichero en el sistema cuyo nombre coincida con el identificador de un bloque de datos, porque en ese caso, habría confusión entre si es el bloque de metadatos de ese fichero o el bloque de datos de otro. La elección del carácter tampoco es arbitraria, se ha buscado un carácter no permitido en los nombrados de UNIX (se utiliza para redireccionar la salida a un fichero) y mnemotécnicamente recuerda a una flecha que indica el bloque a acceder.

Para asegurar que la longitud del identificador del fichero sea siempre la misma y para asegurar que el bloque de datos pertenece exactamente al fichero al que debe estar asociado y no a uno con su mismo nombre, dicho identificador se consigue mediante la aplicación de SHA2-512 sobre la concatenación del nombre del fichero + el segundo de creación del fichero + el milisegundo de creación del fichero.

```
SHA2-512( file_name + create_time(sec) + create_time(millisec) )
```

De esta forma, se consigue uniformidad en la longitud de los identificadores, así como la seguridad de que se identifica a un fichero concreto y no a uno de características similares, siendo el identificador de cada fichero único y distinto al resto.

La elección de SHA-512 se basa principalmente en el tamaño de los hash que genera, ya que, se necesitaba un algoritmo resistente a colisiones pero que usase 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 espacio más que suficiente para el resto de información, 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 concatenando el símbolo ">" y el número de bloque al que se desea acceder. Esta decisión de diseño aporta ventajas fundamentales que serían imposibles de conseguir en un sistema de ficheros habitual.

Dado que la clave a la que están asociados cada uno de los bloques de datos puede ser asignada y no viene asignada por el dispositivo (como suele ocurrir en los dispositivos habituales de almacenamiento), podemos prescindir por completo de guardar en los metadatos del fichero los punteros a los bloques que éste ocupa. Esto nos

aporta dos beneficios: ahorramos espacio de almacenamiento, ya que no es necesario guardar el puntero a los datos (este beneficio se ve mitigado en parte por tener que guardar siempre clave+valor) y, sobre todo, un beneficio de 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 duros, 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.

La otra ventaja aportada tiene que ver con el tamaño máximo de un fichero. Mientras que en la mayoría de los sistemas de ficheros, éste viene dado por la cantidad de punteros a bloques de datos que se pueden guardar en las estructuras (por ejemplo, dentro del inodo), en el caso del MemcahcedFS, el tamaño de fichero viene dado directamente por la longitud de la clave. En el diseño elegido, se utilizan 128 caracteres para el identificador y uno para el carácter especial “>” que nos indica que estamos ante un bloque de datos. Por tanto, hasta los 250 caracteres máximos que puede tener una clave nos quedan un mínimo de 120 caracteres para indicar el número de bloque que se quiere obtener (son 120 caracteres en el caso de los ficheros abiertos en modo caché que incluyen dos caracteres ‘>’ como se verá más adelante).

La idea de eliminar los punteros de los bloques de metadatos vino motivada porque parecía inútil tener que almacenar 250 bytes de metadatos por cada bloque de datos (de mínimo 1 Kbyte) además de los, como máximo, 250 bytes que almacena el propio sistema de caché distribuida al guardar el par clave-valor. En el caso más desfavorable, tendríamos que por cada Kbyte de datos, se tendrían que guardar en los servidores otros 500 bytes, algo realmente ineficiente. El límite mínimo impuesto al tamaño de bloque viene dado también por este hecho, se desaprovecharía demasiado espacio guardando bloques de menos de 1 Kbyte y su correspondiente clave de 250 caracteres.

Sin embargo, esta decisión de diseño, hace de MemcachedFS un sistema muy especial. No existe en la actualidad ningún sistema de ficheros convencional que permita tamaños de fichero de este calibre y mucho menos acceder a cualquiera de sus bloques con un solo acceso. Todo ello manteniendo una gran flexibilidad en el tamaño de bloque, de ser necesario, podría incluso reducirse por debajo del Kbyte, aunque aumentarlo por encima de los 1023 Kbyte implicaría la modificación del código fuente de *Memcached*. Además, no existe fragmentación, ni interna ni externa, ya que, si un bloque ocupa menos de su tamaño máximo, sólo se guardan los datos útiles, no se deja espacio para rellenar el bloque.

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.

- **Identificador (id):** se trata del identificador que se acaba de detallar. SHA2-512(nombre fichero + hora de creación). Es imprescindible guardar el identificador en los metadatos para poder renombrar el fichero renombrando únicamente la clave del bloque de metadatos. Una vez renombrado ese bloque, se seguirá accediendo a los bloques de datos de igual forma.
- **Modo:** el modo de apertura del fichero (solo lectura, solo escritura o ambas). La implementación de permisos es muy elemental debido a que es un sistema experimental.
- **Modo de apertura:** indica si el fichero fue creado como un fichero normal del sistema de MemcachedFS o, por el contrario, se creó para usarlo en modo pre-caché o caché. (Se puede ver más sobre esta decisión de diseño en el apartado 5.3).
- **Modo original:** permisos que tenía el fichero local que se está utilizando en modo caché o pre-caché cuando fue abierto.
- **Identificadores de propietarios:** ID del usuario propietario e ID del grupo propietario. Actualmente no aporta ninguna funcionalidad, se mantiene de la estructura "stat" para facilitar la implementación de permisos en el futuro.
- **Tamaño**: tamaño total que ocupa el fichero en bytes.
- **Tamaño de bloque:** indica el tamaño de bloque que tenía MemcachedFS en el momento en que fue creado.
- **Marcas de tiempo:** tiempo de último acceso (apertura), tiempo de última modificación (escritura), tiempo de último cambio (lectura).

Otras de las decisiones de diseño que se deben tomar a la hora de implementar un sistema de ficheros (sea distribuido o no) son las referentes a la tabla de descriptores 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 solo 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.

Los campos elegidos para las entradas de la tabla de descriptores de fichero son los siguientes:

- **Nombre de fichero:** se guarda el nombre del fichero para poder acceder a su bloque de metadatos cuando se quiera realizar una operación sobre él. Además, si este campo no ha sido rellenado (valor "\0"), significa que ese descriptor no está en uso (no está inicializado o ya se ha cerrado el fichero y no apunta a ningún fichero abierto). Podría haberse guardado el identificador del fichero, de modo que se pudiese acceder directamente a los bloques de datos sin necesidad de consultar los metadatos, sin embargo, siempre que se accede a algún bloque de datos, es necesario acceder a los metadatos para actualizar las marcas de tiempo, por tanto, es preferible guardar en el descriptor el nombre del fichero (que da acceso a los metadatos del fichero y, junto a ellos, al identificador) que el identificador o ambos (el identificador y el nombre de fichero) que ocuparía más espacio.
- **Nombre de fichero original:** esta característica fue introducida para el modo de funcionamiento como caché de un sistema de ficheros local (del cual se detallará su diseño más adelante). Se guarda el nombre del fichero original para poder devolverlo a su lugar (incluida la ruta de directorios) en el momento de cerrarlo.
- **Posición del puntero:** en este valor se indica la posición en la que comenzará la próxima lectura/escritura.
- **Modo:** modo de escritura en el que ha sido abierto el fichero. Solo lectura, solo escritura o ambas. Este modo sí se respeta para evitar, por ejemplo, escrituras accidentales sobre ficheros abiertos con la intención de solo lectura. Sin embargo, no se comprobarán los permisos del fichero, solo el modo en que fue abierto.
- **Modo de apertura:** también ha sido introducido para el modo de caché de ficheros locales. Es necesario para saber el modo en que fue abierto al intentar hacer el cierre sin tener que recurrir a los metadatos (ahorrando accesos a memoria caché distribuida).
- **Tabla caché:** de nuevo otra modificación introducida para los modos de caché. En este caso, guarda una tabla con los bloques del fichero que se encuentran en caché distribuida y cuáles no. En apartados siguientes se detalla porqué la tabla está hecha en memoria principal del cliente y no en la caché distribuida.

5.3 Diseño de caché del sistema MemcachedFS

5.3.1 Modo pre-caché

En el momento en que se terminó de diseñar el modo normal de MemcachedFS, surgió la idea de utilizar el sistema de caché distribuida para el uso que realmente está diseñado. En este caso, dado que se está trabajando con sistemas de ficheros, se pensó en utilizar las estructuras diseñadas para cachear ficheros de un dispositivo de almacenamiento local.

La primera aproximación es inmediata. Es tan sencillo como copiar el fichero desde el disco local hasta la caché distribuida y trabajar con él desde allí. Antes de cerrar el fichero se devuelve al dispositivo local para guardar los cambios.

Este proceso se ha automatizado mediante una llamada para abrir los ficheros en modo "pre-caché". Este modo copia a la memoria distribuida de forma automática el fichero que se encuentra en el dispositivo local en el momento de hacer un "open" y lo devuelve a disco en el momento en que se hace un "close". Para permitir este tipo de funcionamiento, se han tenido que realizar algunas modificaciones sobre el sistema de ficheros original.

Los primeros cambios han sido en las estructuras: tanto en los metadatos del fichero como en las entradas de la tabla de descriptores de fichero. En los metadatos del fichero ha sido necesario incluir el modo de apertura. Los ficheros creados en modo "pre-caché" pueden ser abiertos en modo normal, ya que, se copian como si fuesen ficheros normales. Sin embargo, no es posible utilizar en modo "pre-caché" ficheros distribuidos normales, no tiene ningún sentido, ya que, se sobrescribe el fichero completo con el fichero local para tener la última versión. Otro campo que se ha añadido son los permisos que tiene el fichero original, el motivo principal para guardarlo es evitar problemas de permisos a la hora de sincronizar los datos en el cierre, dado que el programa no es el propietario del fichero, puede tener algún problema a la hora de escribir sobre él, por tanto, se cambian los permisos a unos más relajados, se devuelve la información al fichero local y, por último, se devuelven los permisos originales al fichero.

Con respecto a los cambios introducidos en las entradas de la tabla de descriptores de fichero, se añade el nombre original del fichero para poder acceder a su posición a la hora de sincronizarlo (incluyendo su ruta de directorios). El otro campo incluido es el modo de escritura, cuyo objetivo es conocer el modo en que ha sido abierto el fichero (normal, pre-caché o caché) para saber el tipo de cierre que se tiene que hacer (si se debe devolver a disco el fichero o no). Se ha incluido en el descriptor de fichero para evitar leer los metadatos en caso de un cierre normal.

En cuanto a funcionalidad, los principales cambios son, como es obvio, en la apertura y cierre del fichero, ya que, una vez que está escrito completo en la caché distribuida, se comporta como un fichero normal.

Sobre la apertura, además del cambio sustancial de tener que copiar todo el contenido del fichero, existen más cambios. El primer cambio es que se borra de la caché distribuida el fichero en caso de que ya exista, ya que, los datos que contiene muy probablemente habrán caducado. El segundo es referente a los metadatos, usualmente, cuando se abre un fichero que no existe en el sistema de ficheros distribuidos, éste es creado con las marcas de tiempo del momento actual, sin embargo, en el caso de un fichero de pre-caché, todos los metadatos que existen en un fichero normal y en el sistema distribuido, se copian tal cual, para mantener sus propiedades intactas (es una réplica del fichero local, no un fichero realmente nuevo).

Las novedades en el cierre de ficheros son menos importantes, tan solo se añade al cierre normal la copia del fichero del sistema distribuido al dispositivo de almacenamiento local (datos y metadatos que puedan ser copiados).

Para evitar problemas con los permisos del fichero local, cada vez que se trabaja con él se cambian a "0640", exactamente los mismos permisos que tiene por defecto un fichero en el momento de ser creado al hacer "open" con las opciones "`O_RDWR | O_CREAT`". El significado de "0640" es que el creador puede leer y escribir, mientras que el grupo sólo lo puede leer. Cuando se acaba de trabajar con él, se cambian sus permisos a los que tenía anteriormente (guardados en los metadatos del fichero de pre-caché/caché). En caso de intentar hacerse pre-caché de un fichero que no existe en el sistema local, se creará automáticamente con los permisos "0640".

La decisión de utilizar *write-on-close* de nuevo es, en parte, fruto de tratarse de un sistema experimental. Por un lado, es imposible saber cuándo *Memcached* expulsa un bloque de la caché distribuida, por tanto, ese bloque se perderá para siempre sin posibilidad de devolverlo al dispositivo local antes de ser expulsado imposibilitando el modo *write-back*, a menos que se modifique el código fuente de *Memcached*. Con respecto a *delayed-write* (escrituras retrasadas) pueden suponer un lastre en cuanto a rendimiento, a pesar de que pueden resultar muy interesantes de cara a evitar pérdidas de datos.

Por último, las funciones de borrado y renombrado de ficheros del sistema de ficheros distribuido, dejan de tener sentido (aunque pueden seguir utilizándose), ya que, el objetivo del fichero es hacer de caché, no trabajar con él como un fichero. Aún así, pueden utilizarse tomando las medidas adecuadas, ya que, para permitir la apertura de ficheros que no se encuentren en el directorio actual, es posible abrir ficheros locales con nombres como "/dir1/dir2/fichero", que serán guardados en la caché distribuida como "fichero" eliminando la ruta para evitar problemas de tamaño en el nombre de fichero (el nombre de los ficheros en caché distribuida tiene un máximo de 249 caracteres).

5.3.2 Modo caché

Este modo es una evolución del modo anterior, algo más complejo tanto en su concepción como en su implementación. El objetivo es asemejar su funcionamiento a una verdadera caché de un sistema de ficheros y mejorar sustancialmente el rendimiento.

En primer lugar, deja de existir la latencia en el caso de las aperturas, ya que, no es necesario copiar todo el contenido del fichero, tan solo crearlo vacío. Otra ventaja es que sólo se llevan a caché aquellos bloques de datos que se leen/escriben, de modo que se minimiza el uso de la red de datos. Como última ventaja, en el cierre tan solo es necesario sincronizar con el disco local aquellos bloques que hayan sido modificados (ni siquiera los que han sido llevados a caché distribuida por una lectura), de modo que, de nuevo se mejora el rendimiento: el cerrado será más rápido en la mayoría de los casos, así como el uso de la red. Como contrapartida, en caso de modificarse todos los bloques, tardarán más en ser llevados a disco, ya que, en lugar de utilizarse la función para lecturas múltiples, se utiliza el "get" normal, pero es previsible que este inconveniente sólo aparezca en casos extremos.

Para lograr este modo de caché, el funcionamiento del sistema de ficheros distribuido cambia de manera sustancial, conservándose intacta únicamente la funcionalidad de gestión de bloques (lecturas/escrituras de bloques de datos y lecturas/escrituras de estados).

Para comenzar, los bloques de metadatos dejan de utilizar como clave el nombre de fichero para utilizar un identificador. Este identificador es muy similar al utilizado para los ficheros normales, sin embargo, añade a la fórmula el nombre del cliente (`host_name`) que está cacheando el fichero quedando:

```
SHA2-512( file_name + create_time(sec) + create_time(millisec) +
                                                    host_name)
```

La motivación de este cambio es que deja de tener sentido poder acceder a los datos alojados en caché como si de un fichero se tratase. El único objetivo de la caché es mantener los bloques necesarios en caché distribuida mientras el fichero local esté abierto en modo caché, y que sean lo más fácil y rápido de acceder posible. Por tanto, al tratarse de unos datos completamente temporales, sus metadatos se alojan en un bloque con un identificador que dependa, tanto del momento en que fue abierto (para poder volver a abrirlo en modo caché en otro momento y no confundir los datos) como del cliente que lo ha ejecutado (es perfectamente posible que dos clientes distintos intenten utilizar la caché con ficheros con el mismo nombre en sus respectivas máquinas).

Además, para evitar que un fichero del sistema de ficheros distribuido coincidiese en nombre con uno de estos identificadores, un caso que podría darse, los metadatos se guardarán con la clave "`>`"+`<identificador_caché>`. Siendo destacable recordar que el símbolo '>' está prohibido para los nombres de ficheros del sistema

planteado. En el caso de las claves de los bloques, seguirán el mismo esquema que en los ficheros normales, pero añadiendo el símbolo especial al principio, quedando:

'>'+<id_caché>+'>'+<num_bloque>

Para que la gestión de bloques siga funcionando exactamente igual, en el campo "file_name" o nombre de fichero del descriptor de fichero, se debe escribir este identificador, de modo que cuando vaya a buscar el bloque de metadatos no lo busque por su nombre de fichero, sino con el identificador que es la clave asociada a dicho bloque.

Hay más diferencias con respecto a la apertura normal de ficheros, pese a aprovecharse todos los cambios realizados en las estructuras para el modo pre-caché, en este caso su uso es aún más restrictivo. Si un fichero ha sido abierto en modo caché, no podrá abrirse en ningún otro modo. Lógicamente, no tiene sentido acceder en ningún otro modo, ya que, no tiene porqué estar completo, la caché de un fichero de varios bloques puede ser un solo bloque, aquel que se haya utilizado en la sesión. Por último, cabe destacar que, al igual que en modo pre-caché, se copian todos los metadatos posibles del fichero local. En este caso es especialmente importante, ya que, el tamaño del fichero será virtual (no todos los bloques del fichero tienen porqué estar en caché) y será de vital importancia que el tamaño esté bien gestionado.

El otro cambio sustancial consiste en el cacheo de ficheros como tal. Para conseguirlo, el único cambio introducido en las estructuras con respecto al estado en que habían quedado tras el diseño del modo pre-caché, es la inclusión de un puntero en cada entrada de la tabla de descriptores de fichero. Dicho puntero, hace referencia a un array de caracteres que se reservará cuando se abra un fichero en modo caché. La función de este array es hacer de tabla de estado de la caché, indicando en todo momento los bloques del fichero abierto en modo caché que se encuentran en la caché distribuida. El array tendrá tantas posiciones como bloques tenga el fichero (la posición 0 tendrá el estado del bloque 1, la 1 del bloque 2, etc.) y sus valores podrán ser '-' cuando el bloque no ha sido traído a caché, 'r' cuando ha sido traído para realizar una lectura, 'w' cuando ha sido traído para realizar una escritura y 'b' cuando fue traído para realizar una lectura y posteriormente fue modificado en una escritura o viceversa.

La implementación de la tabla también conlleva una decisión de diseño importante, el lugar en que se almacenará. En principio se planteó realizar la tabla en caché distribuida para que cualquier cliente pudiese conocer el estado del fichero cacheado, sin embargo, su complejidad era altísima (habría que dividirla en bloques, acceder a todos los bloques para recorrerla y solo a los bloques necesarios para modificarla, etc.) Sin embargo, se observó que no aportaba ninguna ventaja, es inútil que todos los clientes puedan conocer el estado del fichero cuando se trata de un fichero completamente temporal y que solo interesa al cliente que ha abierto el fichero local y solo durante el tiempo que está abierto. Por este motivo se decidió mantener la tabla en

memoria principal del cliente, y el mejor lugar era la tabla de descriptores de fichero, ya que, cada sesión con un fichero tiene su propia entrada. Para no penalizar el uso normal del sistema de ficheros distribuido, en la entrada de la tabla de descriptores de fichero solo se encuentra un puntero a la tabla de caché, para evitar ocupar espacio innecesario en el resto de los modos de empleo.

Para facilitar el uso de la tabla, han tenido que ser modificados los métodos de lectura y escritura, eso sí, de forma totalmente transparente al usuario, que sólo verá diferencias en la interfaz de apertura del fichero, a partir de entonces utilizará las mismas funciones y de igual forma, independientemente del modo en el que esté abierto el fichero.

Las modificaciones realizadas en los métodos de lectura y escritura, permiten adaptar el funcionamiento de forma muy sencilla. En el momento en que comienza la función, si se está trabajando en modo caché, se calculan los bloques del sistema de ficheros distribuido que serían utilizados gracias al *offset* del descriptor de fichero, el tamaño de bloque y el tamaño de la lectura/escritura introducido por parámetro. Por ejemplo, con un tamaño de bloque de 1 Kbyte, en el *offset* 0 y una lectura de 2048 bytes, se utilizarían los bloques 1 y 2 del fichero distribuido. Acto seguido, se comprueban las posiciones 0 y 1 de la tabla de caché y se copian a la caché distribuida todos los bloques marcados como '-' (el resto de bloques, 'r', 'w' o 'b', tendrán una información igual o más actualizada que la local). Acto seguido, se marcan los bloques según su estado: en una lectura los '-' pasan a ser 'r' y los 'w' pasan a ser 'b' (los 'r' y los 'b' se quedan como están) mientras que en una escritura los '-' pasan a ser 'w' y los 'r' pasan a ser 'b' (dejando 'b' y 'w' igual). Es también en este momento cuando la tabla es ampliada de forma dinámica en el caso de ser necesario, de modo que puede crecer sin más límite que la cantidad de memoria principal del cliente.

A partir de este momento, la lectura/escritura, se desarrolla con total normalidad, ya que, todos los datos necesarios para la lectura/escritura se encuentran en la caché distribuida y la tabla ya ha sido modificada.

La última modificación es en la función de cierre y es muy similar al caso de pre-caché. Se consulta en el descriptor de fichero el modo de apertura, y en el caso de estar abierto en modo caché, se sincroniza el contenido de la caché con el disco local (*write-on-close*). Eso sí, aporta una ventaja sustancial, y es que sólo se sincronizan con el dispositivo de almacenamiento local aquellos bloques que han sido modificados en la sesión. Es decir, se recorre la lista de caché que se encuentra en el descriptor de fichero y solo se copian a disco aquellos bloques que estén marcados como 'w' o 'b' en dicha tabla, minimizando con ello el uso de la red de comunicaciones y el tiempo de cerrado.

6 IMPLEMENTACIÓN

Dado que se trata de una librería experimental, el código fuente está fuertemente documentado para que sea fácilmente comprensible por cualquier desarrollador, ya sea para utilizarlo o para ampliarlo.

La documentación del código ha sido realizada íntegramente en inglés para ampliar al máximo el público que la pueda comprender y siguiendo la notación *javadoc*. Esta última decisión puede resultar en principio extraña de cara a la documentación de un código escrito en lenguaje C, sin embargo, hay un motivo de peso: la utilización de Doxygen como generador de documentación. Entre las opciones que ofrece Doxygen, se ha utilizado *javadoc* por ser la más moderna (muy utilizada por programadores actualmente) y por ser la más clara para principiantes si no se conoce ninguna de las opciones que ofrece. No es necesario tener una amplia base de *javadoc* para comprender la estructura de los comentarios.

Gracias a Doxygen, se puede generar automáticamente a partir de los comentarios del código documentación en muchos formatos. En la carpeta docs del código se incluirá la documentación en HTML (posiblemente el formato más cómodo y potente), RTF, LaTeX listo para ser compilado en PDF y, la más habitual en Linux, la documentación en formato `man`.

Para mejorar la comprensión del funcionamiento de la librería, también se ha incluido un modo "*debug*" que imprime por la salida estándar la evolución de la ejecución, dando detalles de las variables que se están utilizando.

Este modo "*debug*" se ha desarrollado mediante fragmentos con la etiqueta `#ifdef debug`, de modo que si al principio del código existe una línea `#define debug`, el código actuará en modo "*debug*", mientras que si esta línea no está presente, no imprimirá por pantalla absolutamente nada (salvo algunos errores).

Además de la documentación, se han incluido varias carpetas con código:

- **`libs`**: incluye todas las librerías necesarias para el correcto funcionamiento del código, por ejemplo, la implementación del algoritmo SHA2 de Aaron D. Gifford[16].
- **`stats`**: código implementado para llevar a cabo la evaluación de rendimiento y extraer los datos para la realización de las estadísticas.
- **`tests`**: código implementado para realizar la verificación del correcto funcionamiento del código, para comprobar que todo funciona como se espera.
- **`client examples`**: código de ejemplo de algunos clientes sencillos implementados utilizando la librería del sistema de ficheros distribuido

desarrollada. Son ejemplos sencillos de código para que el usuario comprenda mejor el funcionamiento del sistema y el orden de ejecución de funciones, así como, su sintaxis.

6.1.1 memcachedfs.c y memcachedfs.h

Este módulo incluye todas las funciones de la librería necesarias para que el usuario interactúe con el sistema de ficheros distribuido. Los pasos a seguir son los siguientes:

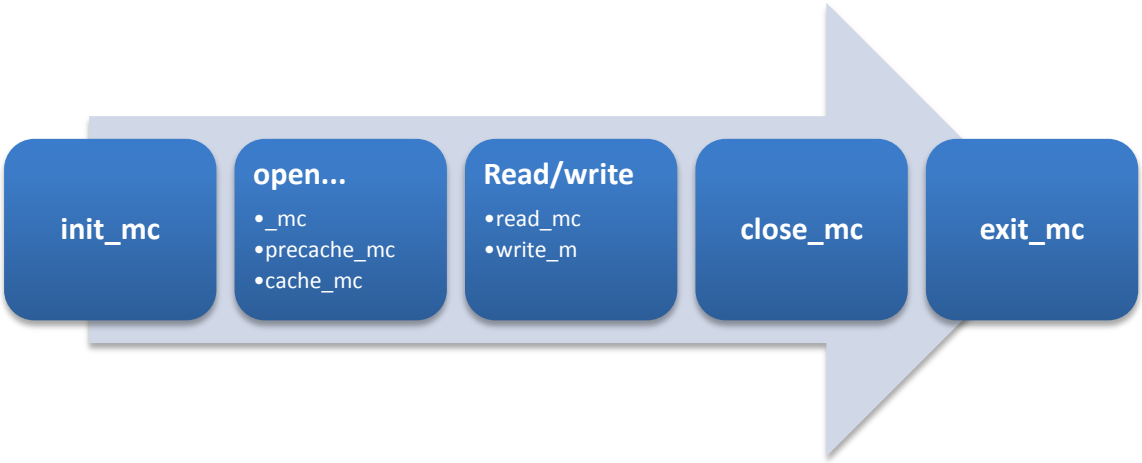

Figura 8: Secuencia de interacción con el sistema de ficheros distribuido (MemcachedFS)

Una vez inicializados los distintos servidores de *Memcached* y conociendo su dirección IP y puerto de escucha, se procede a la puesta en marcha del sistema, es decir, la conexión con *Memcached* y la inicialización de la tabla de descriptores de fichero. Esto se hace mediante el método **`init_mc(char* servers_string)`** que recibe como parámetro el puntero a una cadena de caracteres que contiene los servidores y los puertos siguiendo la sintaxis:

```
<mc_server_ip>:<mc_server_puerto>[,<mc_server_ip>:<mc_server_puerto>]
```

Es decir, las direcciones y los puertos de escucha de los servidores de *Memcached* separados por comas, que podrán ser de 1 a *n*. Esta inicialización debe hacerse en cada cliente, cada vez que se inicie una conexión con los servidores de *Memcached*, no vale con hacerla cuando se inicializa por primera vez el sistema de ficheros. Sería una especie de equivalente al 'mount' más que al 'format' de un sistema de ficheros convencional, y es de vital importancia que la lista de servidores en todos los clientes sea exactamente la misma, tanto en contenido como en orden de los servidores, para mantener la coherencia de los datos. De lo contrario, la función de *hash* podría enviar bloques con claves iguales a servidores distintos.

Una vez realizada esta puesta en marcha, se inicializa la variable global `is_initialized` a `TRUE` (valor 1) y es posible utilizar todos los métodos referentes a la gestión de ficheros de MemcachedFS. Se trabajará con ellos exactamente igual que si se encontrasen en un dispositivo de almacenamiento local, con la única diferencia del nombre de las funciones a las que se le ha añadido el sufijo "`_mc`" (en referencia a *Memcached*) para diferenciarlas de las funciones locales ofrecidas por la entrada y salida estándar de POSIX.

Con respecto al *feedback* ofrecido a los usuarios en referencia a los errores que se puedan producir durante la ejecución de la función, en lugar de utilizar `errno` como ocurre en la librería estándar de POSIX, se devuelve directamente el error en el valor de retorno, mediante un entero negativo. Esta diferencia de comportamiento se produce en toda la librería, no solo en esta función. Los errores que puede devolver esta función son:

Código	Descripción
-1	Lista incorrecta de servidores: el formato en el que han sido introducidos los servidores no es correcto o no se han encontrado.
-2	Tamaño de bloque incorrecto. El tamaño de bloque en "utils.h" debe ser un valor entre 1 y 1023.

Tabla 30: Códigos de error de la función init_mc

El uso más habitual de la librería será open + read/write + close, pero a continuación se detallan todas las funciones implementadas:

`int open_mc(char *file_name, int flags)`: al igual que la interfaz POSIX, el método que abre los ficheros recibe por parámetro el nombre del fichero a abrir y las opciones de apertura (*flags*) y devuelve un descriptor de fichero, que identificará la entrada de la tabla en la que se encuentra la información de las operaciones sobre dicho fichero (modo de apertura, *offset*, etc.), en caso de error se devolverá un número negativo. Como peculiaridades con respecto al uso de su equivalente POSIX cabe destacar dos detalles: el primero es en referencia al nombre de fichero, el sistema de ficheros diseñado no soporta directorios, por tanto, si se quiere jerarquizar el sistema deberá ser de forma completamente autónoma (sin apoyo de la librería) utilizando siempre rutas absolutas. Como segundo detalle, dado que no se ha implementado un sistema de permisos excesivamente complejo, las únicas opciones de apertura (*flags*) que son admitidas por la función son:

- `O_RDONLY`: abre el fichero en modo de solo lectura.
- `O_WRONLY`: abre el fichero en modo de solo escritura.
- `O_RDWR`: abre el fichero en modo de lectura y escritura.

Es importante destacar que el modo de apertura tiene prioridad sobre cualquier otro tipo de especificación de permisos, es decir, dado que el sistema de permisos es muy débil, no se tiene en cuenta si el fichero es de solo lectura para poder abrirlo en un

modo de escritura. El principal objetivo del control de permisos en la apertura es evitar modificar un fichero por error mediante su apertura en modo de solo lectura. Con respecto a las *flags* referentes a la creación de ficheros, dado que no están soportadas, el funcionamiento será el predeterminado de POSIX, es decir, se abrirá el fichero siempre que exista, y en caso contrario, se creará un fichero vacío con el nombre introducido por parámetro. Cuando se abre un fichero, se actualiza su marca de tiempo de acceso (`atime`) en caso de que ya existiese.

No se permite abrir un fichero que en el momento de su creación tuviese como tamaño de bloque un valor distinto del actual de MemcachedFS. Esto se puede saber porque entre los metadatos del fichero se incluye el tamaño de bloque en el momento de su creación.

En la siguiente tabla se describen los errores que pueden ser devueltos por la función, en lugar del descriptor de fichero:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	No queda ningún descriptor de fichero libre.
-3	Carácter no válido en el nombre de fichero. No se pueden usar nombres que contengan el carácter '>'.
-4	Modo de acceso no permitido. Los modos de acceso permitidos son `O_RDONLY`, `O_WRONLY` u `O_RDWR`.
-5	Error escribiendo en la caché distribuida.
-6	Longitud del nombre de fichero excesiva. El máximo son 249 caracteres.
-7	El tamaño de bloque actual del sistema de ficheros distribuido y el tamaño de bloque con el que fue creado el fichero, no coinciden.
-8	El fichero fue abierto en modo "caché", no compatible con modo normal.

Tabla 31: Códigos de error de la función open_mc

`int open_precache_mc(char *file_name, int flags)`: segundo modo de apertura, en el que se utiliza la caché distribuida como caché de un sistema de ficheros local. Este cambio hace que la funcionalidad principal de apertura de fichero cambie. El primer cambio fundamental es el parámetro `file_name`, que en este caso se refiere a un fichero local del sistema en el que se esté ejecutando el cliente. El funcionamiento de apertura es muy similar al anterior, de cara a efectos externos (devuelve un descriptor de fichero o un número negativo en caso de error, las limitaciones de las opciones de apertura y de los permisos son las mismas, etc.) las principales diferencias son a nivel interno: el nombre del fichero se filtra, de modo que en remoto se le quita toda la ruta relativa (desde el principio hasta la última barra que se encuentre) y se guarda el nombre y permisos originales para poder restaurarlo en el cierre. Además, se copia todo el fichero local a la caché distribuida manteniendo sus atributos (metadatos) y borrando previamente el fichero con dicho nombre en caso de existir. A partir de este momento el fichero queda marcado internamente, y de cara al usuario, la administración del fichero se hace exactamente igual que con una apertura normal, como si fuese un fichero de

MemcachedFS, a través de su descriptor de fichero. Puede utilizarse cualquiera de los métodos de administración de fichero (teniendo siempre en cuenta el filtrado en el nombre de fichero a la hora de borrarlo o renombrarlo).

En la siguiente tabla se describen los errores que pueden ser devueltos por la función, en lugar del descriptor de fichero:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	Imposible abrir el fichero de origen (fichero local).
-3	Imposible abrir el fichero de destino (fichero remoto).
-4	Imposible leer el estado (metadatos) del fichero local.
-5	Error cerrando el fichero de origen (fichero local).

Tabla 32: Códigos de error de la función open_precache_mc

`int open_cache_mc(char *file_name, int flags)`: tercer modo de apertura, también para utilizar la caché distribuida como caché de un sistema de ficheros local. En este caso, también se aporta la ruta de un fichero local. Sin embargo, no se copia el fichero completo a disco, sólo se llevan los bloques a caché distribuida bajo demanda, es decir, cuando se requiere y siempre con identificadores temporales. Por este hecho, los ficheros remotos no pueden ser reabiertos una vez cerrados ni se pueden borrar o renombrar. Sin embargo, durante una misma sesión (desde la apertura hasta el cierre) se comporta de cara al usuario igual que un fichero distribuido normal para leer o escribir, mover el puntero (*offset*) y para cerrarlo. No existe absolutamente ninguna restricción en el nombre de los ficheros que se pueden abrir, ya que, en el sistema de ficheros distribuido son guardados con un identificador temporal, generado a partir del nombre de fichero entre otros datos, pero sin implicar ninguna restricción, ni de longitud ni de caracteres especiales. El identificador cambia según el instante de tiempo en que se abra y la máquina desde la que se abra, es decir, siempre se crea un nuevo fichero, nunca un mismo fichero abierto dos veces se guardará en el mismo fichero temporal en la caché distribuida.

En la siguiente tabla se describen los errores que pueden ser devueltos por la función, en lugar del descriptor de fichero:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	No hay descriptores de fichero libres.
-3	Modo de acceso no permitido. Los modos de acceso permitidos son `O_RDONLY`, `O_WRONLY` u `O_RDWR`.
-4	Imposible leer el estado (metadatos) del fichero local.
-5	Imposible actualizar el estado (metadatos) del fichero distribuido.

Tabla 33: Códigos de error de la función open_cache_mc

Funciones que requieren que el fichero esté abierto

`ssize_t write_mc(int fd, const void *buf, size_t count)`: tal y como ocurre en POSIX, se introduce por parámetro el descriptor de fichero, un puntero a un buffer de datos y el tamaño de dicho buffer y el contenido del buffer será escrito en el fichero al que corresponda el descriptor, a partir del lugar al que apunte el *offset* del descriptor. El valor devuelto será el número de bytes que han podido ser escritos realmente o un número negativo en caso de error. Una vez finalizada la escritura, el *offset* queda apuntando al byte siguiente al último byte escrito. El funcionamiento de la escritura es igual a POSIX, si la zona está escrita previamente se sobrescribe y si se sale del tamaño de fichero, se amplía (escribiéndose valores nulos en caso de que exista un espacio entre el último byte del fichero y el primero a escribir). Además, se actualizan todas las marcas de tiempo (`atime` – acceso, `ctime` – cambio y `mtime` – modificación). Es en este punto donde se compone la clave de los bloques de datos con <id>+'>'+<num_bloque>, calculándose el número de bloque como el *offset* dividido entre el tamaño de bloque del sistema de ficheros distribuido.

En este caso, la única funcionalidad excepcional que requieren los modos de caché se centra en el modo caché, que requiere llevar a caché distribuida aquellos bloques que no están aún y que van a ser escritos y marcarlos como escritos o como 'b' (leído y escrito) en caso de que hubiesen sido leídos con anterioridad. Para ello, tan sólo hay que calcular los bloques de datos que ocuparía en caché la porción de datos que quiere ser escrita según el tamaño de la escritura y el *offset* actual y consultar/modificar la tabla que se encuentra en la entrada del descriptor del fichero a escribir.

Los errores que puede devolver esta función son los siguientes:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	El descriptor de fichero no apunta a un fichero abierto.
-3	El archivo fue abierto en modo de solo lectura (`O_RDONLY`).
-4	Imposible leer el estado (metadatos) del fichero remoto. El fichero no existe o hay un problema de conexión con la caché distribuida.
-5	Error leyendo desde o escribiendo en la caché distribuida.
-6	No se puede abrir el fichero local para cachearlo.
-7	Error leyendo desde el fichero local para cachearlo.

Tabla 34: Códigos de error de la función write_mc

`ssize_t read_mc(int fd, void *buf, size_t count)`: la lectura también funciona igual que en el estándar de entrada/salida de POSIX. Como parámetros de entrada se tienen: el descriptor de fichero, el tamaño de la lectura y el puntero a un *buffer* previamente reservado en memoria con el tamaño a leer (si no está reservado, no se puede garantizar el correcto funcionamiento). Lo que hace es leer el número de bytes pasados por parámetro del fichero correspondiente al descriptor de fichero, empezando por el byte al que apunta el *offset* del descriptor, y copiarlos al buffer apuntado por el parámetro. Como resultado de la operación devuelve el número de bytes leídos o un

número negativo en caso de error. Cabe destacar que pueden ser leídos menos bytes de los solicitados sin que medie una situación anormal, por ejemplo, cuando se intenta leer más allá del final del fichero.

La funcionalidad adicional requerida para el modo caché es muy similar a la anterior. Es necesario "volcar" del disco local a caché distribuida aquellos bloques que no se encuentren en los servidores de *Memcached* (marcados en la tabla de caché del descriptor de fichero como '-') y marcar como leídos o 'b' (escrito y leído) aquellos bloques que se lean.

Los errores que puede devolver esta función son los siguientes:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	El descriptor de fichero no apunta a un fichero abierto.
-3	El archivo fue abierto en modo de solo escritura (`O_WRONLY`).
-4	Imposible leer el estado (metadatos) del fichero remoto. El fichero no existe o hay un problema de conexión con la caché distribuida.
-5	Resultado de una lectura de caché distribuida no esperado.
-6	Offset mayor que el tamaño de fichero.
-7	No se puede abrir el fichero local para cachearlo.
-8	Error leyendo desde el fichero local para cachearlo.

Tabla 35: Códigos de error de la función read_mc (y read_mc_mult)

`ssize_t read_mc_mult(int fd, void *buf, size_t count)`: exactamente el mismo funcionamiento externo que `ssize_t read_mc`, la principal diferencia radica en el uso interno de la funciones de "`utils.c`", ya que, se aprovechan las lecturas múltiples simultáneas que permite *Memcached* para mejorar el rendimiento. Si hay que leer más de un bloque completo de *Memcached* se le solicitan todos a la vez para ahorrar en comunicaciones, en caso contrario, funciona exactamente igual que `read_mc`. Se han incluido las dos funciones para poder comparar su rendimiento. Los errores que podría devolver esta función coinciden exactamente con la anterior.

`off_t lseek_mc(int fd, off_t offset, int whence)`: el funcionamiento es exactamente igual al que tiene en POSIX. Se introduce por parámetro el descriptor de fichero cuyo *offset* quiere ser modificado, el *offset* en bytes y el modo a utilizar, existiendo los siguientes tres:

- `SEEK_SET`: sitúa el puntero en el byte que indica el *offset* introducido por parámetro.
- `SEEK_CUR`: sitúa el puntero en la posición resultado de la suma del *offset* actual y el introducido por parámetro, es decir, adelanta el puntero a partir de la posición actual tantos bytes como indique el *offset* introducido por parámetro.

- `SEEK_END`: adelanta el puntero tantos bytes como indique el *offset* introducido por parámetro a partir del final del fichero.

Como resultado, devuelve la posición actual del puntero, es decir, el número de bytes contando desde el principio del fichero en que se encuentra tras el reposicionamiento. Cabe destacar, que esta función no hace ninguna comprobación salvo la de si el *offset* final es negativo, es decir, el puntero se puede colocar fuera del límite del fichero. El tamaño del fichero no se verá modificado hasta que no se realice una escritura fuera de dicho límite (escribiendo ceros en caso de que haya un hueco entre el final de fichero actual y el inicio de la escritura). En caso de que se realice una lectura tras el posicionamiento del puntero fuera de los límites del fichero, será la función de lectura la encargada de devolver un error.

Los errores que se pueden producir en la ejecución de esta función se identificarán con los siguientes códigos:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	El descriptor de fichero no apunta a un fichero abierto.
-3	Modo de desplazamiento incorrecto (`whence`). Los modos soportados son: `SEEK_SET`, `SEEK_CUR` y `SEEK_END`.
-4	Imposible leer el estado (metadatos) del fichero remoto. El fichero no existe o hay un problema de conexión con la caché distribuida.
-5	El *offset* final es un valor negativo.

Tabla 36: Códigos de error de la función lseek_mc

`int close_mc(int fd)`: se introduce por parámetro el descriptor del fichero a cerrar y se recibe como resultado un cero en caso de que todo haya ido como se esperaba o un número negativo en caso de error. Tras comprobar que el descriptor de fichero pertenece realmente a un fichero abierto, se comprueba el modo de apertura (normal, pre-caché o caché) y se actúa en consecuencia. En **modo normal**, se reinicia el contenido del descriptor de fichero para marcarlo como libre, es decir, se pone a -1 el *offset*, el nombre del fichero se cambia a '\0' (indicativo principal utilizado para saber si el descriptor apunta a un fichero abierto o no) y los modos de apertura y permisos se ponen a `NOT_USED`.

En el **modo pre-caché**, además de reiniciar el descriptor de fichero como en el modo normal, se hacen una serie de comprobaciones y acciones adicionales. En primer lugar se comprueba que está guardado el nombre del fichero original (el que tenía en local, con su ruta completa), se borra dicho fichero local, y se copia el fichero que se encuentra en caché distribuida (actualizado) al disco. Una vez copiado, se le devuelven sus permisos originales y se reinician los campos del descriptor de fichero utilizados sólo por los modos de caché.

En el **modo caché**, también se reinicia el descriptor de fichero como en el modo normal, y las comprobaciones adicionales son las mismas que en el modo pre-caché. Sin embargo, en el caso de caché, no se borra el fichero local, ya que, hay datos que no tienen porqué haberse llevado a la caché distribuida. Se recorre la tabla de caché (que se encuentra en el descriptor de fichero) y se copian a disco local todos aquellos y solo aquellos bloques marcados como 'w' (escrito) o 'b' (leído y escrito), es decir, los que han sido modificados, cuyos datos son más actuales que los locales. No es necesario devolver a disco los bloques que están marcados como leídos, puesto que sus datos deberían ser exactamente los mismos que los locales que se leyeron. Por último, se reinician los campos utilizados por el modo caché, es decir, el nombre de fichero original y se libera (*free*) la tabla de caché.

Es importante destacar, que queda sujeto a la buena utilización del usuario que no se modifique de ningún modo el fichero local mientras está abierto en caché distribuida, en caso contrario, no se puede garantizar la integridad de los datos.

Los errores que se pueden producir en la ejecución de esta función se identificarán con los siguientes códigos:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	El descriptor de fichero no apunta a un fichero abierto.
	Errores modo pre-caché
-3	El fichero no está abierto en modo pre-caché.
-4	No es posible abrir fichero de destino (fichero local).
-5	No es posible leer el estado (metadatos) del fichero de origen (fichero distribuido).
-6	Error cambiando los permisos del fichero de destino (fichero local).
-7	Error cerrando el fichero de destino (fichero local).
-8	Error cambiando los permisos del fichero local.
	Errores en modo caché
-13	El fichero no está abierto en modo caché.
-14	No es posible abrir fichero de destino (fichero local).
-15	No es posible leer el estado (metadatos) del fichero de origen (fichero distribuido).
-16	Error actualizando (escribiendo) el fichero de local.
-17	Error cerrando el fichero de destino (fichero local).

Tabla 37: Códigos de error de la función close_mc

Funciones que no requieren que el fichero esté abierto

`int unlink_mc(char *file_name)`: se elimina de la caché distribuida el fichero cuyo nombre coincide con el introducido por parámetro. Devuelve cero en caso de que la acción se produzca correctamente y un número negativo en caso contrario. Se trata de un borrado lógico, es decir, sólo se borra el bloque de metadatos que tiene el nombre de fichero introducido por parámetro. Cabe desatacar, que dicho nombre de fichero debe ser filtrado en modo pre-caché (se le quita toda la ruta relativa, es decir, todo aquello por detrás de la última barra de separación de directorio) y en modo caché

no tiene sentido borrar el fichero, ya que, si se borra en mitad de una sesión se pierde la coherencia de los datos, por tanto, no está permitido su borrado (además, el nombre de fichero es un identificador temporal difícilmente calculable por el usuario).

Los errores que puede devolver esta función son los siguientes:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	Carácter no válido en el nombre de fichero. No se pueden usar nombres que contengan el carácter '>'.
-3	Longitud del nombre de fichero excesiva. El máximo son 249 caracteres.
-4	Error borrando el fichero de la caché distribuida.

Tabla 38: Códigos de error de la función unlink_mc

`int rename_mc(char *old_file_name, char *new_file_name)`: renombra el fichero cuyo nombre coincide con `old_file_name` con el nombre `new_file_name`. Devuelve cero en caso de que la operación se realice correctamente y un número negativo en caso contrario. Cabe desatacar que el nombre de fichero que se quiere cambiar debe ser filtrado en modo pre-caché (se le quita toda la ruta relativa, es decir, todo aquello por detrás de la última barra de separación de directorio) y en modo caché no tiene sentido renombrar el fichero, ya que, si se renombra en mitad de una sesión se pierde la coherencia de los datos, por tanto, no está permitido su renombrado (además, el nombre de fichero es un identificador temporal difícilmente calculable por el usuario).

Los errores que puede devolver esta función son los siguientes:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	Longitud del nuevo nombre de fichero excesiva. El máximo son 249 caracteres.
-3	Carácter no válido en el nuevo nombre de fichero. No se pueden usar nombres que contengan el carácter '>'.
-4	Longitud del nombre de fichero antiguo excesiva. El máximo son 249 caracteres.
-5	Carácter no válido en el nombre de fichero antiguo. No se pueden usar nombres que contengan el carácter '>'.
-6	Error leyendo el fichero antiguo de la caché distribuida.
-7	Error escribiendo el estado (metadatos) del fichero en la caché distribuida.
-8	Error borrando el estado (metadatos) del fichero antiguo de la caché distribuida.

Tabla 39: Códigos de error de la función rename_mc

`int stat_mc(char * file_name, struct stat_mc *stat)`: recibe por parámetro el nombre del fichero del que desea obtener sus metadatos y el puntero a una estructura (reservada en memoria previamente) que se rellenará con los metadatos del fichero. La estructura stat_mc tiene las siguientes características:

```
struct stat_mc {
    char    id[ID_LENGTH+2];
    mode_t  mode;
    int     open_mode; /* FS / PRECACHE / CACHE */
    mode_t  orig_mode;
    uid_t   uid;
    gid_t   gid;
    off_t   size;
    unsigned int  block_size;
    time_t  atime;
    time_t  mtime;
    time_t  ctime;
}
```

Para más información sobre los metadatos de los ficheros consultar el apartado *5.2 Diseño del sistema MemcachedFS*.

Los códigos de los posibles errores que pueden suceder durante la ejecución de esta función son los siguientes:

Código	Descripción
-1	El sistema no está inicializado. La función `init_mc` debe haber sido ejecutada correctamente antes de cualquier operación de gestión de ficheros.
-2	Longitud del nombre de fichero excesiva. El máximo son 249 caracteres.
-3	Carácter no válido en el nombre de fichero. No se pueden usar nombres que contengan el carácter '>'.
-4	Error leyendo el estado (metadatos) del fichero de la caché distribuida.

Tabla 40: Códigos de error de la función stat_mc

`int cp_to_memcached(char* src_file_name, char* dst_file_name)`: se trata de una función auxiliar, pensada en principio para pruebas, pero que puede resultar útil al usuario final. Recibe como entrada por parámetro la ruta de un fichero local y un nombre de fichero remoto y, lo que hace, es copiar el fichero local en la caché distribuida con el nombre elegido. Se encarga de abrir ambos ficheros, copiarlo y cerrarlos, por lo tanto no devuelve ningún descriptor, devuelve cero en caso de que el resultado de la operación sea el esperado y un número negativo en caso de que ocurra cualquier tipo de error, siendo los siguientes los códigos para identificarlos:

Código	Descripción
-1	No es posible abrir el fichero de origen (fichero local).
-2	No es posible abrir el fichero de destino (fichero remoto).
-3	No es posible leer el estado (metadatos) del fichero local.
-4	Error cerrando el fichero de origen (fichero local).
-5	Error cerrando el fichero de destino (fichero remoto).

Tabla 41: Códigos de error de la función cp_to_memcached

`int cp_from_memcached(char* src_file_name, char* dst_file_name)`: se trata del funcionamiento contrario al anterior. Copia un fichero (`src_file_name`) desde la caché distribuida al disco local como `dst_file_name`. También abre ambos ficheros, copia el contenido a disco y los cierra. Devuelve cero si la operación se completa de forma satisfactoria y un número negativo en caso de error, siendo los siguientes los códigos para identificarlos:

Código	Descripción
-1	No es posible abrir el fichero de origen (fichero remoto).
-2	No es posible abrir el fichero de destino (fichero remoto).
-3	No es posible leer el estado (metadatos) del fichero remoto.
-4	Error cerrando el fichero de origen (fichero remoto).
-5	Error cerrando el fichero de destino (fichero local).

Tabla 42: Códigos de error de la función cp_from_memcached

`int exit_mc()`: cierra las conexiones con *Memcached* y libera todas las variables requeridas para mantener dicha conexión. También marca la variable global `is_initialized` como `FALSE`, de modo que no pueda usarse ninguna de las funciones que requieren comunicación con *Memcached* salvo la inicialización. De forma similar a la función `init_mc`, la lógica dentro de un sistema de ficheros habitual de esta función sería una especie de `unmount`, de desconexión con el dispositivo de almacenamiento, pero los datos seguirán siendo válidos en caso de realizarse una nueva conexión (si los servidores se mantienen activos y no se ha expulsado ninguno de los bloques de la caché por la política de reemplazo). Devuelve siempre cero, no puede suceder ningún error. Se mantiene el retorno de un entero para compatibilidad futura (posible implementación de errores con nueva funcionalidad).

Funciones auxiliares internas

Las siguientes dos funciones están en la capa externa de la librería porque son las encargadas de administrar la tabla de descriptores de fichero, para evitar problemas de acceso a variables externas al fichero. Se desaconseja el uso por parte de los usuarios.

`int init_file_desc_table()`: inicializa la tabla de descriptores de fichero. Marca todos los campos como no utilizados o con valores nulos o negativos (*reset* de todos los campos y de todas las entradas de la tabla).

`int search_free_fd()`: se recorre en orden la tabla de descriptores de fichero hasta encontrar una entrada libre (sin usar), se identifican porque el modo de apertura está marcado como `NOT_USED`. Se devuelve el índice de la entrada del descriptor de fichero si se encuentra uno libre o -1 en caso de que estén todos ocupados.

memcachedfs.h

Además de las cabeceras de las funciones, el fichero de cabecera incluye tres constantes.

`MAX_BLOCK_SIZE`: indica el tamaño máximo de bloque permitido por el sistema de ficheros distribuido. Su valor predeterminado es 1023 y no debe cambiarse, ya que, *Memcached* no acepta valores de 1024 Kbyte o superiores.

`NUM_FILE_DESC`: indica el número máximo de descriptores de fichero que tendrá el sistema de ficheros distribuido, es decir, el número de ficheros que pueden estar abiertos de forma simultánea. Tiene como valor predeterminado 128, pero puede cambiarse si es necesario.

`NOT_USED`: se usa para indicar que un campo no está siendo utilizado. Su valor predeterminado es -1 y no debe cambiarse para evitar conflictos con otros valores que pudieran tomar dichos campos.

También se hace un "include" de las librerías `utils.h` y `fcntl.h` para la gestión de ficheros.

6.1.2 utils.c y utils.h

En esta capa se encuentran todas las funciones auxiliares internas necesarias utilizadas por la capa externa, así como todas las funciones que requieren llamadas a la librería libMemcached. En ningún caso se aconseja que el usuario utilice ninguna de estas funciones, la finalidad de documentarlas es que se comprenda mejor el funcionamiento del sistema.

Como están pensadas para uso interno, se obvian muchas comprobaciones, ya que, se realizan en el nivel de `memcachedfs`, por ejemplo, la mayoría de las funciones deben ser llamadas con una conexión con *Memcached* abierta y esto no se comprueba, tampoco se comprueba que las claves tengan un formato válido, etc.

`int generate_id(char *file_name, char final_id[ID_LENGTH+2])`: la funcionalidad principal es crear los identificadores de los bloques de datos de cada fichero. Para ello, recibe por parámetro el nombre de fichero y un array de caracteres en el que guardar el identificador generado. Mediante la función `ftime` se consigue el tiempo actual y se concatenan al nombre del fichero el tiempo actual en segundos y milisegundos. A este conjunto de datos se le aplica el algoritmo de *hash* SHA-512 para unificar el tamaño de los identificadores evitando colisiones. Se añaden las marcas de tiempo para evitar problemas con datos viejos en caché a la hora de crear un nuevo fichero con el mismo nombre.

`int generate_cache_id(char *file_name, char final_id[ID_LENGTH+2])`: tanto el objetivo como la funcionalidad es muy similar a la anterior. En este caso, los identificadores se generan para el modo caché y no sólo identifican los bloques de datos, sino también los bloques de metadatos, por lo que es especialmente importante evitar colisiones no sólo de tiempo, sino también entre máquinas (para permitir que dos máquinas puedan abrir de forma simultánea dos ficheros que se llamen igual). Para conseguirlo, se añade a la concatenación el nombre de la máquina que está ejecutando el cliente mediante la función `gethostname`, o `gethostid` si ésta fallase. Acto seguido, se aplica SHA-512 como en el caso anterior para mantener la uniformidad en la longitud del identificador.

`int read_stat(char *file_name, struct stat_mc *stat)`: lee de *Memcached* un bloque de metadatos, para ello recibe un nombre de fichero (es decir, la clave del bloque de metadatos a recuperar) y el puntero a una estructura `stat_mc` previamente reservada en memoria. El funcionamiento consiste en solicitar a *Memcached* el bloque cuya clave coincide con `file_name` a través de libMemcahed (utilizando `memcached_get`). Como resultado se devuelve cero si la acción termina de forma correcta o -1 si no se consigue extraer de *Memcached* dicho bloque.

`int write_stat(char *file_name, struct stat_mc stat)`: se trata de la función complementaria a la anterior, escribe en *Memcached* a través de libMemcached (`memcached_set`) el contenido de la estructura `stat_mc` pasada por parámetro en un bloque con clave `file_name`. Como resultado devuelve cero si se completa la operación correctamente o -1 si ocurre un error al escribir en *Memcached*.

`int delete_stat(char *file_name)`: borra de la cache distribuida el valor asociado a la clave `file_name`, usualmente un bloque de metadatos. Utiliza la función `memcached_delete` de libMemcached para interactuar con *Memcached*. Devuelve cero si el borrado es correcto y -1 si no lo es.

`int write_chunk(char* key, void * buffer, int size)`: recibe por parámetro una clave (como una cadena de caracteres), un buffer de datos y el tamaño de dicho buffer de datos. Se comprueba que el tamaño del buffer es mayor que cero y menor que el tamaño de bloque y, en caso afirmativo, se escribe en *Memcached* a través de libMemcached (`memcached_set`) con la clave introducida por parámetro. Se suele usar para escribir bloques de datos en caché distribuida y sobrescribe cualquier dato que hubiese con esa clave. Devuelve cero en caso de que la escritura se produzca de forma satisfactoria y un número negativo si ocurre algún error, siguiendo estos códigos:

Código	Descripción
-1	No es posible escribir porciones de datos mayores que el tamaño de bloque ni menores de un byte.
-2	Error escribiendo en *Memcached*.

Tabla 43: Códigos de error de la función write_chunk

`int read_chunk(char* key, void * buffer)`: extrae el valor asociado con la clave `key` (cadena de caracteres) de *Memcached* utilizando libMemcached (`memcached_get`) y lo copia al buffer (previamente reservado en memoria con espacio suficiente, usualmente el tamaño de bloque del sistema de ficheros distribuido). Devuelve el tamaño de la lectura (valor mayor o igual que cero) si es correcta y -1 en caso contrario.

`int read_multi_chunks(int num_blocks, char** keys_list, void * buffer)`: su objetivo es leer varios bloques de datos de una sola vez mediante el uso de las funciones `memcached_mget` y `memcached_fetch` de libMemcached. Recibe por parámetro el número de bloques a leer de una sola vez, el puntero a un array de cadenas de caracteres que debe contener las claves de todos los bloques a leer (tantas claves como bloques) y, por último, un buffer con espacio reservado para alojar todos los bloques leídos (usualmente del tamaño de bloque multiplicado por `num_blocks`), en el que se copiarán. En primer lugar, se hace la lectura de todos los bloques de una sola vez mediante la función `memcached_mget`. Esta es la única conexión que se realiza con cada servidor de *Memcached*. A partir de ese momento, se recorren todos los bloques extraídos con `memcached_fetch` y, con cada valor leído, se recorre la lista de claves introducida por parámetro para conocer en qué posición del buffer se debe escribir (no tienen por qué llegar en orden). Esta operación de búsqueda de la posición adecuada consume mucho tiempo, por lo que la función penaliza mucho el rendimiento y no es utilizada en el código de la librería. La función que se utiliza es `read_multi_chunks_tweak_for_files`. Esta función se incluye por motivos de compatibilidad, ya que, a diferencia de la anterior, se puede utilizar para cualquier tipo de lectura, no solo para leer bloques de datos consecutivos.

Una vez recorridos todos los bloques que se han extraído de la caché distribuida y escritos en el *buffer*, se utiliza la función `memcached_quit` para indicar que se ha finalizado la lectura. Se devuelve el número de bytes leídos en caso de que la lectura de todos los bloques se realice correctamente o un número negativo en caso de que suceda algún error. Los códigos de error utilizados serán los descritos en la siguiente tabla:

Código	Descripción
-1	La clave obtenida de *Memcached* no se corresponde con ninguna de las esperadas.
-2	El bloque no estaba completo. No se espera que ocurra este error.
-3	Error leyendo de *Memcached*. El tamaño del bloque leído no se corresponde con el esperado.

Tabla 44: Códigos de error de la función read_multi_chunks

`int read_multi_chunks_tweak_for_files(int num_blocks, char** keys_list, void * buffer)`: el objetivo de esta función es exactamente el mismo que el de la anterior, dado un número de bloques a leer, una lista de claves (con `num_blocks` claves) y un puntero a un *buffer* reservado previamente, se pretende leer de la caché distribuida todos los valores asociados a las claves introducidas por parámetro y escribirlos en el *buffer*. La diferencia con la anterior función es que ésta está

optimizada para ser utilizada para leer bloques de datos consecutivos de un mismo fichero del sistema de ficheros distribuido, no cualquier conjunto de claves.

Para optimizar la función, se han tenido en cuenta las características de la lectura a realizar. Se conoce que los bloques a leer son consecutivos y está identificada su posición dentro del fichero en la propia clave, por tanto, conociendo cuál es la posición dentro del fichero del primer bloque a leer, se puede deducir la posición que ocupará cada bloque en el *buffer* tan solo conociendo su posición en el fichero. Un ejemplo de esto sería: se quieren leer los bloques del 10 al 20 de un fichero, por tanto, la primera clave de la lista será [id_fichero]>10 (en la posición cero del array). Si se recibe de *Memcached* la clave [id_fichero]>15, se trata del sexto bloque a leer y, lógicamente, ocupará la quinta posición del array de datos. Con esta información y el tamaño de bloque, se puede conocer la posición que ocuparán los datos en el buffer y escribirlos (puntero inicial + 5 * tam_bloque).

Como se puede deducir, esta función sólo es utilizable para leer bloques de datos consecutivos de un mismo fichero del sistema de ficheros distribuido, no para cualquier tipo de lectura múltiple. Pero en la librería implementada, sólo se utilizan las lecturas múltiples para leer bloques consecutivos de un fichero, por tanto, se mejora ampliamente el rendimiento evitando recorrer la lista completa de claves en cada iteración de `memcached_fetch`.

El resto del funcionamiento es exactamente igual que en la anterior función, con `memcached_mget` se leen todos los bloques de una vez, con `memcached_fetch` se recorren y con `memcached_quit` se finaliza la lectura. Se devuelve el número de bytes leídos en caso de que la lectura de todos los bloques se realice correctamente o un número negativo en caso de que suceda algún error. Los códigos de error utilizados serán los descritos en la siguiente tabla:

Código	Descripción
-1	La clave obtenida de *Memcached* no se corresponde con ninguna de las esperadas.
-2	El bloque no estaba completo. No se espera que ocurra este error.
-3	Error leyendo de *Memcached*. El tamaño del bloque leído no se corresponde con el esperado.
-4	Formato incorrecto de clave. La clave debe formar parte de un bloque de datos y contener el símbolo '>'.

Tabla 45: Códigos de error de la función read_multi_chunks_tweak_for_files

`int connect_to_memc_servers(char* servers_string)`: esta función suele ser utilizada en la inicialización del sistema de ficheros distribuido (`init_mc`). Recibe por parámetro una lista de servidores con el formato servidor:puerto separada por comas y conecta con ellos inicializando la variable global `memc` de tipo `memcached_st`, una especie de identificador de sesión necesario para todas las acciones que se realicen sobre *Memcached*. En primer lugar parsea la lista de servidores con la función `memcached_servers_parse` para separar las cadenas de caracteres y estructurarlas.

Acto seguido, introduce dicha estructura en `memc` mediante la función `memcached_server_push` y conecta con ellos. Se libera la lista de servidores (`memcached_server_free`) y se hace una prueba de escritura + lectura + borrado sobre la caché distribuida para comprobar que se ha conectado correctamente. Se devuelve cero si la conexión ha sido correcta y -1 si el formato de los servidores es incorrecto o no se puede conectar con ellos.

`int disconnect_memc_servers()`: desconecta de los servidores de *Memcached* cerrando la conexión con la función `memcached_free` que libera la variable global `memc`. Devuelve cero siempre (la función `memcached_free` no devuelve nada, por tanto, siempre se considera que se ha realizado correctamente). Se mantiene la devolución de un número entero para poder mantener la interfaz aunque se cambiase libMemcached por otra librería o se incluyesen más operaciones en esta función.

Funciones Auxiliares

`int parse_filename (char* orig_path, char file_name[MAX_FILE_NAME_LENGTH+1])`: filtra el nombre del fichero introducido por parámetro (`orig_path`) quitándole toda la ruta relativa que contenga, es decir, todo lo anterior a la última barra que se encuentre (barra incluida) quedando solo el nombre del fichero y su extensión (en caso de tenerla). Guarda el resultado en el array `file_name` pasado por parámetro.

`int file_exists(const char * filename)`: comprueba si el fichero `file_name` existe en el sistema de ficheros local en el que se está ejecutando el cliente. Devuelve `TRUE` (1) en caso afirmativo y `FALSE` (0) en caso contrario.

`int compare_keys(char * key1, int length1, char * key2, int length2)`: compara las dos claves introducidas por parámetro (`length1` y `length2` son las longitudes de la `key1` y la `key2` respectivamente). Se devuelve `TRUE` (1) si las claves son exactamente iguales y `FALSE` (0) en caso contrario.

`int search_last_sym(char* string, int key_length):` dado el puntero a una cadena de caracteres (habitualmente una clave identificadora de un bloque de datos) y su longitud, esta función devuelve la posición del último símbolo '>', es decir, aquel que esté más a la derecha (de haber más de uno) o -1 en caso de no encontrar ninguna ocurrencia.

utils.h

Se trata del fichero de cabeceras, pero además de las cabeceras de las funciones del fichero utils.c se incluyen otras cosas.

En primer lugar, los `include` necesarios para todas las funciones, dado que memcachedfs.c necesita las funciones de utils.c, algunos de sus `include` se han hecho también aquí.

- **`stdio.h`**: librería de entrada y salida estándar de POSIX (`open`, `read`, `write`, `close`, etc.)
- **`string.h`**: incluye funciones de tratamiento de cadenas de caracteres (copiado de cadenas, comparación, longitud, etc.)
- **`unistd.h`**: constantes y funciones utilizadas habitualmente, por ejemplo, `gethostid`.
- **`libs/sha2/sha2.h:`** tiene las funciones para realizar los resúmenes SHA-512. Es necesario que se encuentre en esa carpeta.
- **`sys/timeb.h`**: contiene la función `ftime`, utilizada para sacar la marca de tiempo actual con precisión de milisegundos.
- **`time.h`**: incluye otras funciones para tomar el tiempo con menos precisión, útiles para las marcas de tiempo de los metadatos.
- **`libmemcached/memcached.h`**: contiene todas las funciones de libMemcached para comunicarse con *Memcached*.
- **`sys/stat.h`**: librería con funciones y estructuras para tratar con los metadatos de los ficheros locales.
- **`stdlib.h`**: contiene muchas funciones útiles para programar en C: gestión de memoria dinámica (`malloc`, `calloc`, `free`…), generación de números aleatorios, conversores de cadena de caracteres a número, etc.

También incluye constantes utilizadas en toda la librería:

- **`BLOCK_SIZE:`** tamaño de bloque del sistema de ficheros distribuido medido en Kbyte. Puede ser modificado entre 1 y 1023 según las necesidades.
- **`TRUE:`** se utiliza para emular variables booleanas con enteros, ya que, C carece de este tipo básico. Tiene como valor predeterminado 1 (aunque podría ser cualquiera distinto de 0) y no se debe modificar.
- **`FALSE:`** se utiliza para emular variables booleanas con enteros, ya que, C carece de este tipo básico. Tiene como valor predeterminado 0 y no se debe modificar.
- **`O_RDONLY:`** representa el modo de acceso de solo lectura. Tiene valor predeterminado 0 y no debe cambiarse.
- **`O_WRONLY:`** representa el modo de acceso de solo escritura. Tiene valor predeterminado 1 y no debe cambiarse.
- **`O_RDWR:`** representa el modo de acceso para lectura y escritura. Tiene valor predeterminado 2 y no debe cambiarse.
- **`MAX_FILE_NAME_LENGTH:`** longitud máxima del nombre de fichero. Tiene como valor predeterminado 249 (uno menos que el tamaño máximo de clave para evitar problemas con valores nulos al final de la clave) y no debe ser modificado.

- **MAX_LOCAL_FILE_LENGTH:** representa la longitud máxima del nombre de fichero en sistemas de ficheros locales habituales (ext2). Se debe adaptar al sistema de ficheros local en el que se encuentre el cliente, aunque la mayoría rondan la cifra predeterminada de 255 bytes.
- **ID_LENGTH:** longitud de los identificadores. El valor predeterminado es 128. Todos tienen esta longitud gracias al algoritmo de resumen SHA-512 que se les pasa. No debe modificarse a menos que se modifique el algoritmo de hash utilizado.
- **MAX_KEY_LENGTH:** longitud máxima que puede tener una clave para utilizarla en *Memcached*. El valor predeterminado es 250 y no debe cambiarse a menos que se cambie el sistema de caché distribuida o se quieran claves más cortas por alguna razón, por ejemplo, de ahorro de espacio. Está desaconsejado su cambio, ya que, la generación de muchas claves se basa ampliamente en que el tamaño de clave sea 250 bytes, por ejemplo, los resúmenes de 128 bytes.
- **FS:** representa el modo de apertura normal, es decir, como un sistema de ficheros distribuido. Su valor predeterminado es 0 y no debe cambiarse.
- **PRECAHCE:** representa el modo de apertura pre-caché, es decir, un fichero local se copia totalmente a caché distribuida y se trabaja con él desde allí devolviéndolo a disco cuando se termina de trabajar con él. Su valor predeterminado es 1 y no debe modificarse.
- **CACHE:** representa el modo de apertura caché, es decir, se utiliza la caché distribuida como caché de un sistema de ficheros local. Su valor predeterminado es 2 y no debe modificarse.

También se incluye como variable global un puntero a una estructura del tipo `memcached_st` donde guardar los datos necesarios para todas las funciones de libMemcached que requieren conectar con *Memcached*.

7 VERIFICACIÓN DEL SISTEMA

Debido a que el funcionamiento de la librería desarrollada es muy similar a la librería de entrada y salida estándar de Linux, para probar el correcto funcionamiento del sistema de ficheros distribuido se va a tomar como partida el funcionamiento de las funciones de Linux.

Durante el desarrollo se han ido probando las diferentes funciones del sistema a medida que se desarrollaba, sin embargo, de cara a probar el funcionamiento completo de la librería, se ha preferido optar por pruebas de estrés complejas en lugar de pruebas unitarias, ya que, se pueden comparar los resultados con los arrojados por el sistema de ficheros local, permitiendo pruebas mucho más completas que las unitarias.

7.1 Escrituras aleatorias

El código de esta prueba se encuentra en la siguiente ruta dentro de la carpeta del código: `/tests/test_random_writes.c`. Se trata de una prueba bastante compleja como se ha indicado anteriormente.

Se introducen por parámetro en la línea de comandos el nombre del fichero con el que se va a trabajar y la lista de servidores de *Memcached* que se van a usar en la prueba.

Si no existen ya, se crean cuatro copias del fichero. Acto seguido, se abre cada una de las copias en uno de los cuatro modos posibles: local, sistema de ficheros distribuido, pre-caché distribuida y caché distribuida. En el caso del sistema de ficheros distribuido, además, se copia a la caché distribuida una de las copias del fichero (en los modos pre-caché y caché no es necesario).

A continuación, se genera un bloque de datos aleatorio (tanto en su tamaño como en su contenido) y un *offset* aleatorio (que puede superar el tamaño actual del fichero). Con estos datos se realizan escrituras iguales en los ficheros abiertos en los cuatro modos.

Se cierran los ficheros (en el caso del modo de sistema de ficheros distribuido, además, se copia el fichero del sistema distribuido al disco duro local, en pre-caché y caché se hace automáticamente) y se comparan entre sí mediante la herramienta '`diff`'.

Como el funcionamiento de todas las escrituras realizadas debe ser el mismo, los cuatro archivos que se encuentran en el disco duro local deben ser exactamente iguales y el comando '`diff`' no debe imprimir nada al ser ejecutado. Además, se imprime por

pantalla el tamaño de cada fichero extraído con `stat` (el local, del fichero local, el distribuido y el pre-caché de los ficheros que se encuentran en la caché distribuida, y en el modo caché se imprime un valor inválido, ya que, al ser un fichero temporal, no se puede acceder a sus metadatos sin conocer el identificador temporal).

Para que la prueba sea realmente fiable, debe ser ejecutada múltiples veces (obviamente, cuantas más, mejor), sin que en ninguno de los casos la herramienta '`diff`' lance ningún error. Además, se puede modificar el tamaño de bloque y volver a lanzar la prueba con otro fichero, para comprobar el funcionamiento con distintos tamaños de bloque. También es posible realizar la prueba con diferentes servidores de *Memcached* para comprobar que funciona bien en entornos multi-servidor.

Con esta prueba, se consigue verificar a la perfección el correcto funcionamiento de varias funcionalidades: la inicialización del sistema de ficheros distribuido (`init_mc`), el sistema de metadatos (stat_mc), las aperturas de ficheros en todos los modos (`open_mc`, `open_precache_mc` y `open_cache_mc`) las escrituras aleatorias con diferentes tamaños de bloque y en todos los modos de apertura (`write_mc`) y el cerrado de ficheros en todos los modos de apertura (`close_mc`).

Con respecto a los requisitos que se prueban con este caso, son todos los de capacidad salvo tres: el referente a la tolerancia a fallos (RUC-006) y los referentes a extracciones múltiples desde la caché distribuida (RUC-011 y RUC-012).

El requisito RUC-006 se podría probar "tirando" uno de los servidores de *Memcached* durante el proceso de la prueba y comprobando que se produce un fallo irrecuperable, mientras que los requisitos referidos a las extracciones múltiples solo tienen sentido de cara a las lecturas.

7.2 Lecturas aleatorias

El código de esta prueba se encuentra en la siguiente ruta dentro de la carpeta del código: `/tests/test_random_reads.c`. De nuevo se trata de una prueba bastante compleja.

Al igual que en el caso anterior, se introducen mediante línea de comandos el nombre de un fichero que se encuentre en el sistema de ficheros local y la lista de servidores de *Memcached* con la que se quiere trabajar (que deben estar activos y accesibles).

Los primeros pasos son similares a los anteriores, se crean cuatro copias del fichero introducido por parámetro, en caso de que no existan, y se abre cada copia en uno de los distintos modos: local, sistema de ficheros distribuido, pre-caché y caché. Además, como en el caso anterior, en modo sistema de ficheros distribuido se transfiere una de las copias a la caché distribuida.

Se genera un tamaño de bloque aleatorio y un *offset* aleatorio y se realizan ocho lecturas divididas en dos bloques de cuatro.

Por un lado se realizan cuatro lecturas simples en cada uno de los ficheros abiertos (`read` y `read_mc`). Se guardan los resultados de dichas lecturas y se comparan entre sí, imprimiendo por pantalla la expresión "FAIL!" (que significa fallo) en caso de que exista alguna diferencia en el resultado de la lectura, tanto en el valor de retorno como en el contenido del *buffer* leído.

Por otro lado, se realizan otras cuatro lecturas, sin embargo, para las que se realizan en ficheros de MemcachedFS, se utiliza la función de lecturas múltiples (`read_mc_mult`). Acto seguido, se comparan los resultados de todos los modos de lectura de la misma forma que se ha explicado anteriormente.

A diferencia de la prueba anterior, ya que no es necesario cerrar el fichero y volcarlo a disco para comprobar que la operación ha sido correcta, se realizan 100 lecturas aleatorias cada vez que se ejecuta la prueba, suficientes para comprobar el correcto funcionamiento de todas las funciones.

Como paso final, se cierran todos los ficheros y, como comprobación adicional, se utiliza '`diff`' para comparar los cuatro ficheros que quedan en el disco local, para asegurarse de que no se ha producido ningún cambio en ninguno de ellos tras las lecturas.

Es interesante lanzar esta prueba bajo distintas condiciones, tanto de tamaño de bloque del sistema de ficheros distribuido, como del número de servidores de *Memcached* sobre los que se distribuirán los datos.

Entre las dos pruebas expuestas, se evalúa el correcto funcionamiento de todos los servicios ofrecidos por el sistema de ficheros distribuido, incluidas las funciones auxiliares para copiar ficheros del sistema de ficheros local al distribuido y viceversa, salvo las funciones `rename_mc` y `unlink_mc` que han sido probadas exhaustivamente mediante procedimientos más sencillos que no requieren una explicación en profundidad (consiste simplemente renombrar un fichero y comprobar que no existe el viejo y el nuevo funciona correctamente y borrar un fichero comprobando que deja de poder ser accedido).

Con respecto a los requisitos cubiertos por esta prueba, son todos los de capacidad salvo el referente a la tolerancia a fallos (RUC-006), que puede ser comprobado de manera manual cerrando el proceso `memcached` en cualquiera de los servidores de *Memcached* que se estén utilizando y comprobando que el sistema deja de funcionar correctamente.

8 EVALUACIÓN

8.1 Entorno de pruebas

Para realizar la evaluación de rendimiento se han utilizado dos sistemas diferentes. Por un lado, para la evaluación del rendimiento de un sistema de ficheros local, se ha utilizado el siguiente equipo:

AMD® Phenom II x4 955 @3.20 GHz

4GB DDR3 1333 MHz

Western Digital Caviar Black 640 GB 7.200 rpm SATA2

Sistema de ficheros Ext4

Ubuntu 10.10 Desktop Maverick Meerkat

Se trata de un equipo con la potencia suficiente para que, ni la memoria RAM, ni la capacidad de cálculo del procesador lastren las velocidades de entrada y salida del disco duro. Por su parte, el disco duro es uno de los más veloces del mercado orientado a usuarios domésticos y, por tanto, podrá ser utilizado como baremo fiable de los sistemas de almacenamiento mecánicos.

Para las evaluaciones que requieren más de una máquina, la universidad ha cedido uno de sus clústeres. Se trata de un sistema de 24 nodos de cómputo con las siguientes características:

Intel® Xeon CPU E5405 @2.00GHz

Memoria 4GB DDR3

Tarjeta de red Gigabit Ethernet 100/1000

Ubuntu 10.10 Server Maverick Meerkat

De nuevo se trata de un sistema con potencia suficiente, tanto en lo referente a su capacidad de procesamiento como en la velocidad de la memoria principal, para no tener absolutamente ningún tipo de cuello de botella relacionado con su capacidad de

cálculo. Con respecto a la red utilizada, se trata de un red Gigabit Ethernet, sencilla y barata, muy extendida en la actualidad.

8.2 Evaluación del tamaño de bloque

Como primera evaluación de rendimiento, se va a evaluar el tamaño de bloque óptimo para la utilización de *Memcached*. *Memcached* permite asociar a una clave valores desde 1 byte hasta un máximo de 1024 Kbyte (no incluido). El objetivo de este procedimiento es leer y escribir de *Memcached* valores de distintos tamaños, para evaluar la evolución de su rendimiento. Para ello, no se va a utilizar en absoluto la librería desarrollada para el sistema de ficheros distribuido, sino que se utilizarán las llamadas de 'set' (escritura) y 'get' (lectura) que ofrece libMemcached.

Además, se pretende comprobar las diferencias de rendimiento que puedan existir en función del número de servidores de *Memcached* utilizados.

Debido a que los tiempos de transferencia pueden variar en función de las condiciones de tráfico de la red, se han utilizado dos metodologías: medición del valor medio y medición del peor valor, ambos en 64 intentos.

El primer caso estudiado es el que recopila los peores casos. Consiste en escribir valores aleatorios de entre 1 Kbyte y 1023 Kbyte en *Memcached*, utilizando entre 1 y 16 servidores. Se realiza la escritura del mismo valor en 64 ocasiones y se guarda el peor caso de todos, para intentar evitar posibles efectos de caché.

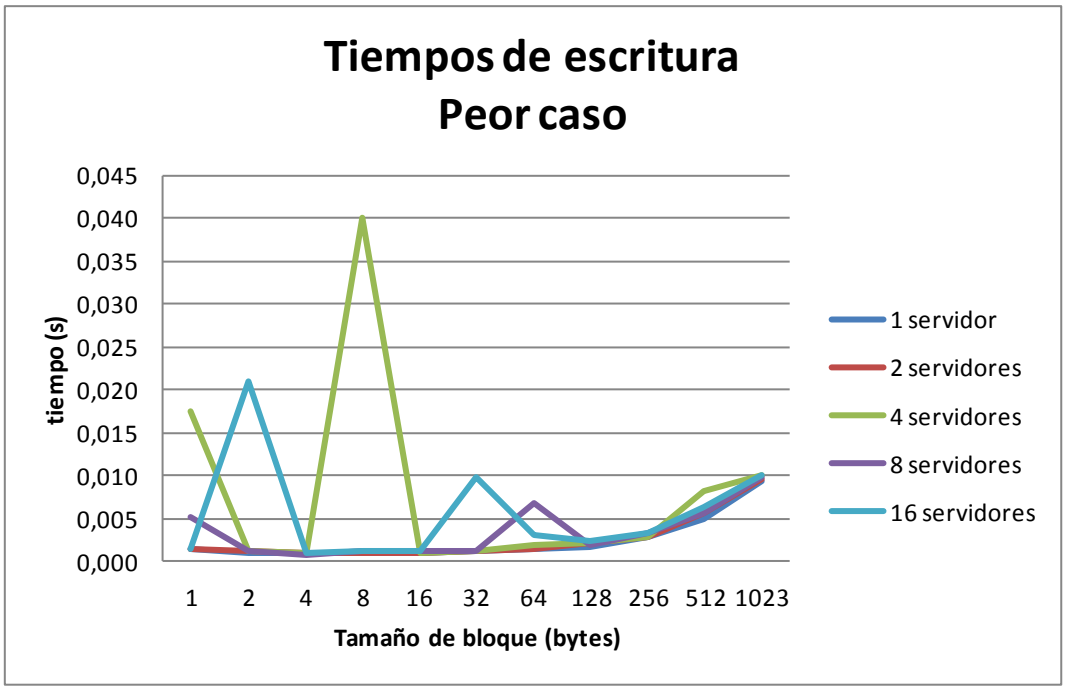

Figura 9: Gráfico de tiempos de escritura de un valor en Memcached (peor tiempo en segundos)

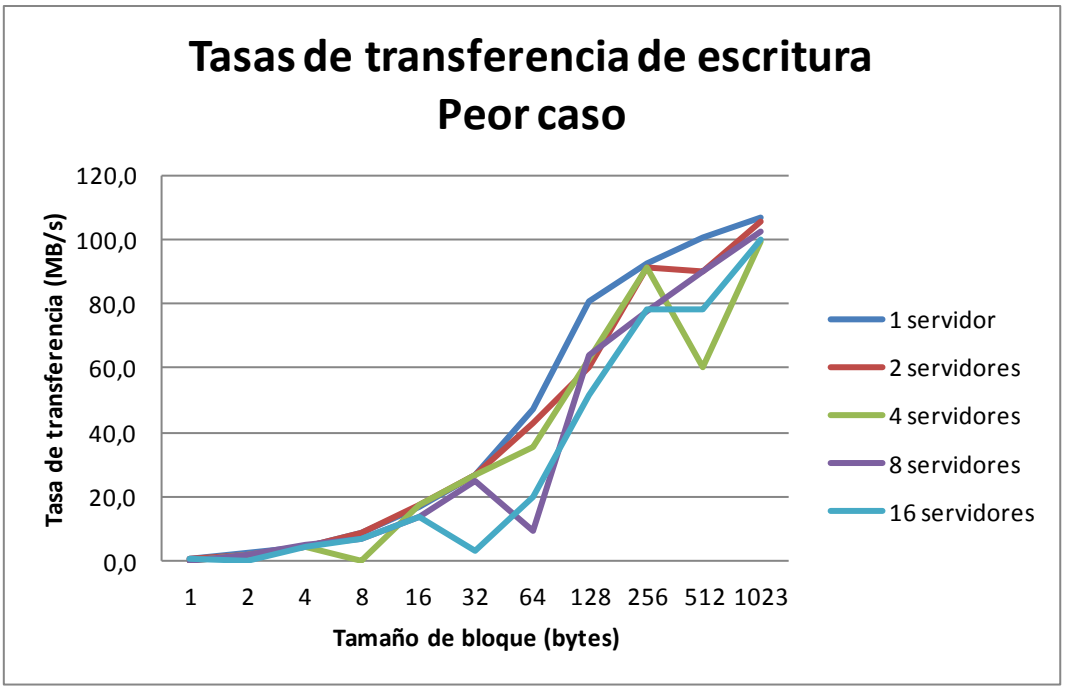

Figura 10: Gráfico de tasas de transferencia de escritura de un valor en Memcached (peor caso en MB/s)

Como segundo caso, se ha realizado exactamente la misma evaluación, pero realizando la media de los tiempos de las 64 escrituras. De este modo, se trata de evitar "picos" de carga de la red o del sistema, que puedan afectar a alguna medición aislada, dando lugar a un resultado poco realista.

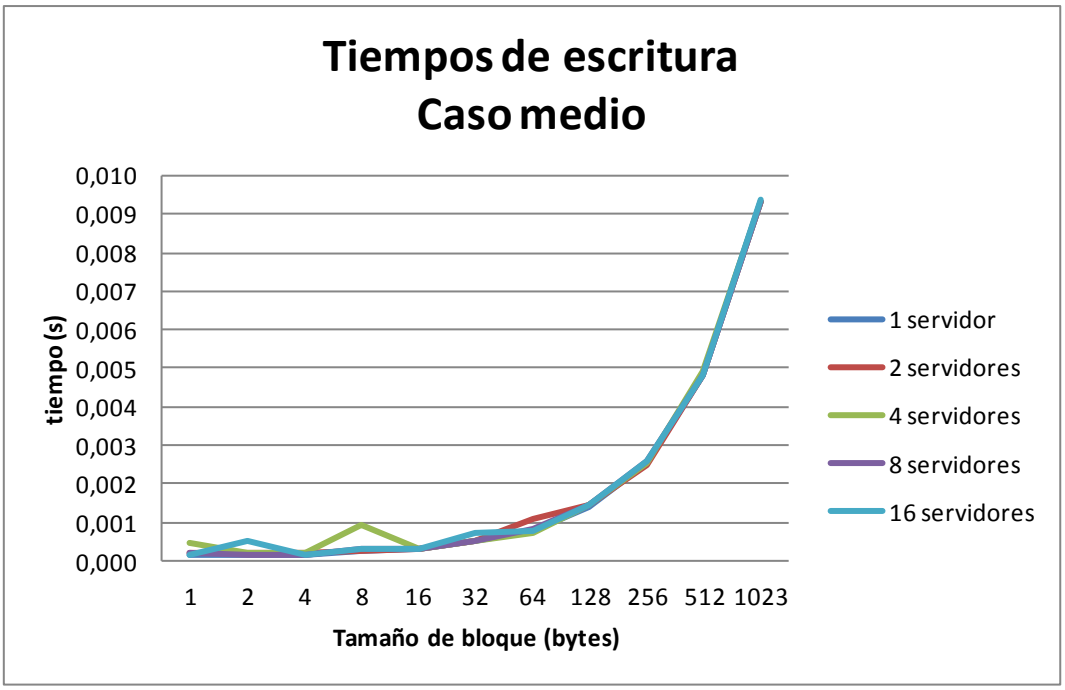

Figura 11: Gráfico de tiempos de escritura de un valor en Memcached (tiempo medio en segundos)

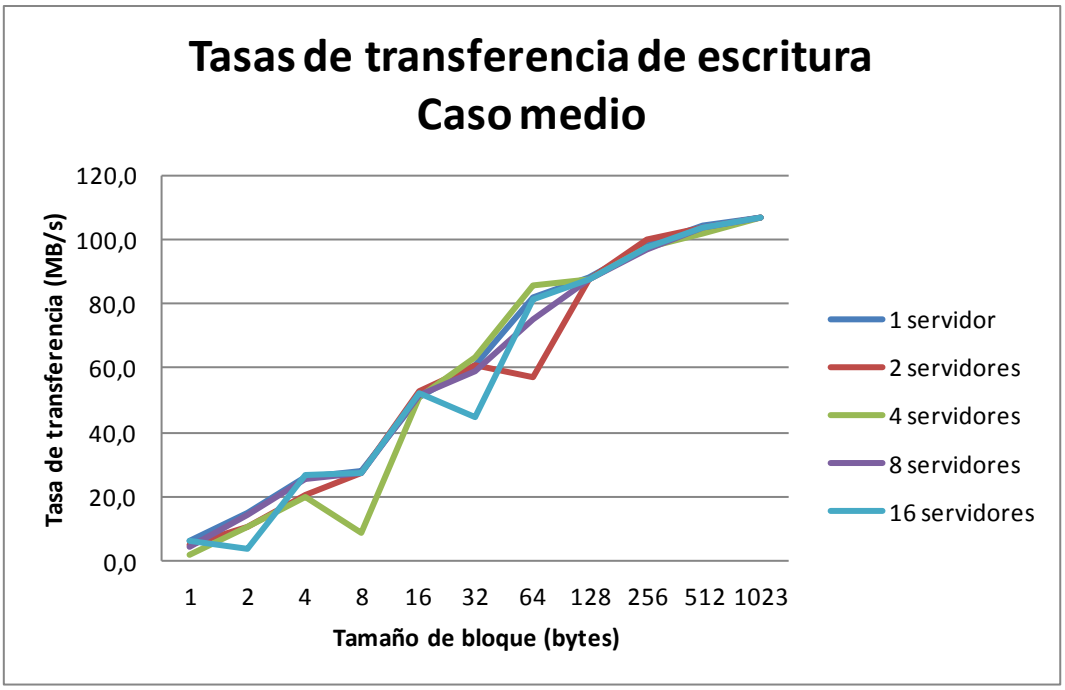

Figura 12: Gráfico de tasas de transferencia de escritura de un valor en Memcached (caso medio en MB/s)

Con estas primeras cuatro gráficas se pueden empezar a sacar algunas conclusiones. Como primer punto, se debe observar las diferencias entre el peor caso y el caso medio. Las diferencias entre ambos casos son más acusadas cuanto más bajo es el tamaño del valor que se escribe en la caché distribuida. Este hecho, se debe principalmente a que se trata de tiempos muy pequeños, en los que cualquier desvío del comportamiento habitual se penaliza enormemente. Por ejemplo, un paquete que se pierda, una carga excesiva puntual del servidor o cualquier otro hecho habitual que dificulte la transferencia, puede penalizar alguno de los casos en tan sólo unas milésimas, que cambian por completo la velocidad de transferencia.

Como se puede observar en el caso medio, las gráficas forman líneas más perfectas, que modelan mejor el resultado, evitando valores extremos que dan lugar a picos irreales. Por todo esto, el análisis de los resultados en este caso se fundamentará en los casos medios, especialmente en las tasas de transferencia, que son más sencillas de interpretar.

De las dos últimas gráficas, se puede concluir casi con total seguridad que, salvo casos puntuales, el rendimiento no depende en absoluto del número de servidores utilizado, manteniendo una escalabilidad perfecta.

Por otro lado, a medida que aumenta el tamaño de bloque, mejoran las tasas de transferencia. Las causas más probables de este comportamiento son dos:

- **Conexión:** como en casi todos los protocolos autocontenidos, en *Memcached* se requiere enviar el estado en cada transacción. Esta

información es siempre del mismo tamaño, por tanto, cuanto mayor sea la carga posterior, más sencillo es "rentabilizar" la conexión.

- **Optimización de la transferencia:** cuanto mayor es el volumen de datos a transferir por la red, mejor se aprovechan las características de la red, llenando al máximo los paquetes de datos y maximizando la eficiencia.

Es interesante destacar, que con valores a partir de 256 Kbyte, se roza el límite de la red. Gigabit Ethernet tiene un límite teórico de 1 Gbps, es decir, 1000 Mbps, o lo que es lo mismo, 125 MB/s. Sin embargo, debido a cuestiones técnicas y de protocolo, la velocidad máxima real de las redes Ethernet para el transporte de datos, es decir, sin incluir las cabeceras como en el límite teórico, se suele considerar en torno a un 80% de la teórica, es decir, unos 100 MB/s. De este modo, se demuestra que *Memcached* aprovecha al máximo la capacidad de la red sobre la que trabaja con valores lo suficientemente grandes.

El siguiente caso estudiado, es el de las lecturas, siguiendo exactamente la misma metodología anterior. Lecturas de valores entre 1KB y 1023KB, utilizando entre 1 y 16 servidores de *Memcached*. 64 lecturas de cada valor, en las que se estudiará tanto el peor caso como el caso medio. En primer lugar, estos son los resultados obtenidos de medir el peor caso.

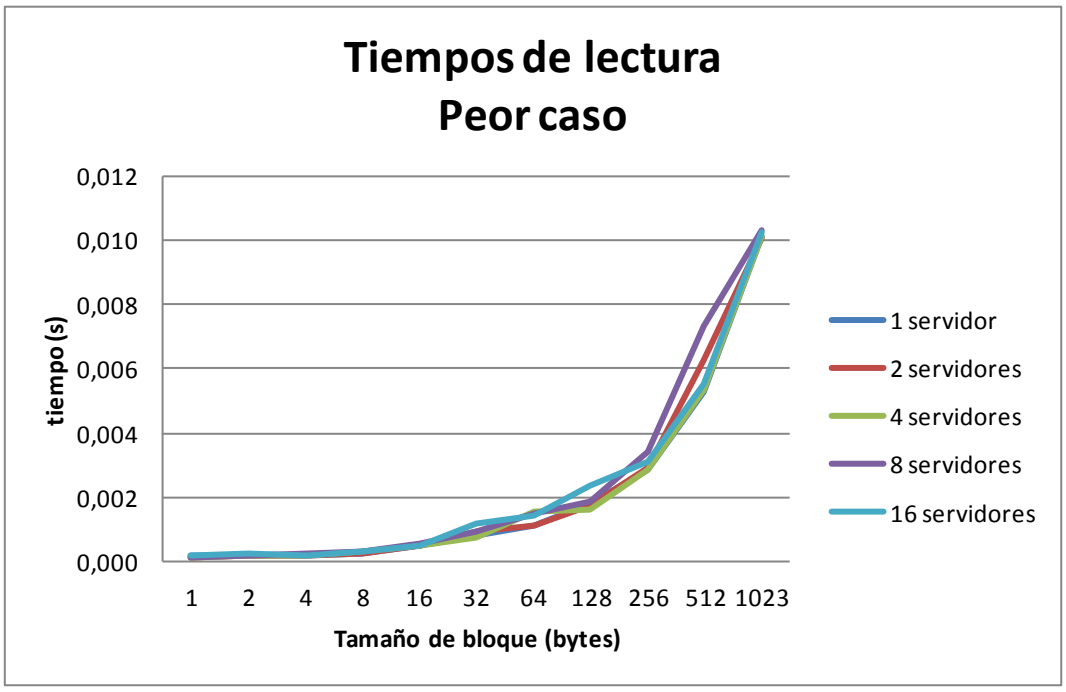

Figura 13: Gráfico de tiempos de lectura de un valor en Memcached (peor tiempo en segundos)

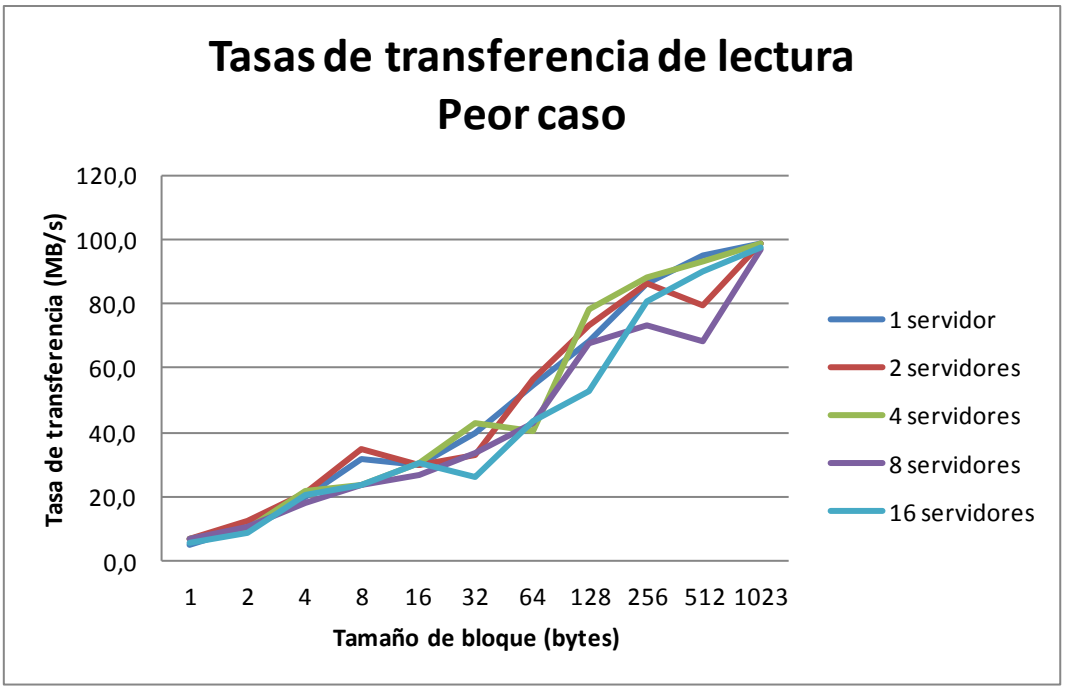

Figura 14: Gráfico de tasas de transferencia de lectura de un valor en Memcached (peor caso en MB/s)

A continuación, se incluyen los resultados obtenidos de la evaluación en la que se realizó la media de 64 mediciones.

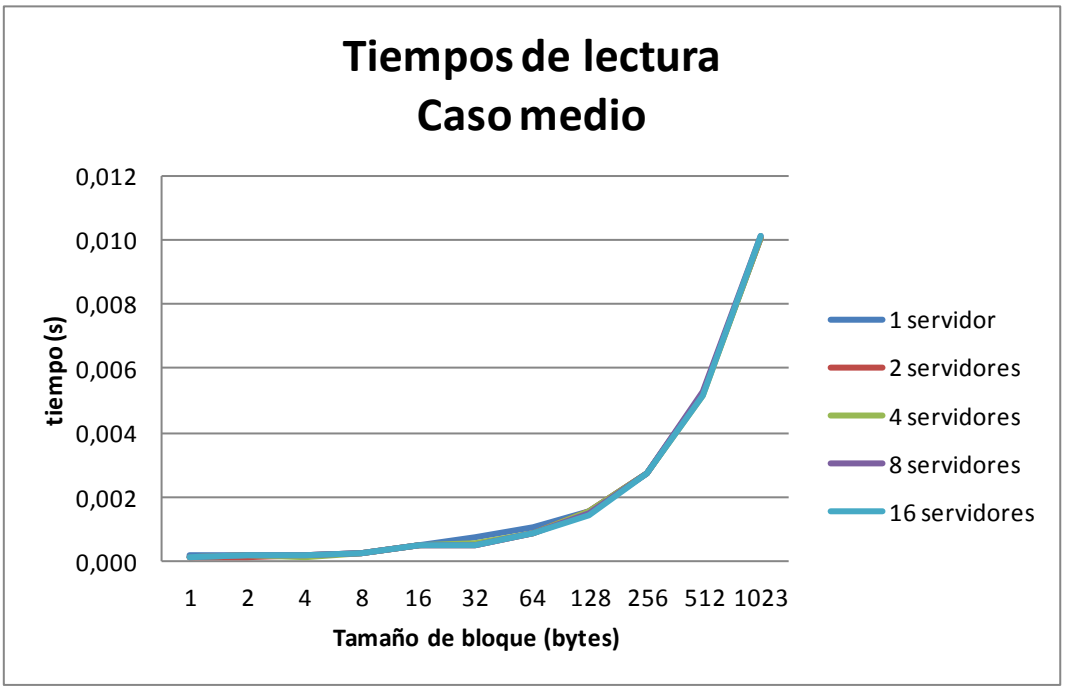

Figura 15: Gráfico de tiempos de lectura de un valor en Memcached (tiempo medio en segundos)

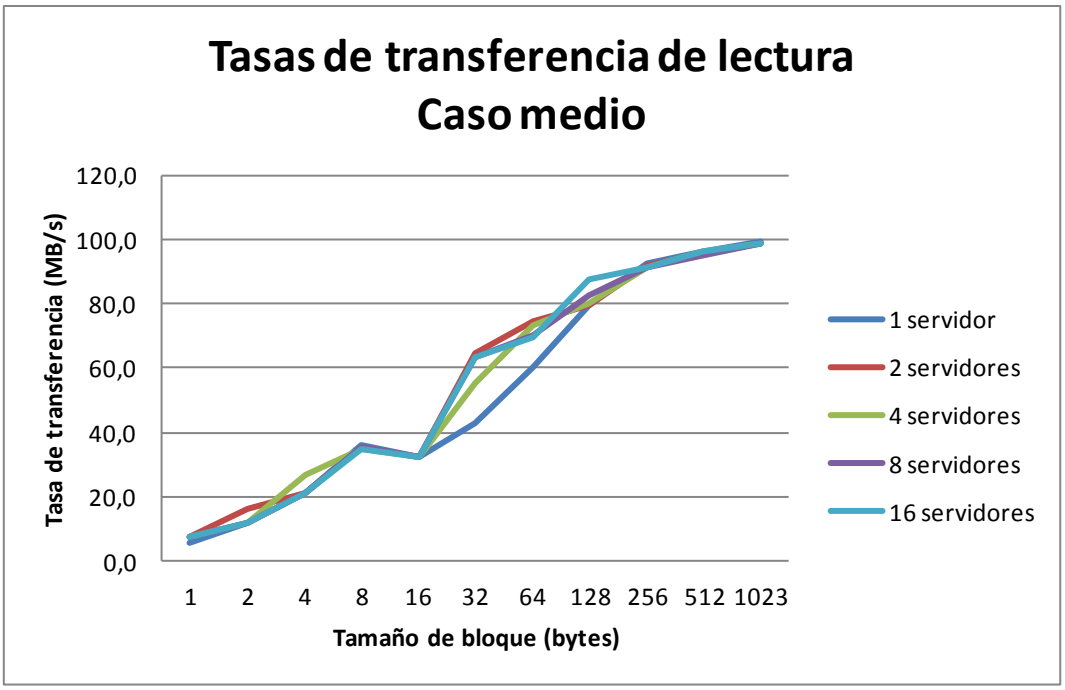

Figura 16: Gráfico de tasas de transferencia de lectura de un valor en Memcached (caso medio en MB/s)

Las conclusiones que se pueden sacar de esta evaluación de lectura, son las mismas que se han detallado sobre las escrituras: perfecta escalabilidad, mejora de las tasas de transferencia a medida que aumenta el tamaño del valor y aprovechamiento total de la red en transferencias a partir de 256 Kbyte.

Los únicos detalles que diferencian los resultados de esta evaluación con respecto a los obtenidos en las evaluaciones de escritura son:

- **Menos diferencia entre peor caso y caso medio:** en este caso se notan aún menos diferencias entre el peor caso y el caso medio, haciendo aún más sencillo el análisis y más fiables los resultados.

- **Resultados más fieles al modelo:** apenas existen valores extremos en ninguna de las gráficas, esto es especialmente visible en la Figura 15 en la que todas las líneas, independientemente del número de servidores, son prácticamente iguales y formando a la perfección la figura exponencial.

8.3 Evaluación de lectura/escritura de ficheros

8.3.1 Sistema de ficheros local

En primer lugar, se evalúa el rendimiento de un sistema de ficheros local. Para ello, se escribirán en el disco duro ficheros con datos aleatorios de diferentes tamaños, desde 1 Kbyte hasta 64 Mbytes. Las características del sistema de ficheros son las

indicadas en el entorno de pruebas, Ext4 sobre un Western Digital Caviar Black de 640 GB con unas tasas de transferencia máximas de en torno a 80 MB/s en condiciones ideales sin utilizar su caché. Para conocer este valor orientativo se ha utilizado la herramienta `hdparm`, mediante el comando **`hdparm -t /dev/sda`**, cuyo resultado es:

```
Timing buffered disk reads:  248 MB in  3.03 seconds =  81.93 MB/sec
```

De cara a la obtención de unos valores fiables, era necesario evitar a toda costa la caché del disco duro, de otro modo, los resultados obtenidos no serían extrapolables a cualquier uso del sistema de ficheros, sino a unos muy concretos en los que la caché funcionase exactamente igual que en la prueba. Además, los resultados estarían falseados, ya que, en algún momento es necesario volcar la caché para que los datos queden de forma persistente en el dispositivo.

Para esquivar la caché se han utilizado dos herramientas de Linux. La primera, es la función `sync()` de C, que vuelca al disco físico todos los datos que se encuentran en la caché del disco duro. La otra herramienta utilizada, es una herramienta que mediante el comando "**`sudo echo 3 | sudo tee /proc/sys/vm/drop_caches`**" es capaz de borrar la caché del disco duro.

De este modo, al iniciar la evaluación se vuelca a disco duro todo el contenido de la caché, para evitar problemas de coherencia y, a partir de ese momento, cada vez que se realiza una lectura local, se envían los datos de las cachés a disco con sync() para evitar que se pierdan y, acto seguido, se borran las cachés. En el caso de las escrituras, se borran las cachés antes de comenzar el proceso de escritura y se hace sync() tras cerrar el fichero para hacer persistentes los datos. Es necesario borrar la caché antes de comenzar cada operación de escritura, ya que en caso contrario, se estaría penalizando el rendimiento del sistema, debiendo volcar a disco no sólo la última operación, sino todos los datos que estén almacenados en la caché de operaciones previas.

Las mediciones de la evaluación no se reducen a medir tan sólo el tiempo de lectura/escritura. Se han medido los tiempos de apertura del fichero, lectura/escritura, cierre del fichero y el tiempo total que tarda en hacer las tres operaciones. Todas las mediciones han sido realizadas tres veces, arrojando los siguientes resultados al hacer la media de las tres evaluaciones:

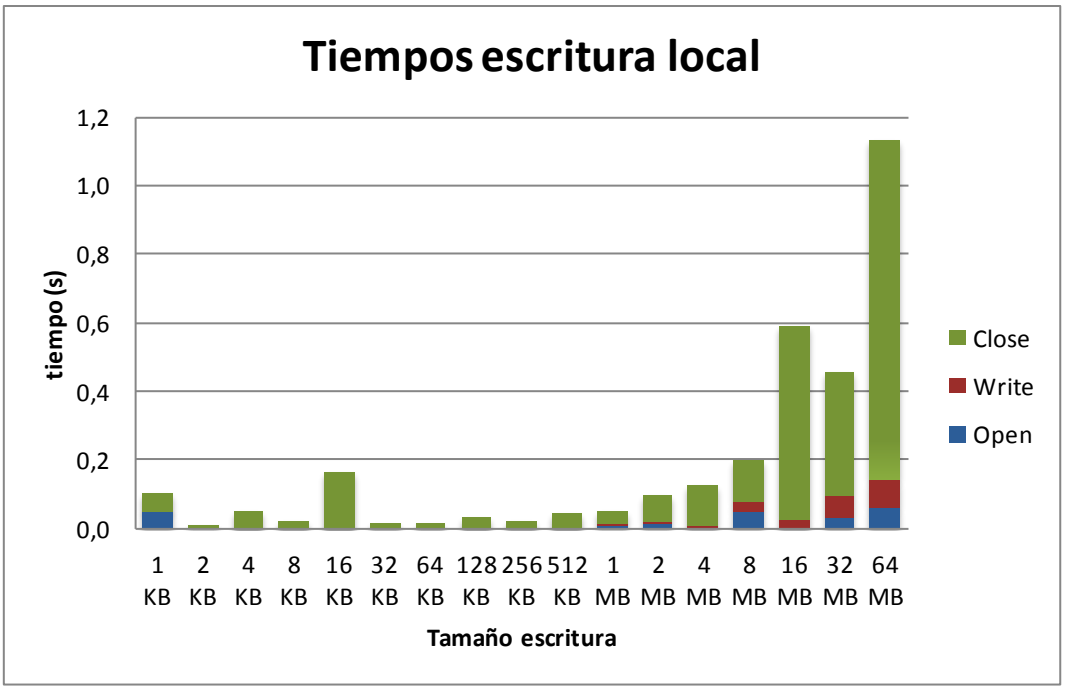

Figura 17: Gráfico descomposición tiempos de escritura de un sist. fich. local

En el gráfico se puede observar que la mayor parte del tiempo empleado para escribir un fichero en disco se invierte en el proceso de cerrado. Esto es así debido a que Linux mantiene el fichero en memoria principal o caché (acceso más rápido) hasta que el fichero se cierra, momento en que se considera un estado estable y se vuelca a disco. Además, es este el momento en que se utiliza la función `sync()` para volcar la caché a disco. Para medir la eficiencia del sistema de ficheros local completo (no solo del dispositivo) y compararla con el distribuido, utilizaremos el total del proceso (open + write + close) como medición de la escritura completa de un fichero. Tomando el total, se obtienen las siguientes tasas de transferencia.

Tamaño fichero	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB
Tasa (MB/s)	0,010	0,261	0,084	0,397	0,096	2,459	4,579	3,930	13,856

Tamaño fichero	512 KB	1 MB	2 MB	4 MB	8 MB	16 MB	32 MB	64MB
Tasa (MB/s)	12,422	22,212	20,852	32,740	40,457	27,070	70,093	56,510

Tabla 46: Tasas de transferencia escritura de un sist. fich. local

Para las lecturas, se ha realizado exactamente el mismo proceso (media de tres lecturas de un fichero, obteniendo los siguientes resultados:

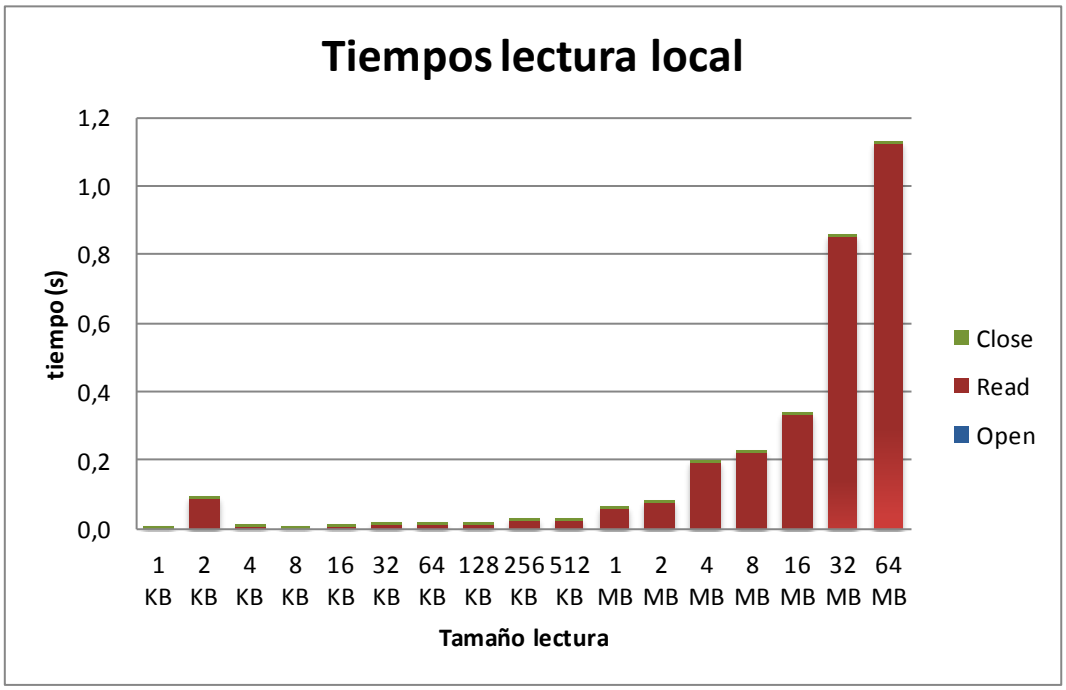

Figura 18: Gráfico descomposición tiempos de lectura de un sist. fich. local

En este caso, se observa que la operación que más tiempo demanda es la lectura en sí misma, esto se debe a que el fichero no sufre ninguna modificación, por tanto, no hay que volcar su contenido al disco, una vez realizada la lectura, el cierre es casi inmediato.

Aún así, el tiempo en el que se basará el análisis, será de nuevo el total, por el mismo motivo de antes, medir el tiempo completo de la lectura de un fichero en un sistema de ficheros, no solo el rendimiento del dispositivo.

Tamaño fichero	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB
Tasa (MB/s)	1,371	0,022	0,638	2,001	1,798	2,956	4,203	11,385	11,338

Tamaño fichero	512 KB	1 MB	2 MB	4 MB	8 MB	16 MB	32 MB	64MB
Tasa (MB/s)	22,450	16,387	25,782	20,607	36,421	47,948	37,547	56,827

Tabla 47: Tasas de transferencia lectura de un sist. fich. local

Las observaciones destacables sobre las tasas de transferencia, coinciden tanto en el caso de las lecturas como en el de las escrituras.

- **Evolución del rendimiento:** se nota una clara evolución en el rendimiento del dispositivo a medida que el tamaño de fichero crece. Cuanto mayor es el fichero, más posibilidades hay de hacer escrituras/lecturas secuenciales de gran tamaño, el mejor caso de este tipo de discos.

- **Inconsistencia de los resultados:** a pesar de que a medida que crece el tamaño mejora la velocidad, no es así en absolutamente todos los saltos de tamaño. El motivo es que no siempre las escrituras/lecturas son secuenciales en toda la extensión del fichero. La velocidad dependerá, por tanto, de las secciones secuenciales que haya y de la posición de estas (no se escribe/lee igual de rápido en todas las zonas del disco). Además, si otro proceso del sistema intenta utilizar el disco duro durante la evaluación, la aguja tendrá que desplazarse a la zona a la que quiera acceder y volver a la requerida por el procedimiento de evaluación, penalizando enormemente el rendimiento. Este tipo de operaciones involuntarias sobre el disco son difícilmente evitables, se ha pretendido minimizar sus efectos mediante la toma de varias mediciones, tres en concreto, y el cálculo de su media.

- **Aprovechamiento de la velocidad del dispositivo:** la máxima velocidad de lectura/escritura en condiciones óptimas del disco duro ronda los 80 MB/s, sin embargo, a pesar de acercarse, no se han conseguido estas tasas. Los motivos son, en primer lugar lo comentado en el punto anterior y, en segundo lugar, la utilización del sistema de ficheros, las operaciones de `open` y `close`, así como la escritura/lectura de metadatos pueden suponer cierto coste que penalice el rendimiento.

8.3.2 Sistema de ficheros MemcachedFS

La metodología de evaluación será muy similar a la anterior para poder comparar más adelante los resultados obtenidos. Sin embargo, en este caso existen muchas más variables que deben ser tenidas en cuenta.

- **Número de servidores:** el número de servidores de *Memcached* utilizados en los procedimientos de evaluación puede influir en el rendimiento del sistema de ficheros distribuido. Se realizarán evaluaciones con uno, dos, cuatro y ocho servidores de *Memcached*.

- **Tamaño de bloque:** el tamaño de bloque del sistema de ficheros distribuido es fundamental de cara a su análisis de rendimiento. Las evaluaciones se realizarán, en principio, con el tamaño de bloque máximo, que fue el que mostró un mejor rendimiento en las evaluaciones sobre el tamaño de bloque.

Pero también se probará el rendimiento con 128 Kbyte (el límite en el que empezaba a perder rendimiento), 1 Kbyte (el menor tamaño de bloque posible) y 8 Kbyte (un valor habitual en los sistemas de ficheros locales, quizá menos habitual que 4 Kbyte, pero más alejado de 1 Kbyte para observar mejor las diferencias).

- **Lecturas múltiples:** se han implementado dos tipos de lecturas. En un caso (lectura normal) se lee bloque a bloque desde *Memcached* y, en otro, (lectura múltiple) se realiza la lectura de todos los bloques completos de una sola vez. Será interesante observar las diferencias entre ambos tipos de lectura.

- **Modos de apertura:** se han implementado tres modos de apertura, como un sistema de ficheros normal, y dos modos de cacheo (Pre-caché y caché). Se comenzará por el estudio del modo normal para poder compararlo con el sistema de ficheros local y, a continuación, se procederá al estudio de los modos de cacheo.

Teniendo en cuenta estos datos, las primeras evaluaciones a realizar serán lecturas y escrituras con bloques de 1023 Kbyte, en 1, 2, 4 y 8 servidores de *Memcached*. Primero las escrituras y, después, las lecturas. La distribución de los tiempos open/write/close es muy similar en todos los casos, por lo tanto, se ha seleccionado un caso intermedio (4 servidores) para mostrarlo mediante una gráfica:

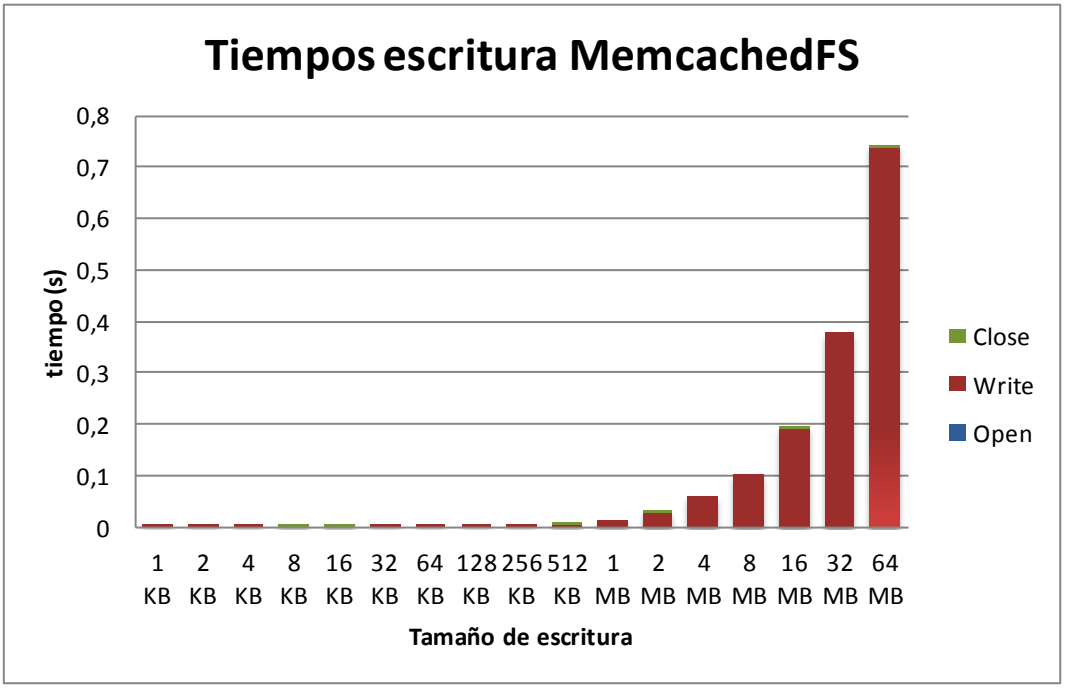

Figura 19: Gráfico descomposición tiempos de escritura de MemcachedFS

En este caso se observa perfectamente cómo prácticamente el total del tiempo necesario para la operación de escritura de un fichero, se va en la escritura en sí, la apertura es bastante rápida y el cierre es casi inmediato (sólo hay que invalidar los descriptores de fichero, los datos de los ficheros distribuidos no se modifican en absoluto, por tanto, no es necesario comunicarse con la caché distribuida).

De nuevo, al igual que se decidió con el sistema de ficheros local, se toma como tiempo de escritura, la escritura completa en el sistema de ficheros, desde la apertura hasta el cierre. Teniendo en cuenta esta restricción, las tasas de transferencia son las siguientes.

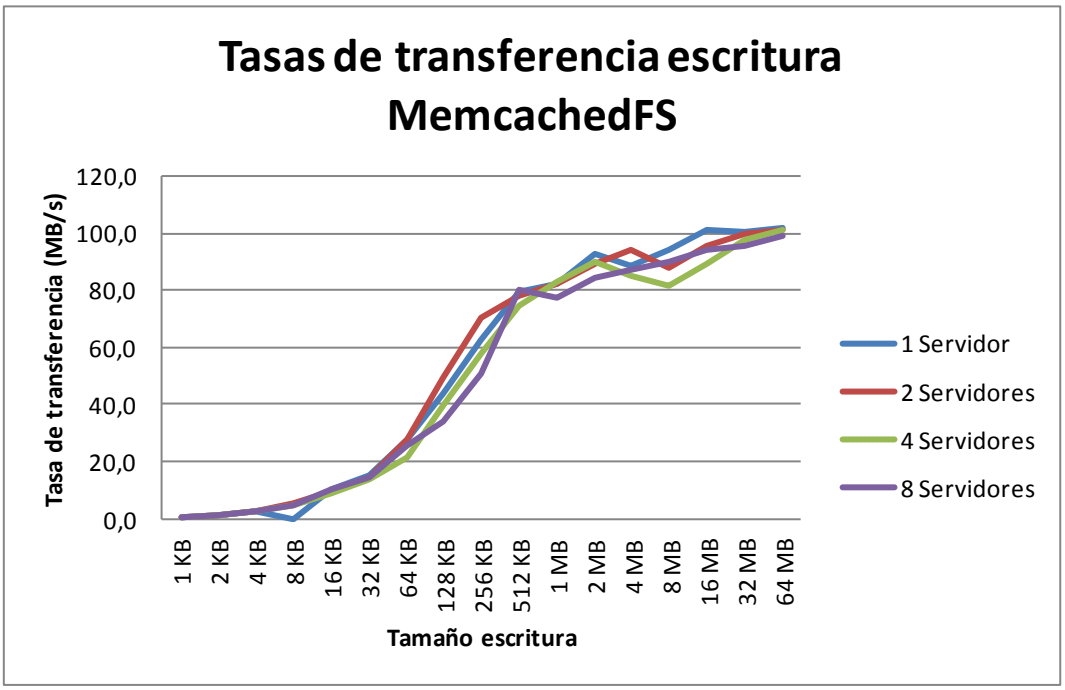

Figura 20: Gráfico de tasas de transferencia de escritura de MemcachedFS

Las principales características del comportamiento que se observan con estos primeros resultados, son muy similares a las observadas al analizar cómo afectaba el tamaño de bloque a la utilización de *Memcached*:

- **Escalabilidad:** una vez implementado el sistema de ficheros, se mantiene la escalabilidad casi perfecta. Las diferencias de rendimiento entre uno y ocho servidores son marginales y difícilmente achacables al hecho de que sean diferente número de servidores, más bien pueden tener relación con picos de carga del sistema.

- **Evolución del rendimiento:** al igual que sucedía con los bloques individuales, el rendimiento aumenta rápidamente a medida que aumenta el

tamaño de la escritura hasta llegar en torno a los 512 Kbyte, tamaño a partir del cual las mejoras son mucho más sutiles, aunque siguen mejorando las transferencias en todos los saltos hasta los 64MB. Obviamente, cuanto mayor es el tamaño del fichero a escribir, más se optimizan las transferencias por la red y, también, tienen un impacto menor las acciones de apertura y cerrado del fichero que, aunque sean pequeñas, se hacen notar ligeramente (unas milésimas podrían parecer despreciables, pero es un ámbito de décimas de segundo, por tanto, se puede observar su impacto, pequeño, pero patente).

- **Aprovechamiento de la red:** al igual que ocurría en la evaluación del tamaño de bloque, se aprovecha al máximo la velocidad de la red, rozando sus límites. Esto indica que, con tamaños de bloque grandes, la administración de los datos del sistema de ficheros distribuidos es muy eficiente, penalizando muy escasamente el rendimiento.

A continuación se exponen los resultados obtenidos mediante la lectura normal, es decir, solicitando a *Memcached* de forma individual cada uno de los bloques que se desea leer. De nuevo las distribuciones de tiempos son muy similares en los cuatro casos. Se mostrarán gráficamente tomando los datos del caso con cuatro servidores como ejemplo:

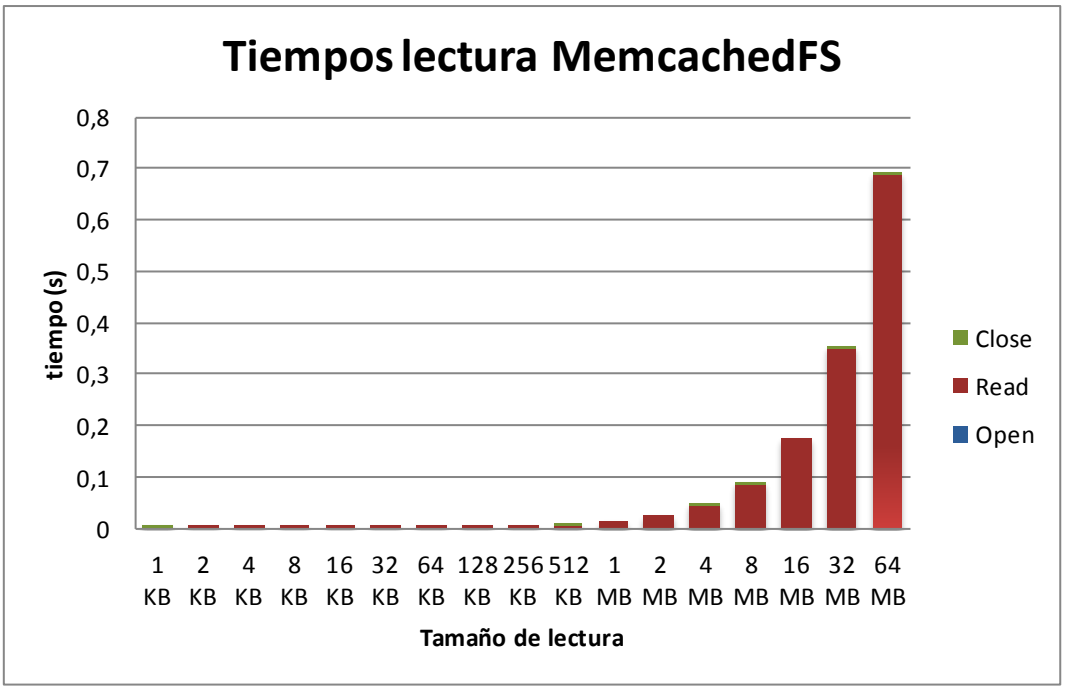

Figura 21: Gráfico descomposición tiempos de lectura normal de MemcachedFS

En este caso, también el grueso del tiempo se lo lleva la lectura en sí, siendo tanto la apertura como el cierre muy cortos. Las tasas de transferencia del proceso completo son las siguientes:

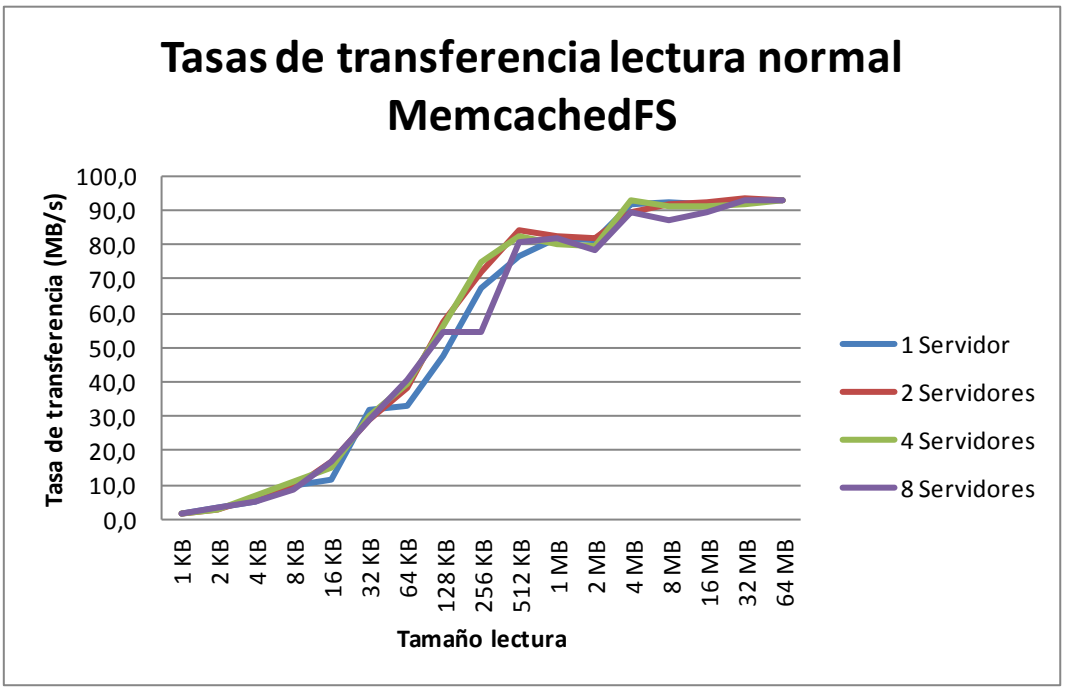

Figura 22: Gráfico de tasas de transferencia de lectura normal de MemcachedFS

Las características que desprenden los resultados sobre el rendimiento de las lecturas son las mismas que en el caso de las escrituras: el sistema escala a la perfección, mejora su rendimiento a medida que aumenta el tamaño del fichero a leer, estabilizándose sobre los 256 Kbyte, y aprovecha al máximo la red. La mayor diferencia con respecto a las escrituras es que no se llega a los 100 MB/s en ningún caso. Los motivos son principalmente dos: mayores tiempos de apertura y menor aprovechamiento de la red.

A continuación, se muestran los resultados de las lecturas múltiples, es decir, aquellas en las que se solicita a *Memcached* todos los bloques completos de un mismo fichero a la vez. De nuevo se tomará como ejemplo el caso de cuatro servidores para mostrar la distribución de tiempos gráficamente:

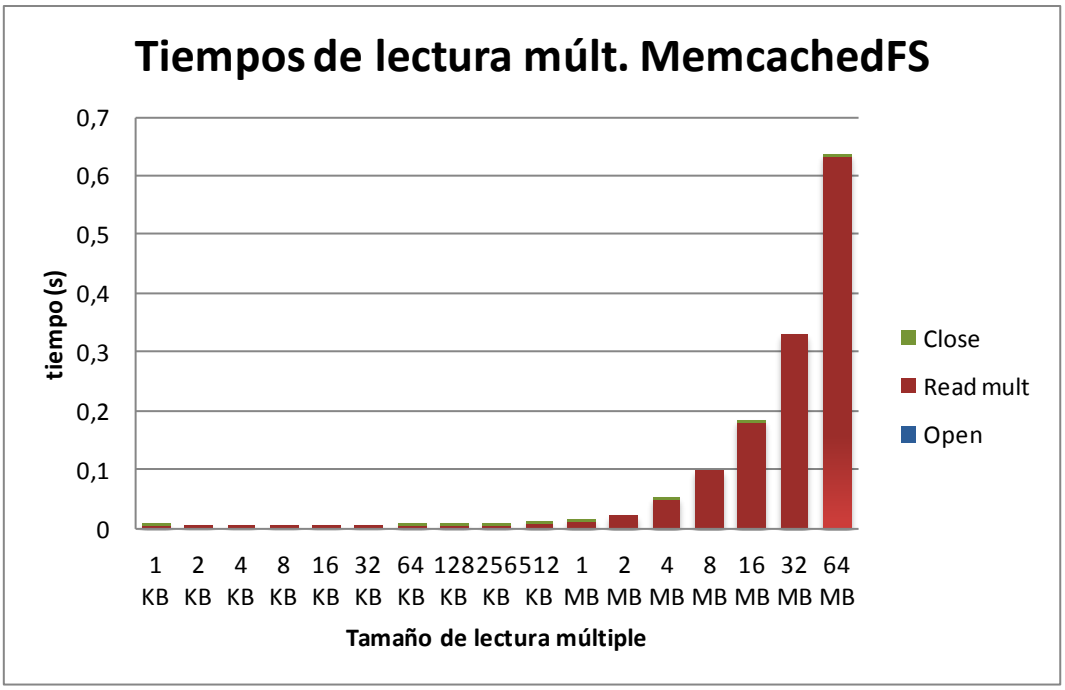

Figura 23: Gráfico descomposición tiempos de lectura múltiple de MemcachedFS

Las distribuciones son muy similares al resto de los casos, tomando la lectura casi todo el protagonismo. Con respecto a las tasas de transferencia, se estudiarán sobre el proceso completo de apertura + lectura + cerrado.

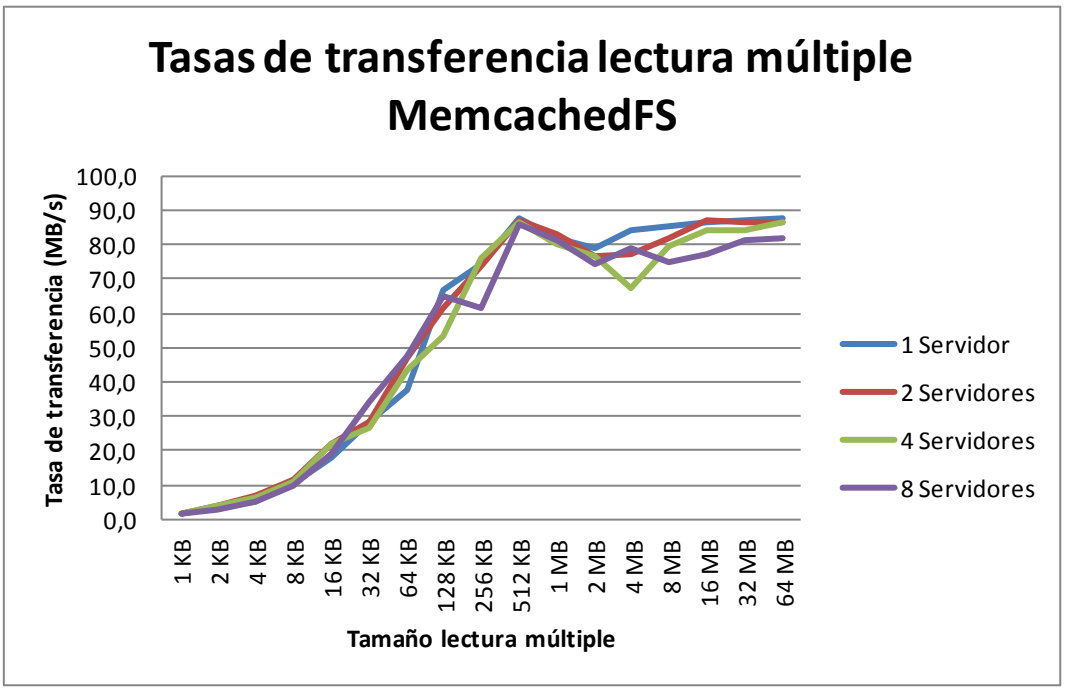

Figura 24: Gráfico de tasas de transferencia de lectura múltiple de MemcachedFS

En este caso, las características cambian ligeramente con respecto los dos casos anteriores. En rasgos generales el comportamiento es similar, sin embargo, las tasas de transferencia se estancan a partir de 512 Kbyte y a duras penas superan los 80 MB/s. El motivo de este cambio de tendencia, es que las lecturas múltiples requieren más procesamiento que las simples. Mientras que en las simples es tan sencillo como pedir el bloque y guardarlo en la zona del buffer que le corresponde, en el caso de las lecturas múltiples, se reciben todos los bloques desordenados y se debe ir buscando uno a uno y colocándolo en su lugar. Por este motivo, hasta 512KB – 1MB (ficheros de un solo bloque) no se nota penalización, pero el rendimiento baja con ficheros de más de un bloque. Hay que tener en cuenta, que en el modo de lecturas simples se alcanzaba casi el límite de transferencia óptimo de la red, por tanto, cualquier procesamiento adicional, siempre repercutirá en el rendimiento. A pesar de todo, las tasas son bastante aceptables, rondando el 80% del máximo.

Otro cambio de tendencia desfavorable es con respecto a la escalabilidad del sistema. Aunque las diferencias son pequeñas, se nota una pequeña pérdida de rendimiento a medida que aumenta el número de servidores utilizado. No es algo determinante como para ajustar el número de servidores al máximo, pero sin duda se puede apreciar claramente en las gráficas.

A continuación, se evaluará el rendimiento con otros tamaños de bloque: 128 Kbyte, 8 Kbyte y 1 Kbyte. En todos estos casos tan sólo se mostrarán los resultados de las tasas de transferencia, ya que, la distribución de los tiempos es similar a lo estudiado anteriormente, con tiempos de apertura y clausura de ficheros muy bajos. Además, a todos los gráficos se añadirá en naranja los valores del caso con tamaño de bloque 1023 Kbyte y 4 servidores (caso medio) para facilitar las comparaciones.

Tamaño de bloque 128 KB

Escritura:

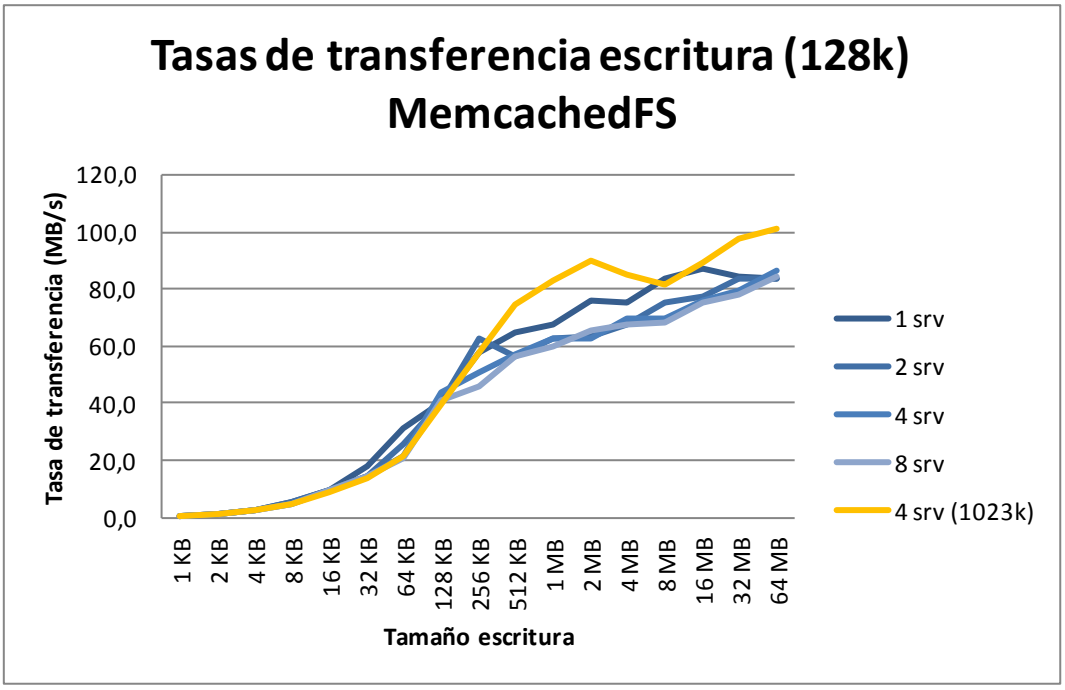

Figura 25: Gráfico de tasas de transferencia de escritura (128k) de MemcachedFS

En este caso, se observa cómo el sistema pierde algo de eficiencia con respecto a los bloques de 1023 Kbyte. Se trata de un resultado esperable, ya que, es necesario realizar más conexiones para escribir el mismo volumen de datos (se deben escribir ocho veces más bloques, por tanto, son ocho veces más llamadas a la función 'set' y, con ello, ocho veces más conexiones con *Memcached*).

La escalabilidad del sistema sigue siendo buena, destacando únicamente sobre el resto el caso de un solo servidor de *Memcached*.

Lectura:

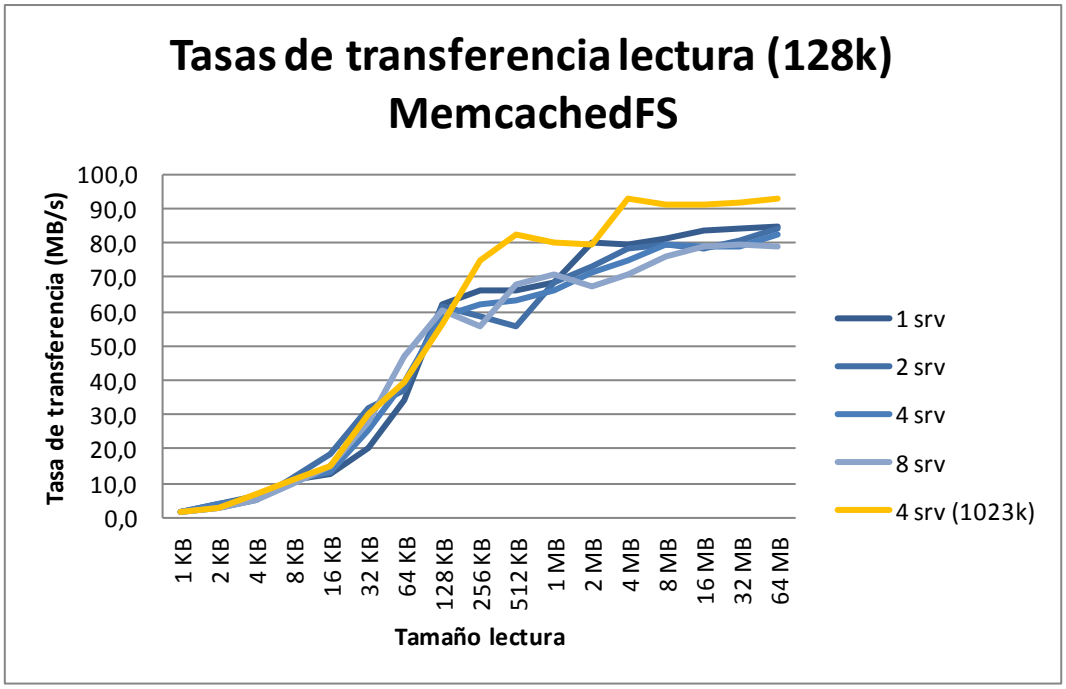

Figura 26: Gráfico de tasas de transferencia de lectura normal (128k) de MemcachedFS

En la lectura simple se observan exactamente los mismos rasgos que en la escritura. Pérdida de rendimiento a causa del mayor número de conexiones necesarias para escribir los bloques más pequeños y rendimiento ligeramente mejor con un solo servidor que con más de uno. Además, en los casos medios (entre 256 Kbyte y 8 Mbyte) se observan fluctuaciones entre los distintos casos con servidores múltiples. No se puede concluir que el rendimiento decaiga a medida que aumente el número de servidores, pero sí se observa un rendimiento más irregular.

Lectura múltiple:

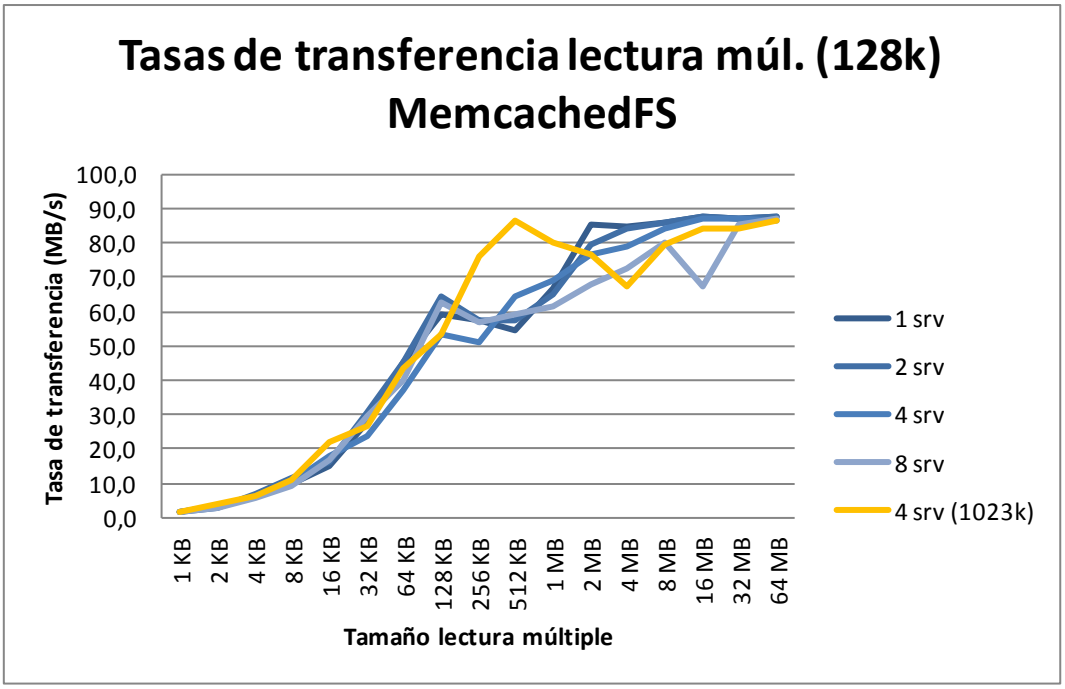

Figura 27: Gráfico de tasas de transferencia de lectura múltiple (128k) de MemcachedFS

Este caso es el más interesante de los tres. Gracias a la utilización de las lecturas múltiples se reduce el número de conexiones y se mejoran las tasas de transferencia, igualando las obtenidas con lecturas múltiples con bloques de 1023 Kbyte y mejorando las lecturas simples de 128 Kbyte, aunque aún por detrás de las lecturas simples de 1023 Kbyte. Se observa por primera vez cómo se compensa el tiempo adicional necesario para procesar las lecturas múltiples con el tiempo ahorrado en conexiones.

Como en el caso de bloques de 1023 Kbyte, existe un punto de inflexión en el momento en que el tamaño de fichero iguala el tamaño de bloque, ya que, sólo se requiere una conexión para leerlo. A partir de ese punto baja el rendimiento pero se va recuperando poco a poco hasta estabilizarse en torno a los 2 Mbyte, consiguiendo tasas incluso de 80 MB/s, similares a ese mismo tamaño de fichero en lecturas simples de 1023 Kbyte.

Como punto negativo, las lecturas múltiples vuelven a penalizar el rendimiento a medida que se aumenta el número de servidores, al menos en escrituras entre 1 Mbyte y 16 Mbyte. En el resto de valores se observa un rendimiento muy irregular, pero comparable en la mayoría de los casos al ofrecido con menor número de servidores.

Tamaño de bloque 8 KB

Se trata de un tamaño más similar a lo que se suele encontrar en sistemas de ficheros locales, por tanto, su comportamiento es interesante de estudiar.

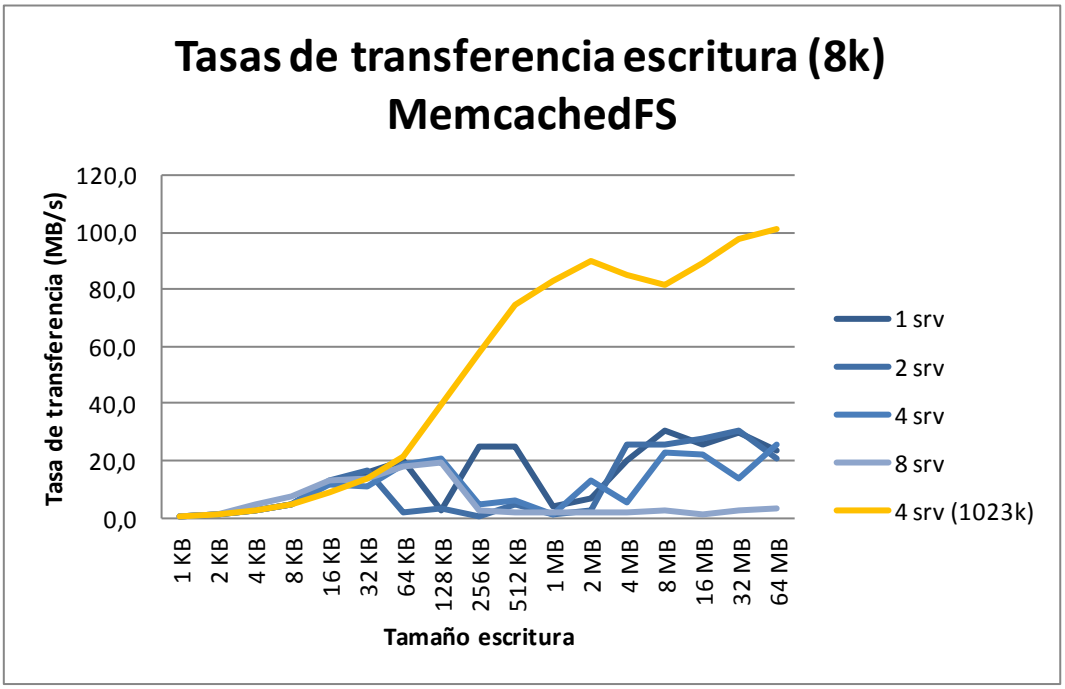

Figura 28: Gráfico de tasas de transferencia de escritura (8k) de MemcachedFS

Las conclusiones que se pueden sacar de los resultados son, en parte, esperadas. A medida que se reduce el tamaño de bloque, aumenta el número de conexiones necesarias, pero en este caso se nota una penalización especialmente fuerte. El rendimiento se mantiene estable hasta los 128 Kbyte, pero a partir de ese punto, el rendimiento es completamente inestable y muy pobre. Además, la diferencia entre un servidor y cuatro servidores no es muy acusada, pero el rendimiento con ocho servidores se desploma por completo, siendo insuficiente para la mayoría de los usos.

Lectura:

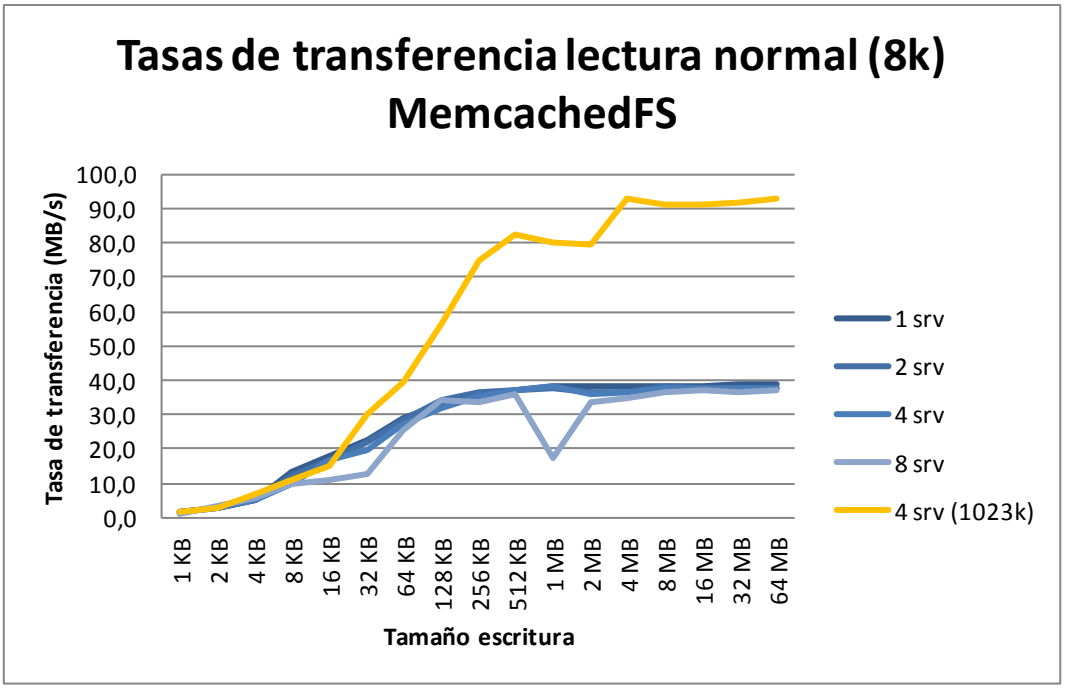

Figura 29: Gráfico de tasas de transferencia de lectura normal (8k) de MemcachedFS

Los resultados obtenidos en este caso son más esperados y más favorables. Las lecturas mejoran a medida que aumenta el tamaño de bloque hasta estabilizarse entre los 64 Kbyte y los 128 Kbyte. Las tasas obtenidas no son muy altas, pero son muy estables, por tanto, un comportamiento admisible para casos en los que interese un tamaño de bloque bajo por algún motivo concreto.

Otro punto positivo es que la escalabilidad se mantiene casi perfecta, salvo algún caso inestable con ocho servidores. En casi todos los casos el rendimiento disminuye a medida que se aumenta el número de servidores, pero la diferencia es muy pequeña y es compensada con creces por las ventajas de la escalabilidad.

Lectura múltiple:

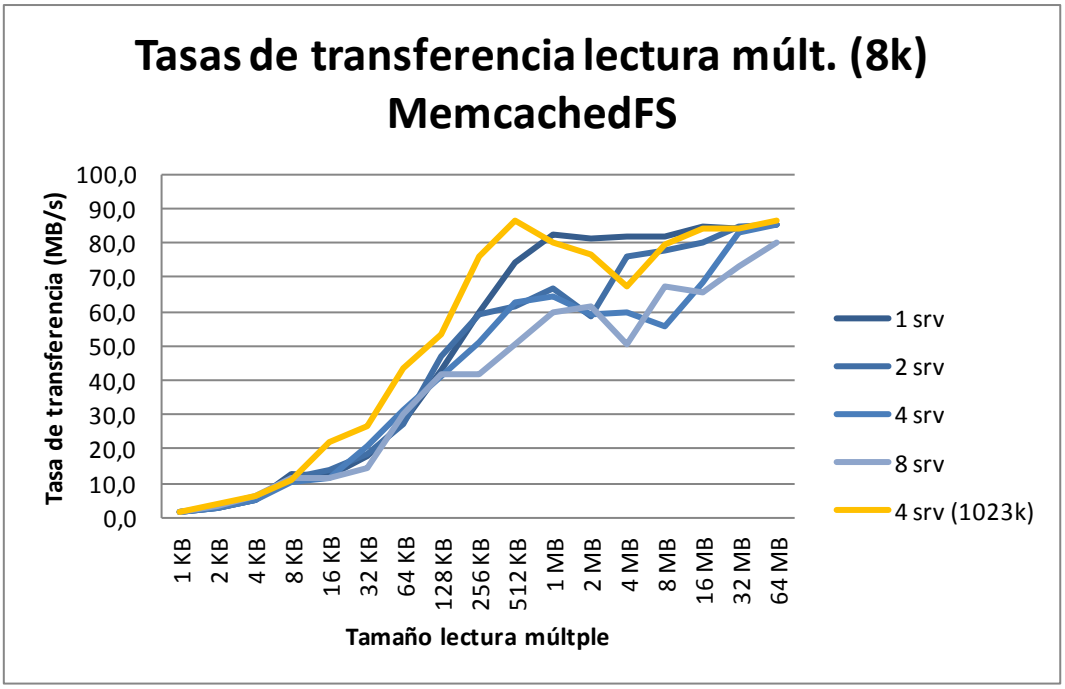

Figura 30: Gráfico de tasas de transferencia de lectura múltiple (8k) de MemcachedFS

El resultado obtenido en este caso vuelve a ser muy positivo, casi sorprendente. Gracias a la minimización de las conexiones, el rendimiento ofrecido por las lecturas múltiples de bloques de 8 Kbyte es casi comparable con el que ofrecen las de bloques de 1023 Kbyte. También se vuelve a observar una penalización en la escalabilidad a partir de escrituras de 256 Kbyte o superiores, pero afortunadamente, en ningún caso excesivamente graves.

Tamaño de bloque 1 KB

Este caso de estudio se plantea, principalmente, por ser el valor extremo de menor tamaño de bloque. Viendo los resultados obtenidos con los bloques de 8 Kbyte, se puede intuir que la caída de prestaciones seguirá su curso, pero es interesante estudiar el comportamiento extremo del sistema.

Escritura:

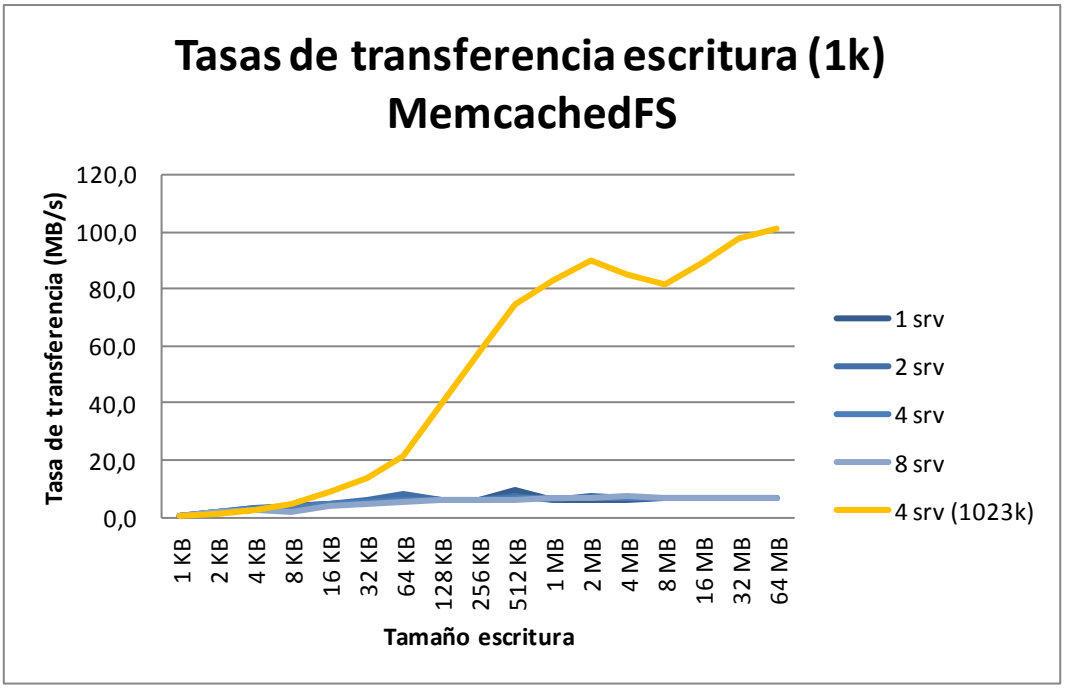

Figura 31: Gráfico de tasas de transferencia de escritura (1k) de MemcachedFS

Como se podía esperar, el rendimiento baja hasta niveles realmente bajos, a la altura de *pen drives* o tarjetas de memoria de baja calidad, muy lejos de las prestaciones esperadas de un sistema de alto rendimiento. El único punto positivo que se puede extraer de esta evaluación, es que el sistema escala de forma casi perfecta, aunque también es cierto que una desviación de un 10% en este caso no supone ni siquiera una diferencia de un MB/s, por tanto, es más fácil disimularlas.

Lectura:

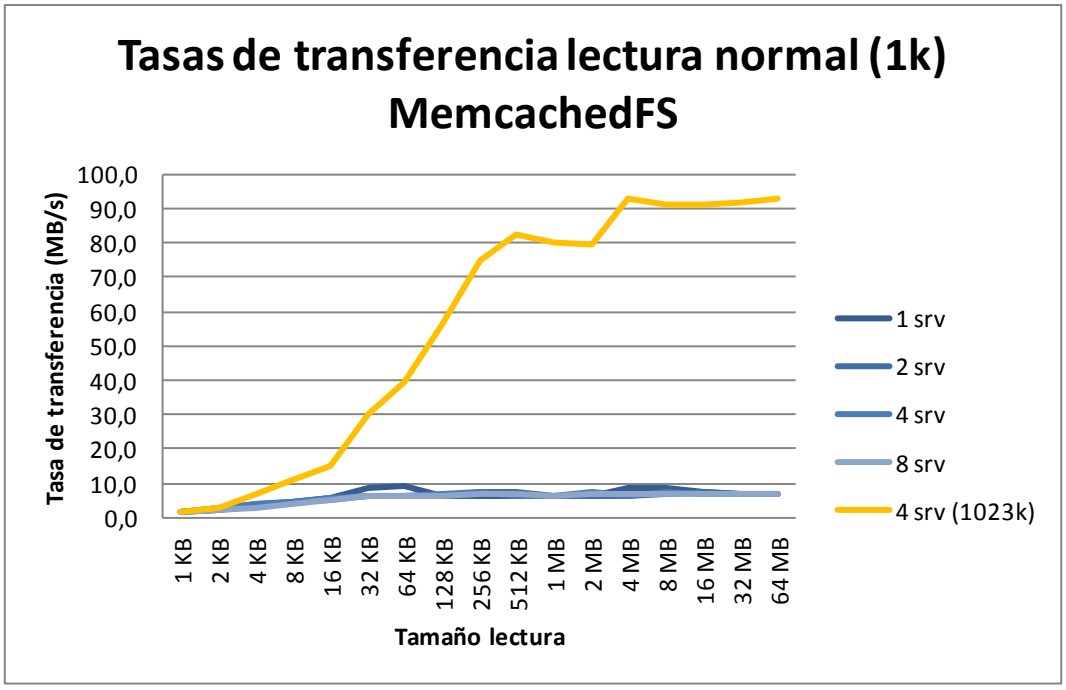

Figura 32: Gráfico de tasas de transferencia de lectura normal (1k) de MemcachedFS

Las conclusiones que se pueden extraer de estos resultados son muy similares a las anteriores. Un rendimiento muy pobre y una escalabilidad relativamente buena (salvo el caso de un solo servidor, que destaca sobre el resto).

Lectura múltiple:

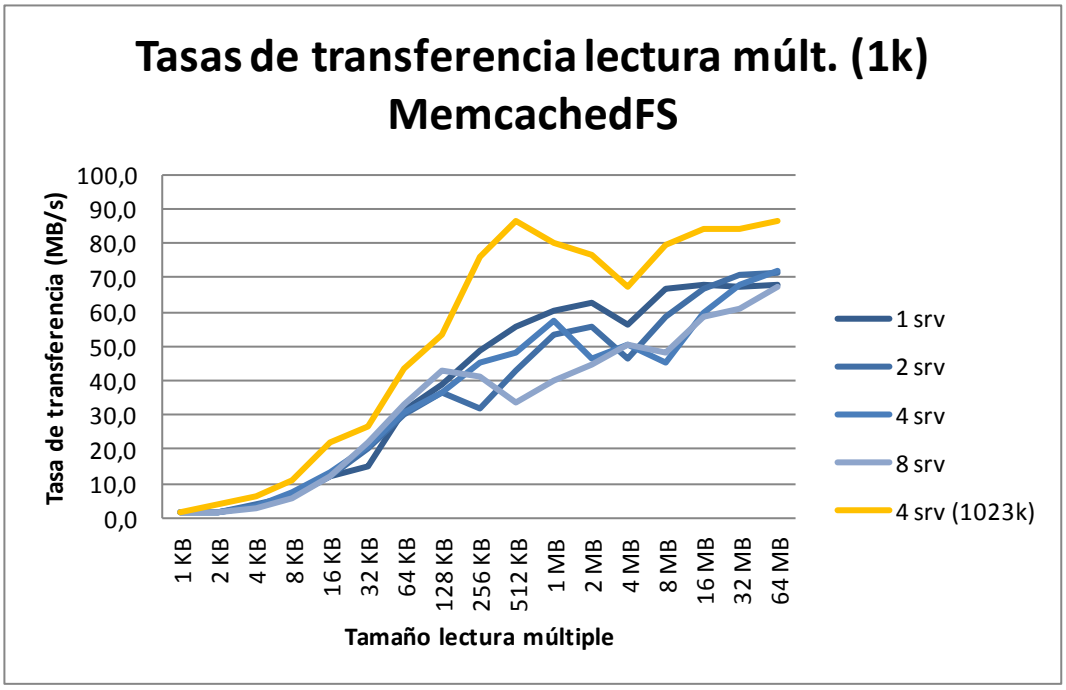

Figura 33: Gráfico de tasas de transferencia de lectura múltiple (1k) de MemcachedFS

Este último caso es, una vez más, el más interesante de todos. A pesar de que no llega a las tasas de transferencia del caso con bloques de 1023 Kbyte, mantiene unas tasas de transferencia muy aceptables, convirtiéndose incluso en una opción válida de cara a que las necesidades inviten a utilizar tamaños de bloque muy pequeños. Con respecto a la escalabilidad, se observa dominación del caso con un solo servidor, lastrándose el rendimiento cuando se añaden servidores y volviéndose bastante inestables los resultados (no se puede asegurar fehacientemente que el aumento de servidores esté relacionado directamente con una bajada del rendimiento, pero es cierto que el caso de los ocho servidores está casi en todas las lecturas por debajo del resto).

8.4 Lectura/escritura Sistema Ficheros Local vs MemcachedFS

En este apartado se compara el rendimiento entre el sistema de ficheros distribuido desarrollado y un sistema de ficheros local. Lo primero que se va a estudiar, son las escrituras. Para ello, se utilizarán los datos obtenidos anteriormente, es decir, escrituras completas de ficheros de entre 1 Kbyte y 64 Mbyte.

En el siguiente gráfico, los valores del sistema de ficheros distribuido están ordenados por colores claros en función del tamaño de bloque, utilizando la misma tonalidad para los casos con el mismo tamaño de bloque y marcando ligeramente más fuerte el caso medio, con cuatro servidores. A su vez, los valores del sistema de ficheros local son resaltados en naranja para facilitar la comparación.

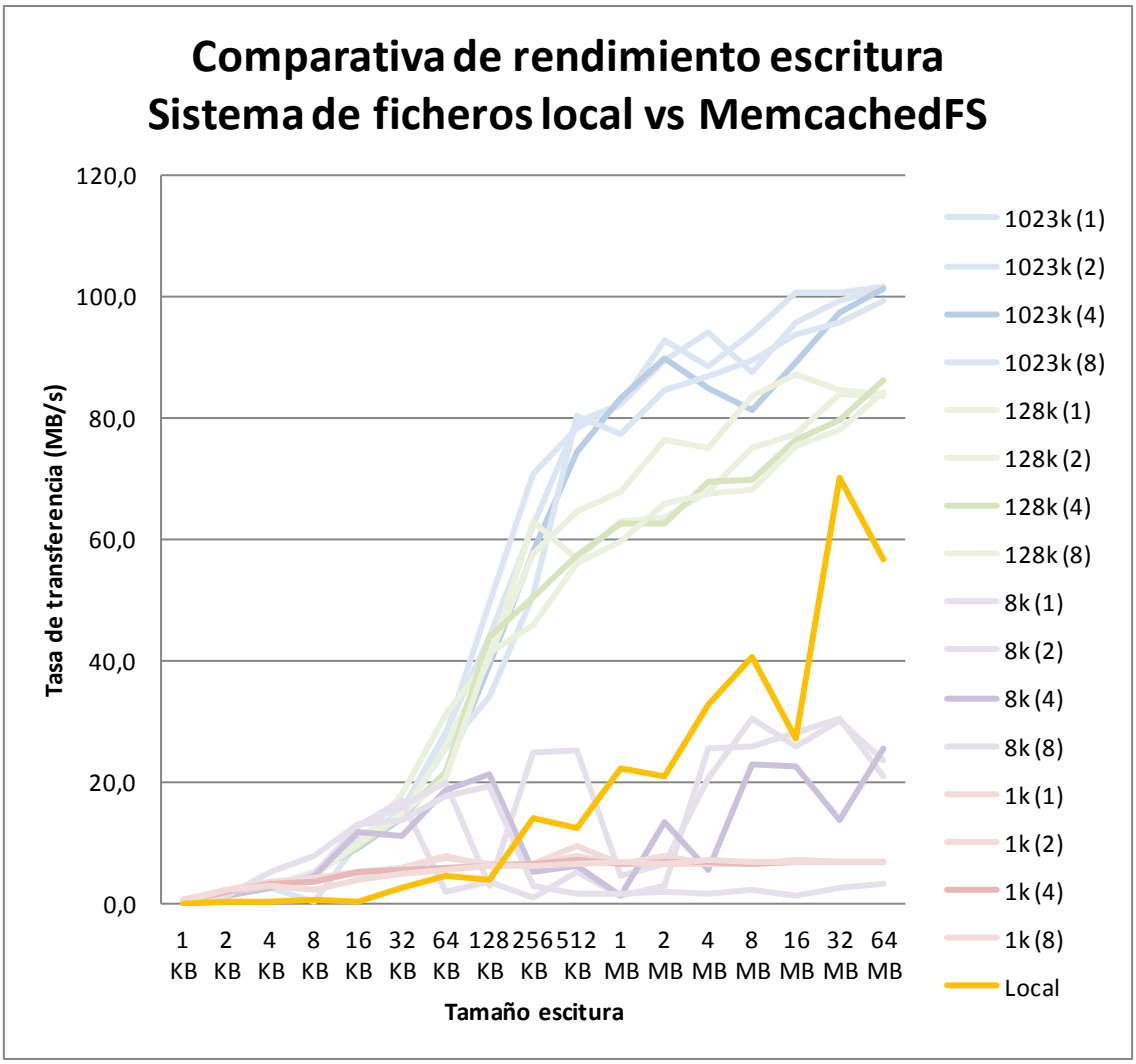

Figura 34: Gráfico comparativo de rendimiento en escritura entre MemcachedFS y un disco local

La característica más interesante que se puede observar en este primer gráfico es que, en escrituras por debajo de 32 Kbyte, el sistema de ficheros distribuido es muy superior en todas sus modalidades. De este modo, se pone de manifiesto la dificultad de los discos duros tradicionales para operar en modos no secuenciales, con lecturas pequeñas.

A partir de ese tamaño de escritura empiezan los problemas para el peor de los casos del sistema de ficheros distribuido (bloques de 1 Kbyte). Sin embargo, el modo con bloques de 8Kbyte, a pesar de ser peor que el sistema local, se mantiene competitivo.

Con tamaños de bloque grandes la diferencia es completamente abismal, desde escrituras pequeñas hasta grandes ficheros de datos. Además, cabe destacar que el sistema de ficheros distribuido está limitado por la red (llega a sus límites en los casos extremos), por tanto, tiene margen de mejora, mientras que el rendimiento del sistema de ficheros local viene dado por el dispositivo, y solo podría mejorarse recurriendo a

dispositivos de estado sólido (SSDs) o a combinaciones de discos (por ejemplo, RAIDs).

El siguiente paso en el estudio serán las lecturas normales, siguiendo el mismo procedimiento anterior. A continuación, las tasas de transferencia con las tasas del sistema de ficheros local y con el sistema de ficheros distribuido en todas sus posibles configuraciones.

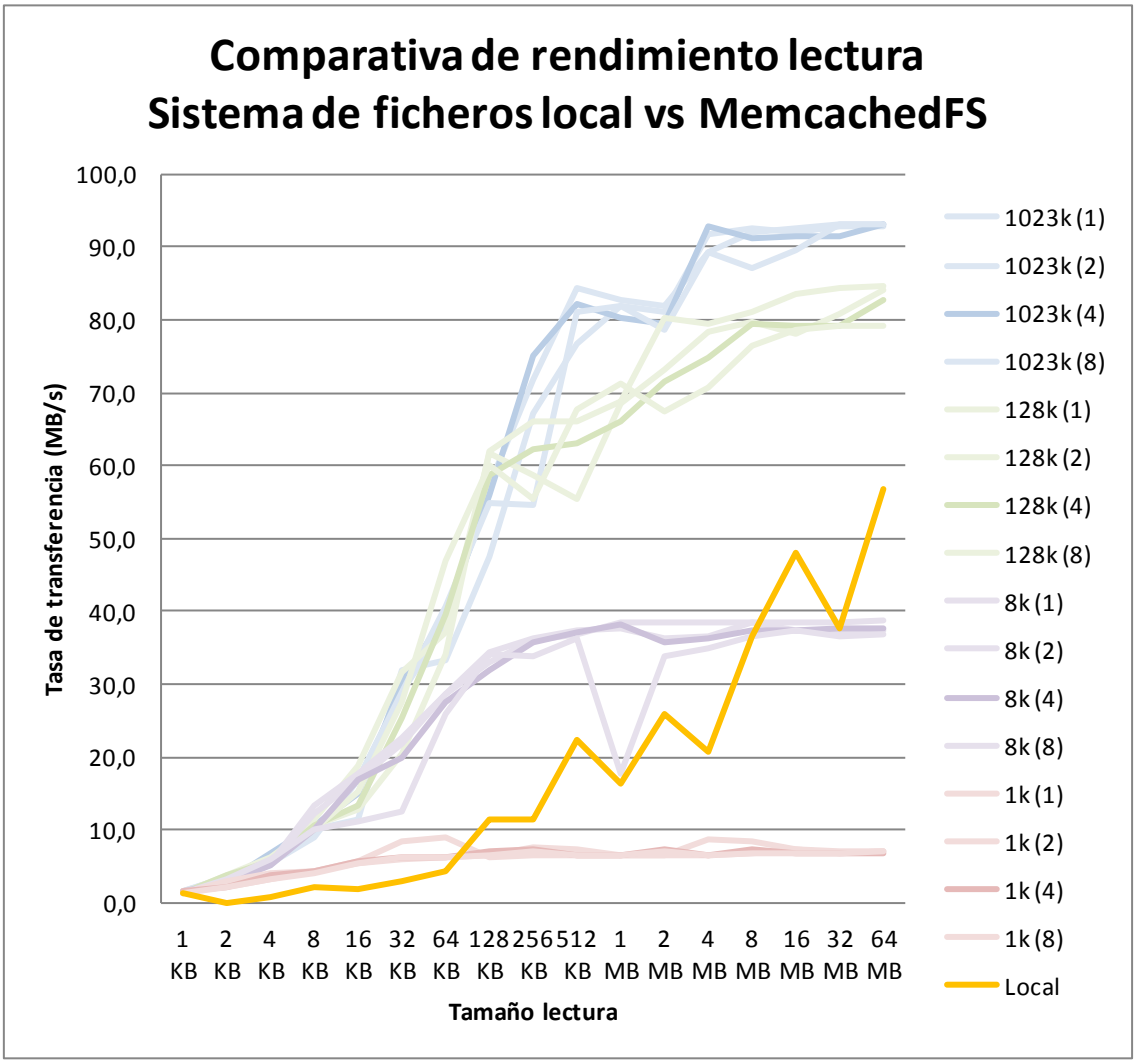

Figura 35: Gráfico comparativo de rendimiento en lectura entre MemcachedFS y un disco local

En este caso, se vuelve a observar que las lecturas hasta tamaños de 32 Kbyte, son mucho más rápidas en el sistema de ficheros distribuido, sea cual sea su tamaño de bloque. Es decir, se vuelve a observar la laguna que tienen los sistemas mecánicos con las operaciones de entrada y salida que no son secuenciales de gran tamaño.

También se vuelve a observar que la velocidad del sistema de ficheros distribuido con tamaños de bloque grandes es muy superior al local, sin embargo,

aparece una novedad destacable. Incluso con tamaños de bloque de 8 Kbyte, el sistema de ficheros distribuido es más competitivo en la mayoría de lecturas que el local. Sólo el peor caso del sistema de ficheros distribuido es incapaz de hacer frente al sistema local en tamaños de lectura a partir de 32 Kbyte.

Es interesante destacar, una vez más, que el rendimiento del sistema de ficheros distribuido depende enormemente de la red, por tanto, una mejora de la misma podría traducirse fácilmente en una mejora de su rendimiento.

A continuación, se estudiarán las lecturas múltiples del sistema de ficheros distribuido en comparación con las tasas de transferencia que ofrecen las lecturas del sistema de ficheros local (que son las mismas que en el caso anterior, ya que, sólo existe un modo de lectura no cacheada en los discos duros).

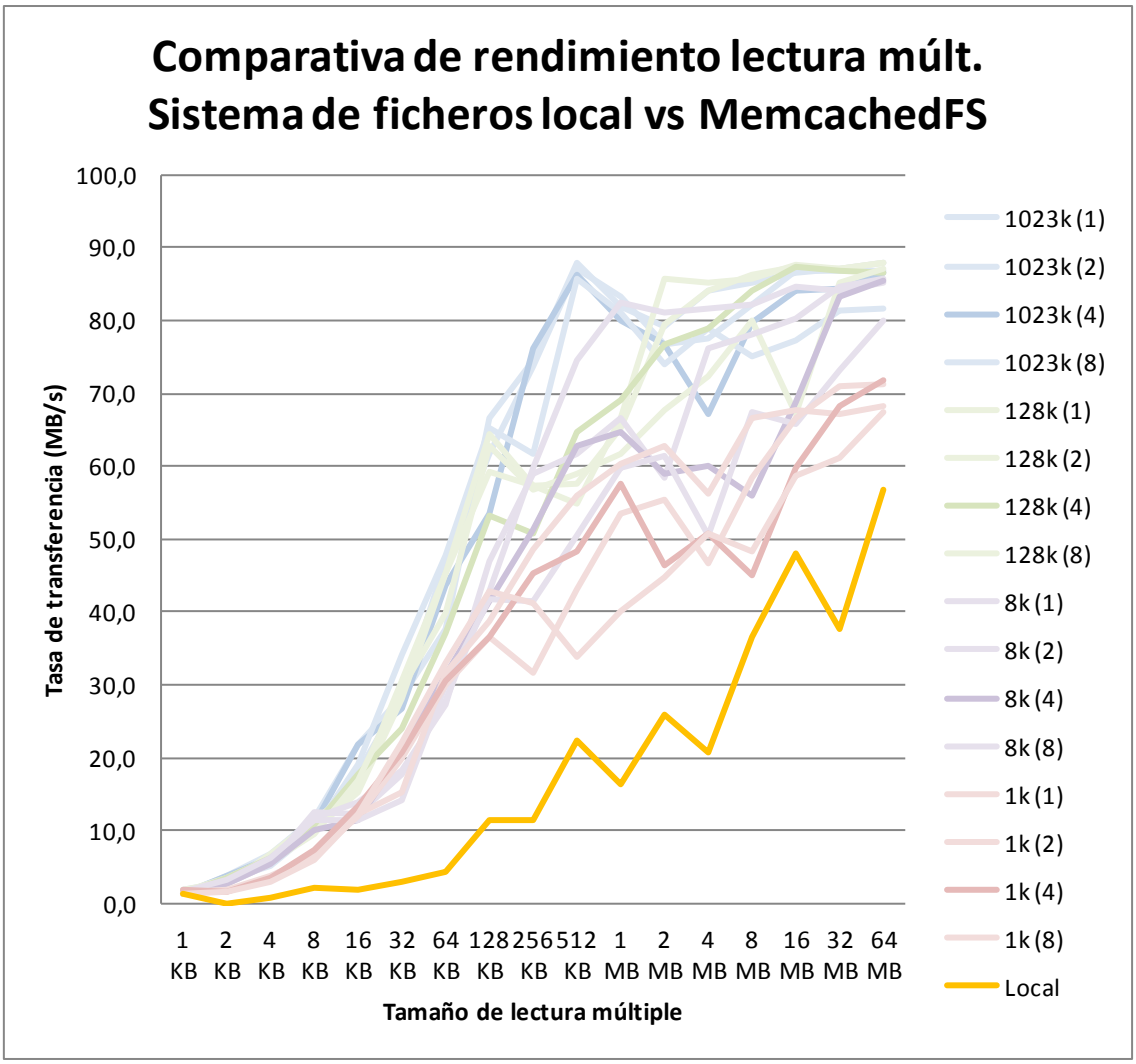

Figura 36: Gráfico comparativo de rendimiento en lectura múltiple entre MemcachedFS y un disco local

En esta ocasión el dominio del sistema de ficheros distribuido es total. En todos los tamaños de ficheros, con cualquier tamaño de bloque y con cualquier configuración de servidores. Se hace patente, por tanto, lo que se ha comentado en varios lugares de este documento: *Memcached* está pensado principalmente para acelerar las lecturas, es decir, utilizarlo como caché de aplicaciones web con muchas consultas de ciertos elementos. Si bien, en el caso de los bloques de 1023 Kbyte se pierde algo de rendimiento con respecto a las lecturas simples, es recomendable en la mayoría de los casos utilizar la función `read_mc_mult` en lugar de `read_mc`, ya que, por norma general mejora ampliamente el rendimiento.

Sin embargo, esto solo es así con lecturas que superen el tamaño de bloque del sistema de ficheros distribuido (habitualmente, a partir de varias veces su tamaño), ya que, las lecturas que no lo superan se hacen exactamente igual. El modo de lectura múltiple solo se aprovecha a la hora de pedir a *Memcached* bloques completos de datos. Esta característica se puede ver absolutamente en todos los datos recogidos, en los que

las lecturas por debajo del tamaño de bloque tienen unos tiempos muy similares a las lecturas simples del mismo tamaño.

En ningún momento se roza el límite máximo de transferencia de la red, por tanto, no se puede afirmar rotundamente que el rendimiento mejorase al mejorarse la capacidad de la red, pero es muy probable, haciendo de este sistema una verdadera alternativa con respecto a los discos duros tradicionales en sistemas de alto rendimiento para tareas que requieran gran cantidad de lecturas. Estará especialmente recomendado para ficheros grandes pero con lecturas por debajo de los 64 Kbyte, donde la diferencia con el sistema de ficheros local es especialmente apreciable.

Con respecto al tamaño óptimo de bloque para el sistema de ficheros distribuido desarrollado, una vez vistos todos los resultados, parece que el valor más equilibrado es el de 128 Kbyte. Sin embargo, 8 Kbyte también sigue siendo bastante competitivo y puede aportar enormes ventajas en lecturas muy pequeñas (para leer un byte de datos no será necesario transferir por la red 128 Kbyte, sino tan sólo 8 Kbyte).

Si todas las lecturas van a ser de grandes tamaños, por ejemplo, sistemas multimedia, un tamaño de bloque grande (1023 Kbyte) y lecturas normales (`read_mc`) pueden ser más eficientes.

En definitiva, el sistema de ficheros distribuido desarrollado, se muestra competitivo con respecto a un sistema de ficheros local en un disco duro tradicional, especialmente en lecturas, y muy flexible, pudiendo ser aplicado a sistemas muy diversos obteniendo buenos resultados si se configura adecuadamente. Además, aporta la ventaja de ser distribuido y poder ser utilizado por varios usuarios de forma simultánea (aunque no se garantiza el correcto funcionamiento en accesos simultáneos a un mismo fichero).

8.4.1 Modos Caché

Se han implementado dos modos de caché: pre-caché y caché. Sin embargo, dada la metodología de estos procedimientos de evaluación, no es adecuado estudiar sus resultados en profundidad. La explicación es sencilla, el objetivo de estos modos es, bien cachear pequeñas porciones de datos para acceder rápidamente a ellas (modo caché) o bien cachear un fichero completo para luego acceder más rápido a él en operaciones complejas.

En este caso, ninguno de esos dos objetivos es perseguido. Se mide la velocidad pura de las operaciones de lectura y escritura de ficheros, no los accesos a datos concretos. Para entender mejor la falta de eficacia de esta evaluación en esos casos, se detallará el comportamiento del modo Pre-caché para realizar el procedimiento de evaluación: en primer lugar se abre el archivo y se carga, por completo, en la caché distribuida, en segundo lugar se realiza la lectura/escritura (como en los casos del sistema de ficheros distribuido realizadas anteriormente) y, por último, se vuelve a

copiar el contenido del fichero desde la caché distribuida hacia el disco duro. Por tanto, además de realizarse la operación de lectura/escritura que se pretende medir, se debe leer completamente de disco el fichero, escribirlo en caché distribuida, leerlo de caché distribuida y volverlo a escribirlo en el disco. Cinco operaciones para medir el rendimiento de una sola de ellas.

Aún así, a continuación se muestran las distribuciones de tiempo de un caso elegido como promedio analizando los resultados obtenidos en las evaluaciones: tamaño de bloque de 128 Kbyte y cuatro servidores de *Memcached*.

Escritura:

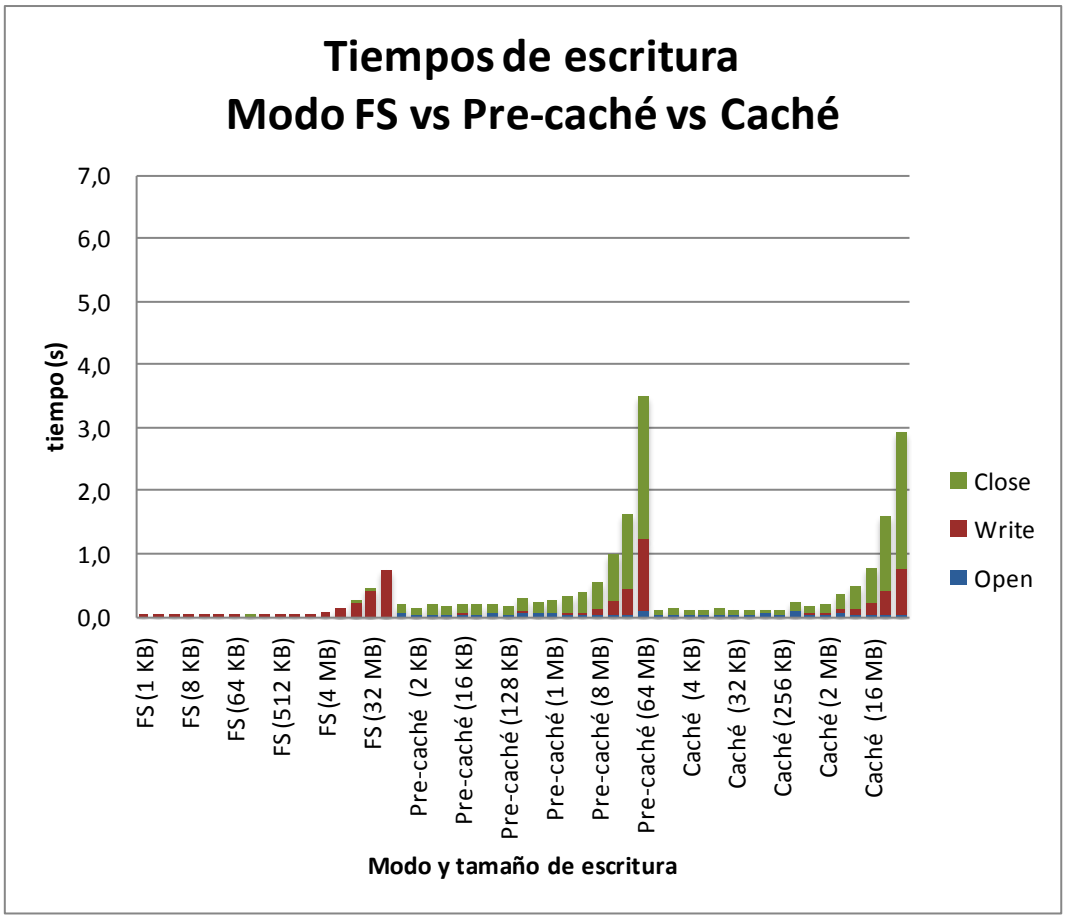

Figura 37: Gráfico de distribución de tiempos en escritura. Modo FS vs Pre-caché vs Caché

El caso de la escritura es muy interesante, ya que, se observa a la perfección el ejemplo que se ha puesto como introducción. Los tiempos de escritura, se mantienen exactamente iguales en los tres casos, sin embargo, no ocurre así en con los tiempos de apertura y cerrado del fichero. En la apertura, en el modo caché es muy pequeña (los archivos se crean vacíos en caché distribuida y se cargan los datos según se van necesitando), mientras que en el modo pre-caché hay que comprobar si el fichero existe

para llevarlo a caché distribuida y, si no existe, crearlo (en este caso no se produce ninguna transferencia porque el fichero no existe).

Acto seguido, la operación de escritura es prácticamente igual en los tres modos y, es por ello que tardan los mismo. La única diferencia existe en el modo Caché, que hay que recorrer la tabla para marcar las entradas como escritas, pero la operación es muy sencilla, ya que, tan solo hay que recorrer un array en memoria principal.

Por último está la operación de cerrado, donde realmente se nota la penalización de rendimiento, ya que, se debe leer todo el fichero de la caché distribuida (en modo caché también, ya que, se ha modificado el fichero entero y hay que actualizarlo) y escribirlo en disco actualizado.

El modo caché realiza esta operación mediante lecturas simples (se comprueba bloque a bloque si debe ser devuelto a disco), mientras que en modo pre-caché se hace con lecturas múltiples de varios bloques. El rendimiento de las lecturas simples y múltiples con 128 Kbyte era muy similar, y aquí se observa cómo ambos tiempos de cerrado son muy similares.

Se nota una pequeña imprecisión en el caso de los 64 Mbyte del modo pre-caché, tanto la escritura como el cerrado tardan más de lo esperado, debido a alguna imprecisión en la prueba, probablemente por algún pico de carga del clúster.

En definitiva, la conclusión que se puede sacar es que el proceso de escritura se realiza exactamente igual y su rendimiento es, por tanto, equivalente. La diferencia reside principalmente en la apertura y cerrado de los archivos, que es especial para poder utilizarlos como caché, lo cual penaliza el rendimiento para este tipo de operaciones.

A continuación, se realiza un procedimiento similar, en este caso, midiendo el rendimiento completo de la lectura de un fichero mediante lecturas simples.

Lectura:

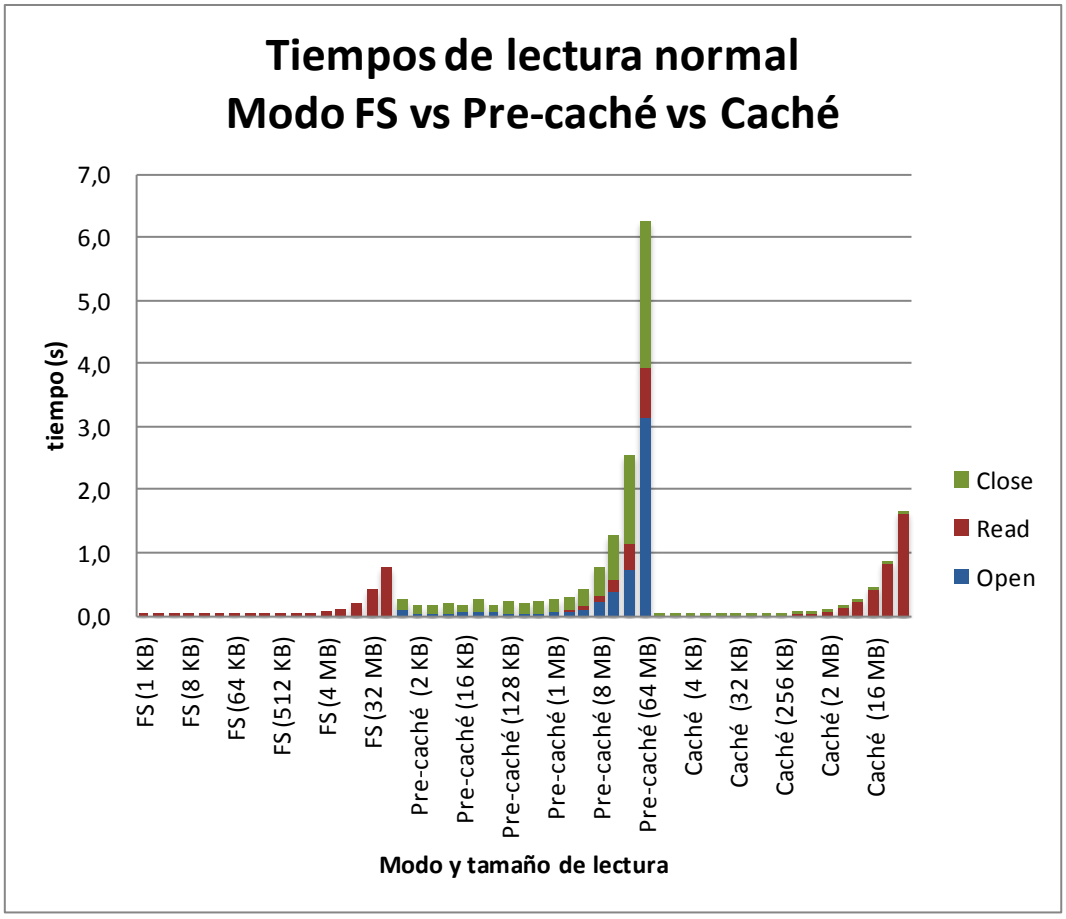

Figura 38: Gráfico de distribución de tiempos en lectura normal. Modo FS vs Pre-caché vs Caché

En primer lugar se analizará el modo caché, ya que, sus resultados son más claros e ilustradores. Como se puede observar, las operaciones de abrir y cerrar el fichero consumen un tiempo pequeño tal y como ocurre con el modo normal del sistema de ficheros distribuido. Esto es así por una razón simple, en la apertura en modo caché se realiza una operación muy similar al modo normal, es decir, se crea el archivo en la caché distribuida y se le ponen sus atributos, que en este caso serán del archivo existente en local en lugar de atributos nuevos como en el modo normal.

Por otro lado, en el cerrado del fichero no hay que realizar ningún esfuerzo, ya que, todas las operaciones realizadas sobre los ficheros son de lectura. Si el fichero no es modificado, no es necesario actualizarlo en el disco duro local. Por tanto, el único procesamiento adicional con respecto a un cerrado normal es recorrer un array en memoria principal comprobando que no hay ningún bloque marcado como modificado.

Por último, se observa que el tiempo de lectura es aproximadamente el doble que en modo normal. La explicación es sencilla, para realizar una lectura en modo caché, primero hay que trasladar la parte del fichero que quiere ser leída a memoria distribuida en caso de que no esté (trabajando bajo demanda, como una caché en la que se da un

"fallo caché") y, acto seguido, leerla. Es decir, una operación de lectura y una de escritura es normal que consuma aproximadamente el doble de tiempo que una lectura simple. Es interesante apuntar, que una vez se encuentran los datos en caché distribuida (tras la primera lectura de un bloque), el modo caché comenzaría a dar "aciertos caché" y los tiempos de lectura deberían ser equivalentes a los del modo normal.

Con respecto al modo pre-caché, los resultados son los esperados también. Los modos de apertura y cerrado del fichero tardan bastante tiempo y que, además, dicho tiempo es aproximadamente el mismo (se debe leer el fichero de disco y volcarlo a caché distribuida en un caso, y lo contrario en el otro). Y, por último, las lecturas tardan aproximadamente el mismo tiempo que en modo normal. Esto es así porque el proceso de lectura es exactamente el mismo. A diferencia del modo caché en el que hay que llevar el bloque a caché distribuida antes de leerlo, en este caso se lleva durante la apertura, por lo que la lectura no es penalizada de ninguna manera con respecto al modo normal.

Lectura múltiple:

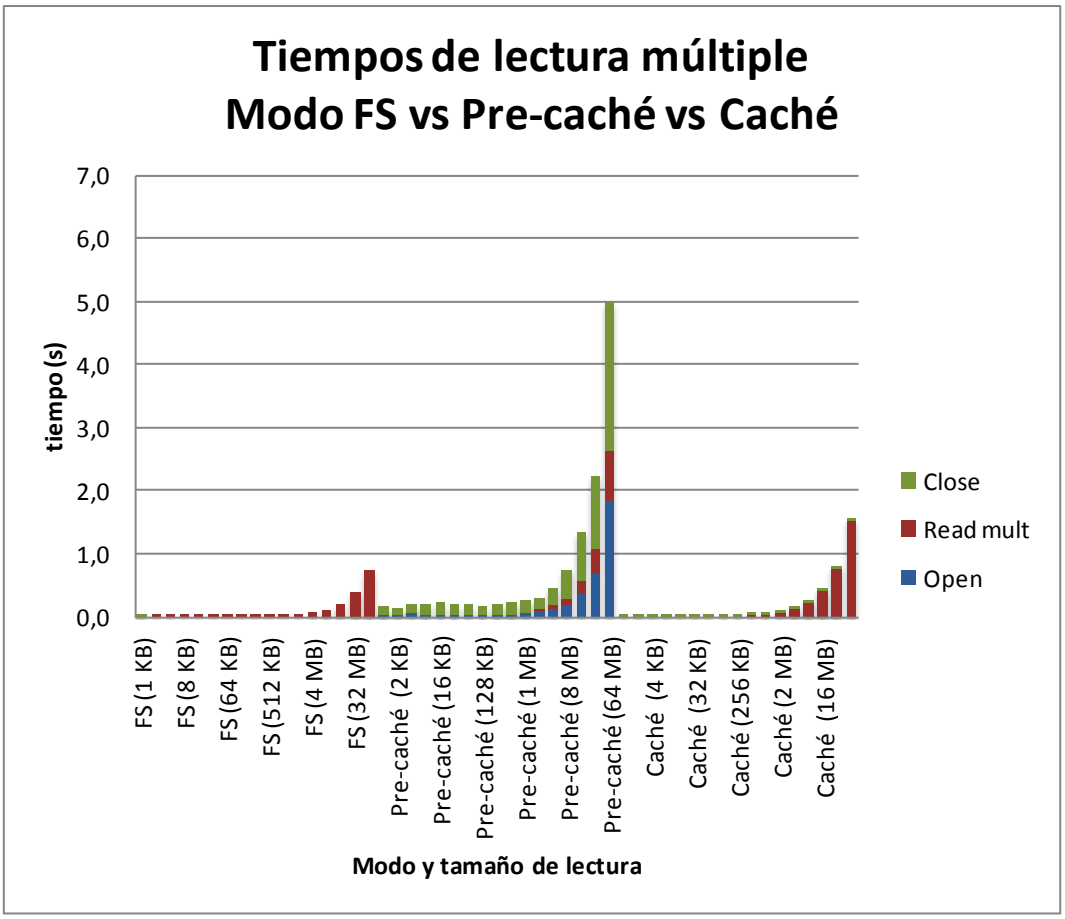

Figura 39: Gráfico de distribución de tiempos en lectura múltiple. Modo FS vs Pre-caché vs Caché

Dado que el rendimiento de las lecturas múltiples con bloques de 128 Kbyte no difiere mucho de las lecturas simples, las características que se observan en este caso son exactamente las mismas que en el anterior.

Resultados lógicos tanto en el modo caché como en el modo pre-caché. Ambos lastrados por un uso inapropiado para sus características.

NOTA: Cuando se habla de leer de disco o escribir en disco durante esta evaluación, dado que se está trabajando sobre un clúster que utiliza NFS para montar las cuentas de usuario, realmente las lecturas y escrituras se hacen sobre NFS en modo remoto. En cualquier caso, siempre que se hable de disco, se referirá al sistema de ficheros sobre el que esté ejecutándose el cliente.

8.5 NFSv4 y PVFS2 vs MemcachedFS

Para evaluar el rendimiento, no solo contra un sistema de ficheros local, sino también con otros sistemas de ficheros distribuidos, se han realizado evaluaciones con NFSv4 y PVFS2.

El procedimiento seguido es el mismo que en el caso del sistema de ficheros local. Para las escrituras, se crean ficheros de datos aleatorios de entre 1 Kbyte y 64 Mbyte de tamaño y, para las lecturas, se leen esos mismos ficheros creados. La medición se hace sobre el proceso completo de apertura, escritura y cerrado del fichero y se enfrentará a todos los casos de MemcachedFS (distinto número de servidores de *Memcached* y tamaño de bloque).

8.5.1 NFSv4

Se comenzará la evaluación con el sistema más utilizado actualmente, NFS (Network FileSystem). Las características del sistema de NFS montado sobre el clúster son las siguientes:

- NFS versión 4.
- 512 Kbyte de tamaño de boque en remoto.
- En local funciona con ext4, con tamaño de bloque de 4 Kbyte.

Los resultados de la prueba de escritura son los siguientes:

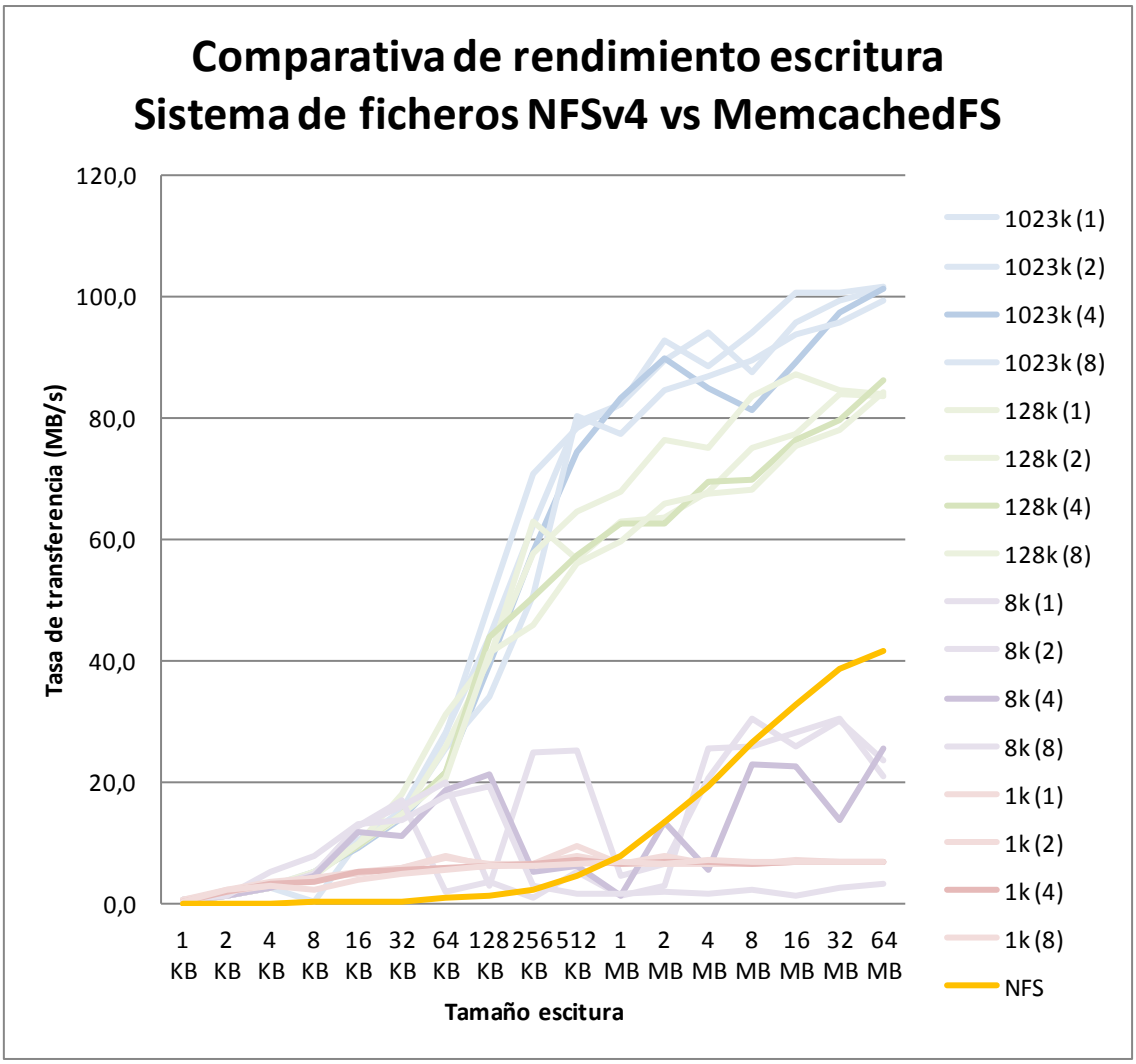

Figura 40: Gráfico de tasas de transferencia en escritura de MemcachedFS vs NFSv4

En este primer caso, referido a las escrituras, se observa 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.

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.

El siguiente caso estudiado es el de las lecturas, en el que se ha utilizado la herramienta "`/proc/sys/vm/drop_caches`" para vaciar las cachés del *kernel* de Linux antes de la lectura y obtener unos resultados fiables. Estos son los resultados:

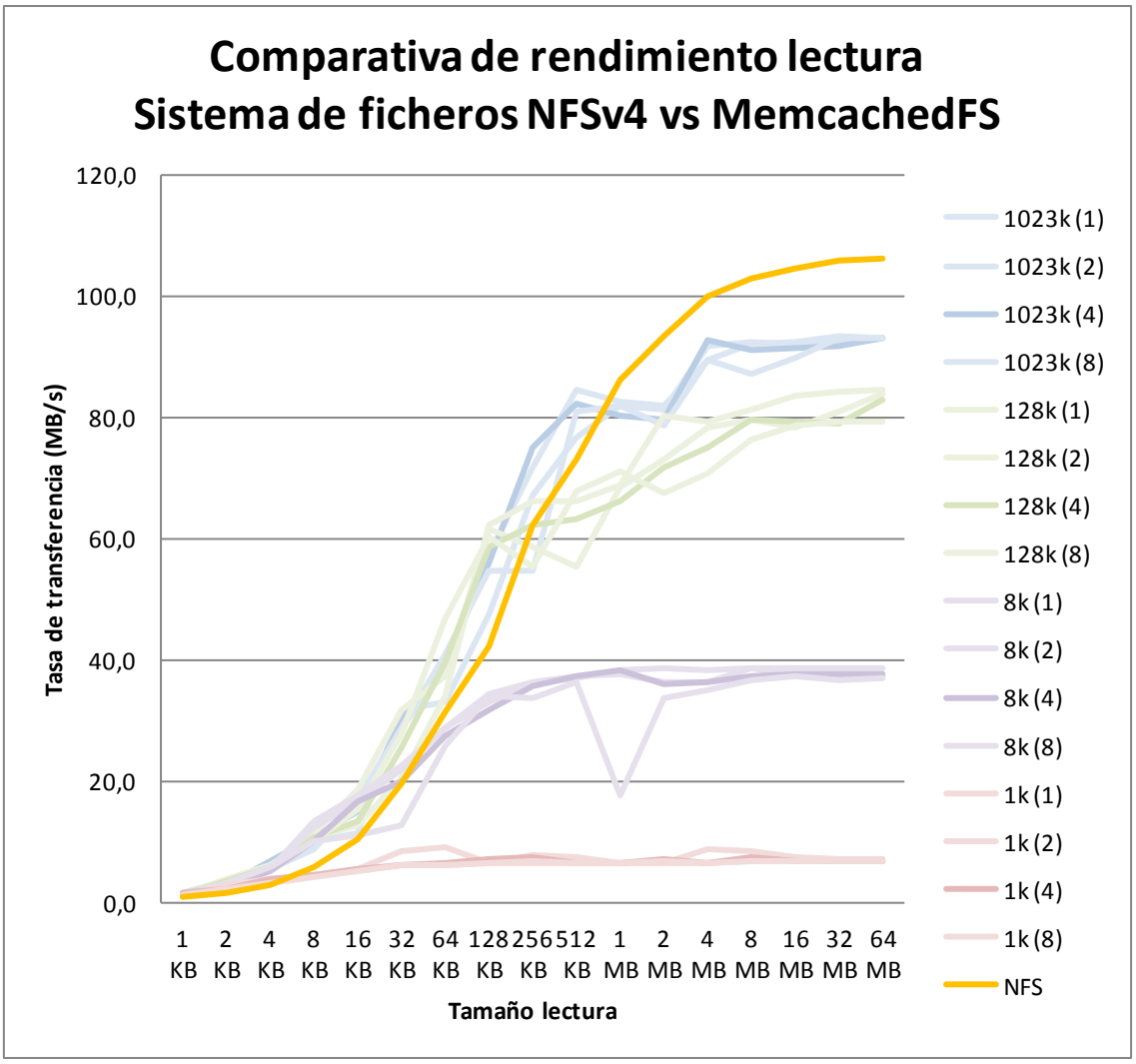

Figura 41: Gráfico de tasas de transferencia en lectura de MemcachedFS vs NFSv4

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 pocos picos.

A continuación el caso estudiado es el de NFSv4 en comparación con MemcachedFS utilizando lecturas múltiples. Se ha utilizado, como en el caso anterior, la funcionalidad que ofrece el *kernel* de Linux para borrar sus cachés y conseguir de este modo los resultados más fiables posibles. Los datos utilizados para las lecturas de NFS son exactamente los mismos que en el caso recién estudiado.

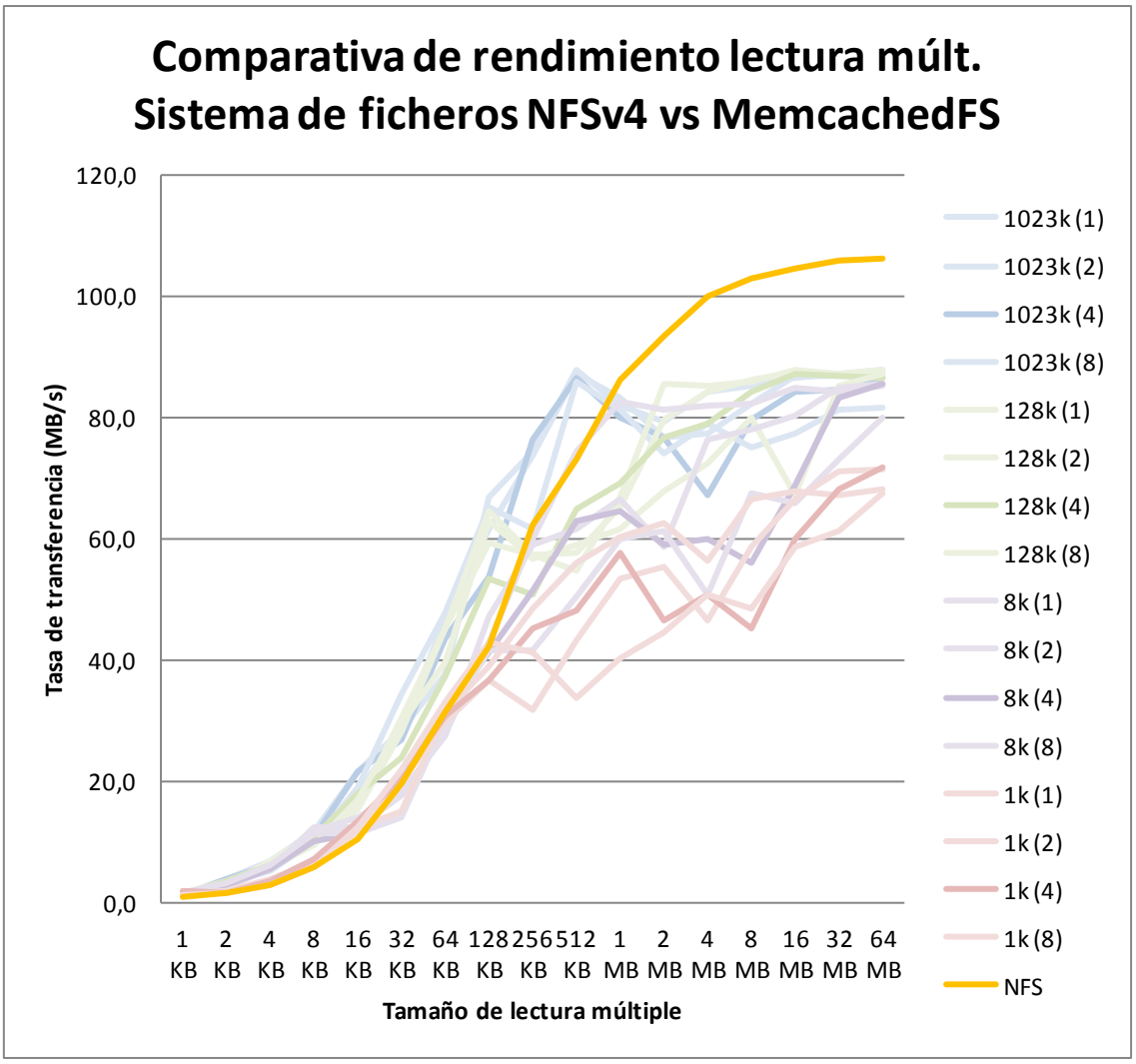

Figura 42: Gráfico de tasas de transferencia en lecturas múltiples de MemcachedFS vs NFSv4

Los resultados son similares a los anteriores. Bajando ligeramente el rendimiento de MemcachedFS con bloques de 1023 Kbyte como ya se había estudiado. La característica más interesante observada es cómo a partir de los tamaños de fichero de 1 Mbyte, MemcachedFS se estanca en su rendimiento mientras que NFS sigue mejorando hasta rozar los límites de la red.

8.5.2 PVFS2

Con respecto a la evaluación de PVFS en comparación con MemcachedFS, se ha utilizado PVFS2 en su versión 2.8.2, con tamaño de bloque de 64 Kbyte y cuatro servidores de E/S. La evaluación de escritura arroja los siguientes resultados:

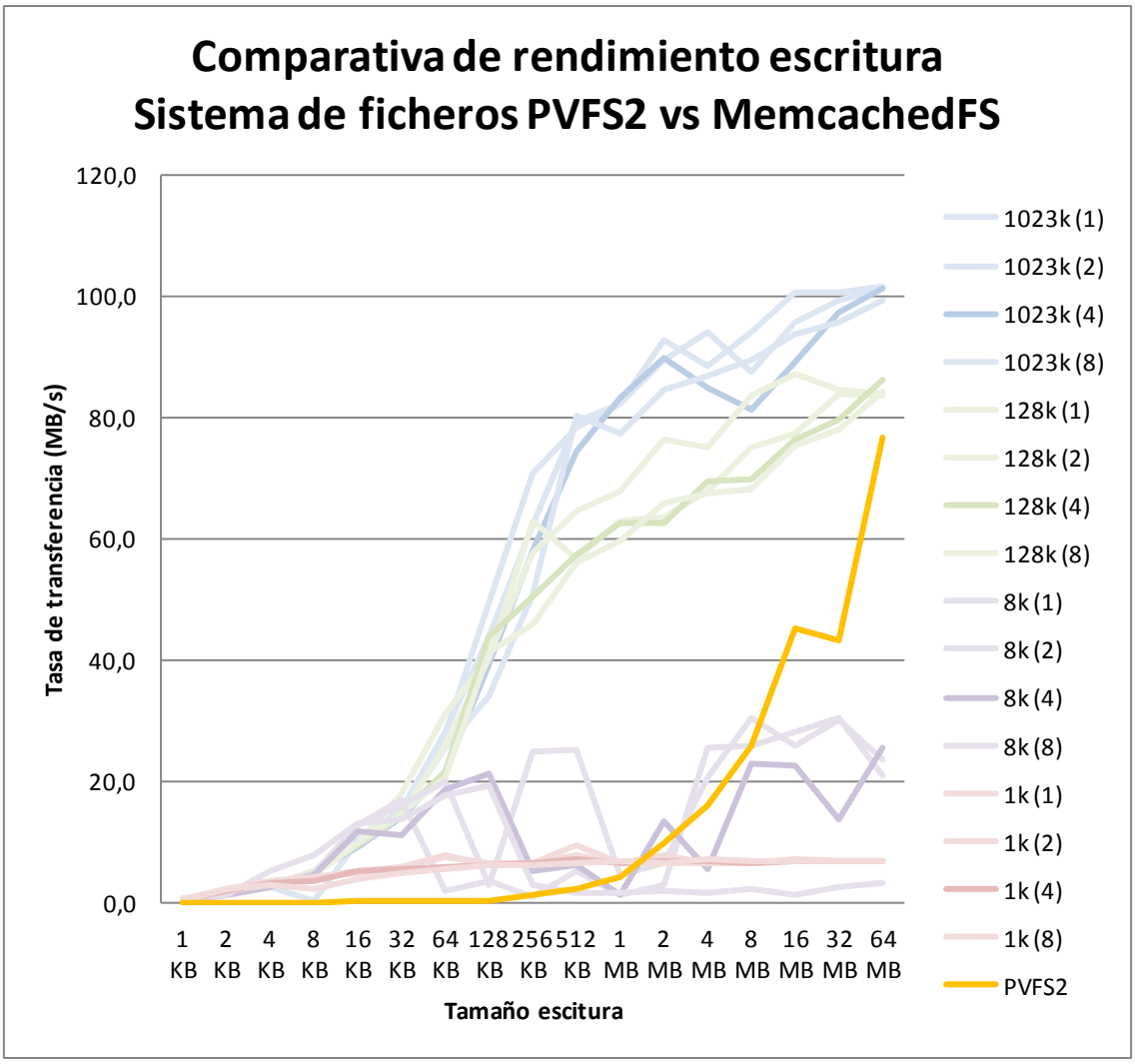

Figura 43: Gráfico de tasas de transferencia en escritura de MemcachedFS vs PVFS2

Lo primero que se observa es que, tanto MemcachedFS como PVFS2 mejoran su rendimiento a medida que aumenta el tamaño de la escritura. 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 64Kbyte (su tamaño de bloque) y el crecimiento de rendimiento es completamente exponencial (al igual que lo es el aumento del tamaño de fichero), MemcachedFS llega al 50% de su rendimiento con tamaños de fichero inferiores al Mbyte y se mantiene siempre por encima de PVFS2 en el resto de casos.

Incluso con un tamaño de bloque de 8 Kbyte, MemcachedFS se mantiene competitivo con respecto a PVFS2, salvo para tamaños de fichero extremadamente grandes, en cuyo caso no tiene sentido el uso de un tamaño de bloque tan pequeño.

A continuación, los resultados de la evaluación de lectura normal, es decir, lecturas de PVFS2 comparadas con lecturas simples de MemcachedFS:

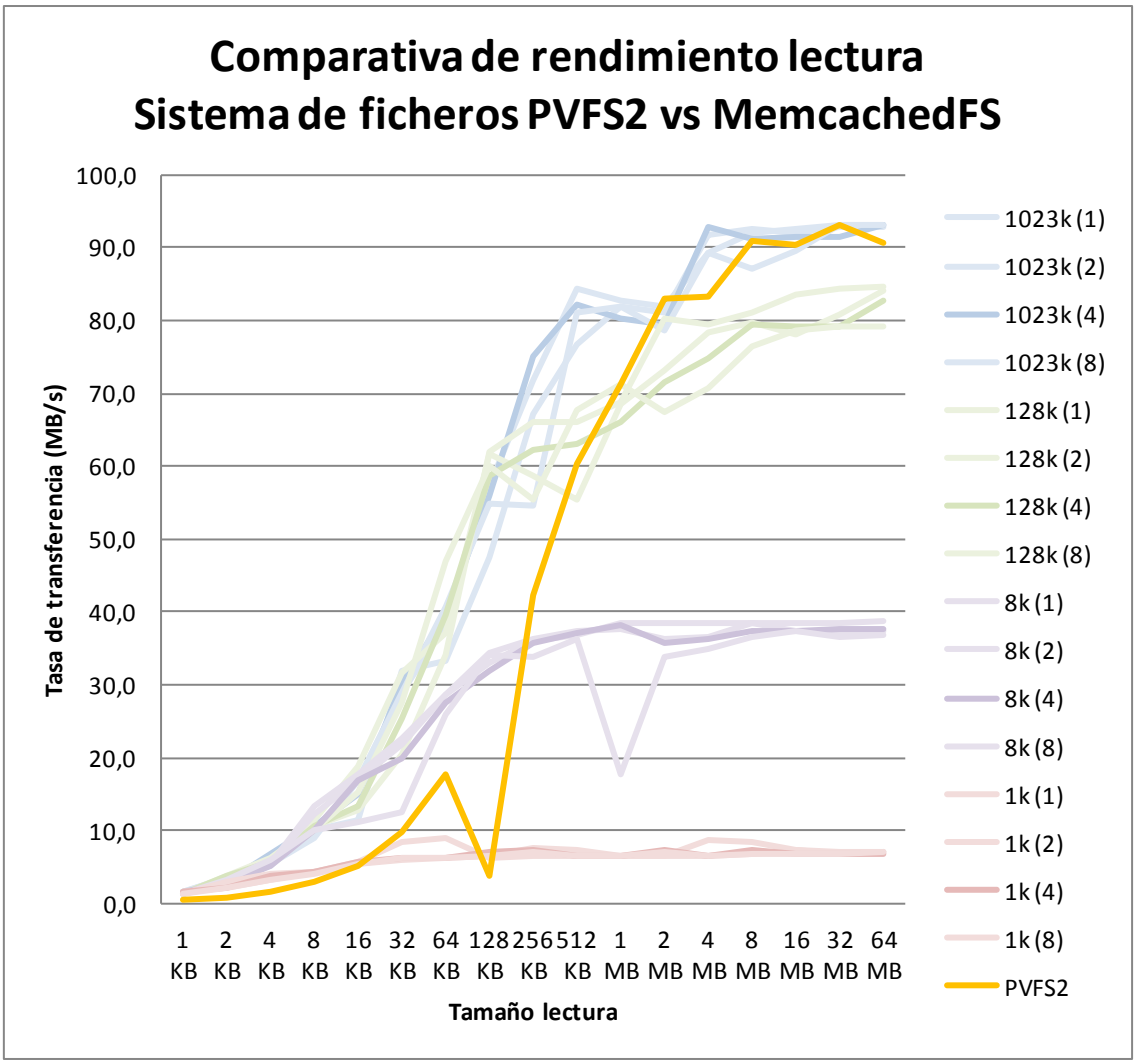

Figura 44: Gráfico de tasas de transferencia en lectura de MemcachedFS vs PVFS2

En este caso, el modelo que sigue el rendimiento de PVFS2 es muy parecido al que sigue MemcachedFS con tamaño de bloque de 1023 Kbyte. 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, MemcachedFS consigue resultados bastante competitivos con tamaños de bloque grandes, mientras que su rendimiento es un poco flojo con tamaños de bloque pequeños, algo lógico debido al aprovechamiento de la red.

Por último, se han evaluado las lecturas múltiples de MemcachedFS contra las lecturas de PVFS2 (los mismos datos que antes para este último) dando los siguientes valores:

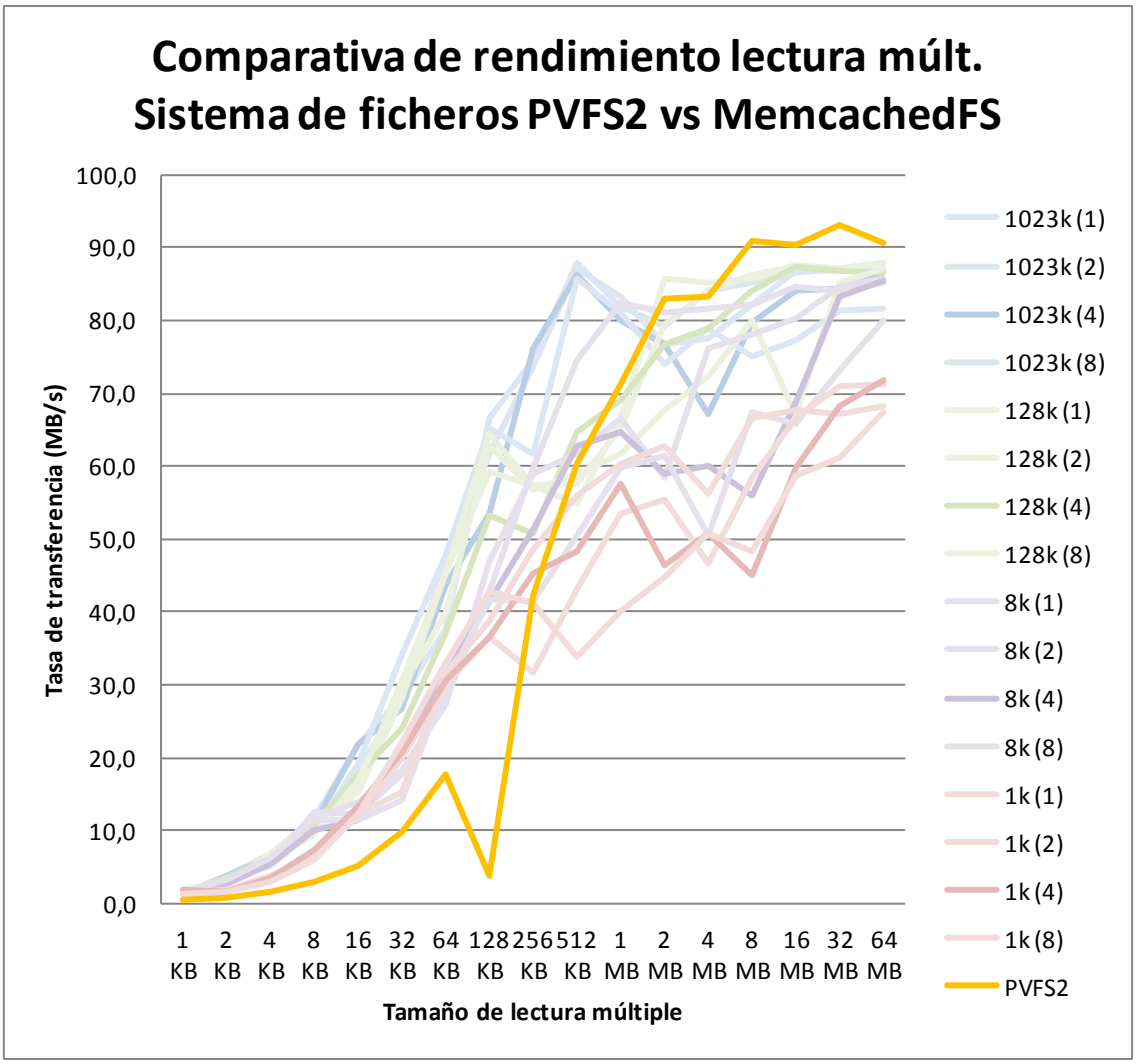

Figura 45: Gráfico de tasas de transferencia en lectura múltiple de MemcachedFS vs PVFS2

Las lecturas múltiples de MemcachedFS favorecen ampliamente los tamaños de bloque pequeños para optimizar el rendimiento de la red. Por tanto, existe una diferencia más reducida entre el mejor caso y el peor. Sin embargo, penalizan ligeramente las lecturas de ficheros de gran tamaño con tamaños de bloque grandes. Por este motivo, con tamaños de fichero medios MemcachedFS se muestra muy fuerte sea cual sea su configuración, pero para tamaños de fichero grandes su rendimiento cae ligeramente por debajo de PVFS2.

En definitiva, como conclusiones generales, se puede afirmar que el rendimiento de MemcachedFS es competitivo con sistemas de ficheros maduros como PVFS2 o NFSv4. Incluso, en algunos casos se muestra superior, haciendo que sea una alternativa viable a estos sistemas.

Además, la principal ventaja que no se observa en estas pruebas es que el rendimiento de MemcachedFS depende exclusivamente de la red (internamente, trabaja con los datos en memoria principal, por lo que nunca será un cuello de botella) mientras

que en algún punto, los sistemas mecánicos que corren tanto sobre PVFS2 como sobre NFS pueden suponer un lastre en su rendimiento.

8.6 Lectura/escritura aleatoria

Todas las evaluaciones realizadas hasta ahora, se basan en lecturas y escrituras secuenciales, hechas sobre ficheros completos. A continuación se mostrarán los resultados de los procedimientos realizados para evaluar el rendimiento de MemcachedFS en lecturas y escrituras aleatorias de tamaños relativamente pequeños. Además, se han comparado los resultados con los obtenidos por un sistema de ficheros local (ext4), y dos distribuidos (NFSv4 y PVFS2).

Las plataformas de pruebas de estos tres sistemas son las indicadas en las evaluaciones anteriores, es decir, el sistema de ficheros local está montado sobre un disco duro Caviar Black de 640 Gbyte, y los sistemas de ficheros distribuidos han sido evaluados en el clúster (en el caso de MemcachedFS con cuatro servidores, el caso medio).

El procedimiento seguido es sencillo. Para poder asemejar los datos con las lecturas y escrituras secuenciales, se ha decidido que el tamaño de las escrituras y lecturas fuese de 64 Mbyte. Sin embargo, en lugar de realizar las operaciones en modo secuencial, debían realizarse en modo aleatorio. Para conseguirlo, se han realizado dos procedimientos distintos. En primer lugar se explica el caso de las escrituras.

Como inicialización, se ha creado un fichero de datos aleatorios de 1 Gbyte de tamaño. A continuación, se realizan 1024 escrituras de 64 Kbyte, cada una en una zona completamente distinta y aleatoria del fichero (se genera un *offset* aleatorio y se coloca el puntero del descriptor de fichero en dicha posición mediante `lseek`). Como segundo procedimiento, se realiza una prueba muy similar, pero en este caso con 64 escrituras de 1 Mbyte, también en zonas completamente aleatorias del fichero de 1 Gbyte creado al inicio.

Hay varias formas de realizar la medición, la que se ha utilizado es la siguiente: se toma el tiempo de inicio justo antes de comenzar la escritura, se escribe, se cierra el fichero y se toma el tiempo de fin. Esto para cada una de las escrituras (1024 y 64) y se suman sus resultados. Para el caso del sistema de ficheros local, se ha esquivado la caché liberándola mediante la herramienta "`/proc/sys/vm/drop_caches`" antes de comenzar cada uno de los dos bloques de escrituras, y utilizando la función `sync()` tras cerrar el fichero, dentro del tiempo de cronometrado.

Los resultados obtenidos para las escrituras de 64 Kbyte son los siguientes:

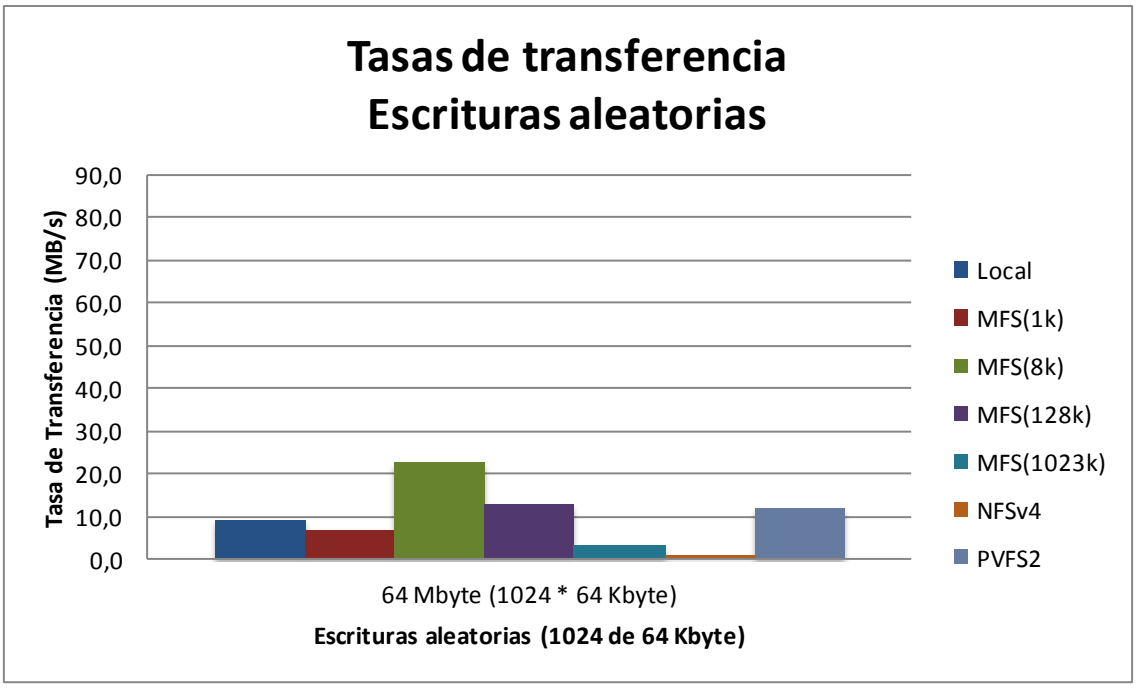

Figura 46: Comparativa de tasas de transferencia de escrituras aleatorias (1024 de 64 Kbyte)

NOTA: en las leyendas MFS se refiere a MemcachedFS, estando entre paréntesis el tamaño de bloque con el que estaba configurado.

En este primer caso se observa claramente cómo el tamaño de bloque influye de forma muy importante sobre los resultados. MemcachedFS es capaz de superar al resto de competidores con los tamaños de bloque que más se ajustan al tamaño de las escrituras (8 Kbyte y 128 Kbyte para escrituras de 64 Kbyte). NFSv4 sufre mucho debido a su tamaño de bloque de 512 Kbyte y, probablemente, también debido su gestión de persistencia, es decir, aparecen los inconvenientes de los discos mecánicos y los inconvenientes de los sistemas distribuidos con un tamaño de bloque grande para escrituras pequeñas.

Con respecto a PVFS2, a pesar de tener un tamaño de bloque de 64 Kbyte, exactamente igual que el tamaño de las escrituras, no es capaz de superar a MemcachedFS, pero se queda muy cercano. Probablemente sea debido, de nuevo, a los problemas de los discos duros para este tipo de escrituras.

Por último, se observa que el disco duro local obtiene unos resultados muy decentes para ser uno de los peores casos a los que se le podía exponer, situándose en la media de rendimiento de todos los casos estudiados.

A continuación, se muestra la gráfica que contiene los resultados obtenidos de cada sistema de ficheros evaluado con escrituras de 1 Mbyte:

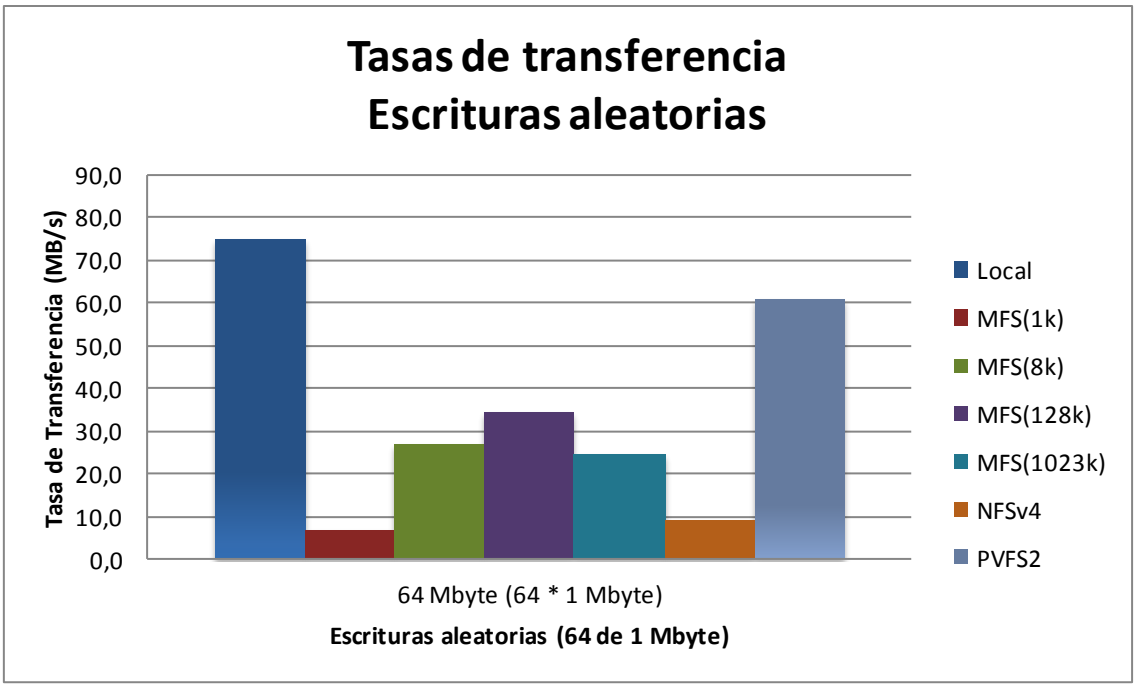

Figura 47: Comparativa de tasas de transferencia de escrituras aleatorias (64 de 1 Mbyte)

Cuando crece el tamaño de bloque, mejora enormemente el rendimiento de las escrituras de los discos duros tradicionales. Este caso, unido a la mejor eficiencia de la red en transferencias grandes, hacen que MemcachedFS quede por debajo de rendimiento en todos los casos (salvo NFSv4).

Cabe destacar un dato interesante, a pesar de aumentar el tamaño de las escrituras, el caso de bloques de 128 Kbyte sigue siendo superior al de 1023 Kbyte. Esto se debe a que la escritura es justo de 1024 Kbyte, por tanto, MemcachedFS con 1023 Kbyte de tamaño de bloque, tiene que realizar dos escrituras. Si la escritura fuese menor a 1023 Kbyte, el rendimiento prácticamente se duplicaría, quedando muy cercano al obtenido por PVFS y ext4 en local.

Como conclusión, se puede sacar que MemcachedFS con un tamaño de bloque 128 Kbyte es capaz de comportarse bien en todos los ámbitos probados. Escrituras secuenciales de cualquier tamaño, así como escrituras aleatorias. Además, en caso de que el sistema tenga unas necesidades muy específicas, MemcachedFS puede adaptar su tamaño de bloque optimizando con ello el rendimiento.

A continuación se muestran los resultados de la prueba de lecturas simples. En este caso, solo han sido evitadas las cachés del modo local, y solo antes de comenzar el bloque completo de lecturas, es decir, no se borraba la caché después de cada lectura, ya que, las lecturas aleatorias ya son bastante complejas de por sí, sería una penalización injusta. Además, se han cronometrado los dos bloques completos de lecturas (generación de *offset* aleatorio, mover el puntero y leer), no cada lectura por separado y sumadas al final.

El resto del procedimiento es similar, sobre un fichero de 1 Gbyte de datos aleatorios, primero se realizan 1024 lecturas de 64 Kbyte en posiciones aleatorias del fichero y, después, 64 lecturas de 1 Mbyte, también en posiciones aleatorias del fichero.

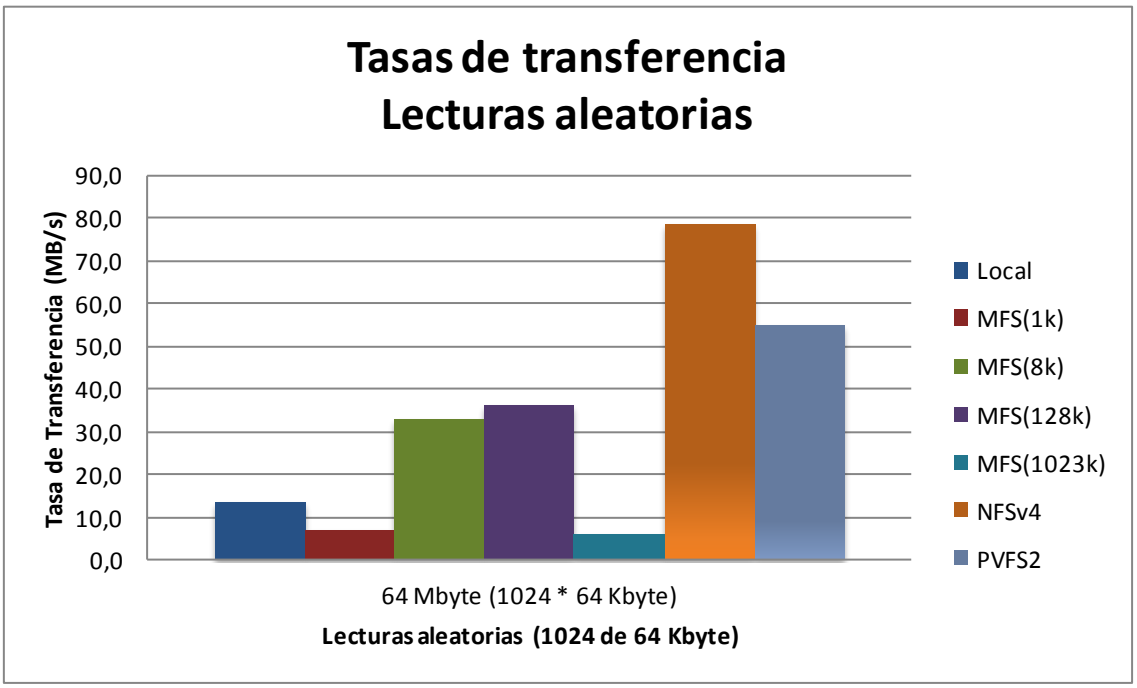

Figura 48: Comparativa de tasas de transferencia de lecturas aleatorias (1024 de 64 Kbyte)

Los mejores resultados de MemcachedFS vuelven a ser arrojados por el tamaño de bloque de 128 Kbyte pero, en este caso, aún quedan lejos de los mejores resultados de NFSv4 y PVFS2, que obtienen muy buen rendimiento, NFSv4 incluso cercano a los límites de la red, lo que sugiere que la caché puede haber influido (se recupera el valor casi de inmediato y se envía por la red a la máxima velocidad posible o se recuperan algunos valores de los servidores remotos y otros se consiguen de la caché local, consiguiendo que, en media, los resultados mejoren).

Además, es curioso cómo NFSv4 es el que mejores resultados consigue, teniendo un tamaño de bloque de 512 Kbyte. Si tuviese que acceder de forma remota a todas las lecturas realizadas, tendría que transferir por la red 512 Mbyte, algo imposible en los tiempos que se han medido, dando indicios de que existe algún tipo de caché local interfiriendo en los resultados.

También influye en el rendimiento de MemcachedFS que se modifican los metadatos del fichero en cada lectura, teniendo que realizar dos operaciones adicionales por cada lectura (en el caso de los metadatos, leer el bloque y escribirlo). Es posible que los otros sistemas de ficheros distribuidos sólo actualicen los metadatos al cerrar el fichero.

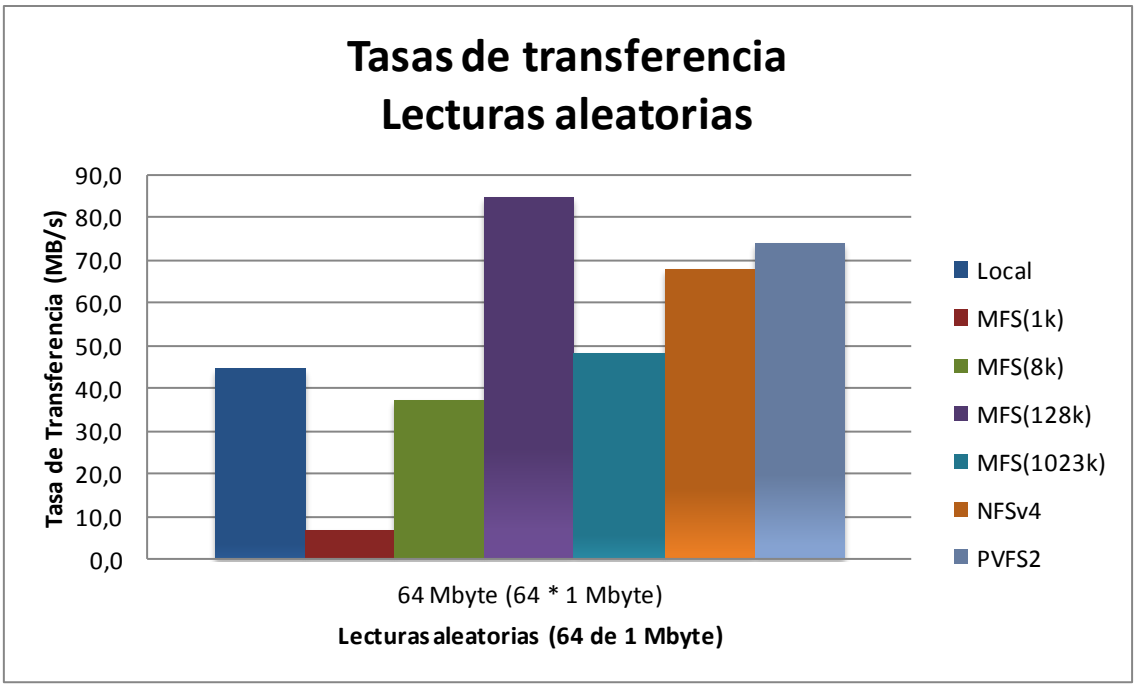

Figura 49: Comparativa de tasas de transferencia de lecturas aleatorias (64 de 1 Mbyte)

Con un tamaño de bloque adecuado y un tamaño de lectura suficiente para aprovechar al máximo el rendimiento de la red, MemcachedFS toma la delantera en el caso de 128 Kbyte (y lo haría también el caso de 1023 Kbyte de bloque si se leyesen menos de 1024 Kbyte, como se ha explicado en el caso de las escrituras).

También mejora el rendimiento de PVFS2, al que le afecta más el rendimiento de la red que a NFSv4. De hecho, es curioso cómo mejora el resultado, cuando las lecturas anteriores se adaptaban a la perfección a su tamaño de bloque, sin duda, debido a un mejor aprovechamiento de la red.

En este caso, es más difícil que NFSv4 salga beneficiado por las cachés, ya que, no se suelen cachear tamaños tan grandes (1 Mbyte completo), porque se saturaría la caché enseguida.

El caso local obtiene los resultados esperados. Mejora el rendimiento con respecto al caso exterior al aumentarse el tamaño de la lectura (más datos secuenciales, mejor rendimiento). Aunque es superado o igualado por MemcachedFS en la mayoría de los casos (el modo de 1 Kbyte es realmente ineficiente por el alto número de conexiones necesarias).

En definitiva, de nuevo el tamaño de bloque en comparación con el tamaño de la lectura es un punto fundamental, principalmente para MemcachedFS cuyos tiempos de procesamiento en el servidor son bajísimos, y su rendimiento depende enormemente del aprovechamiento de la red.

A continuación se comparan estos mismos datos de ext4, NFSv4 y PVFS2 con la misma prueba realizada esta vez con lecturas múltiples (`read_mc_mult`) en MemcachedFS.

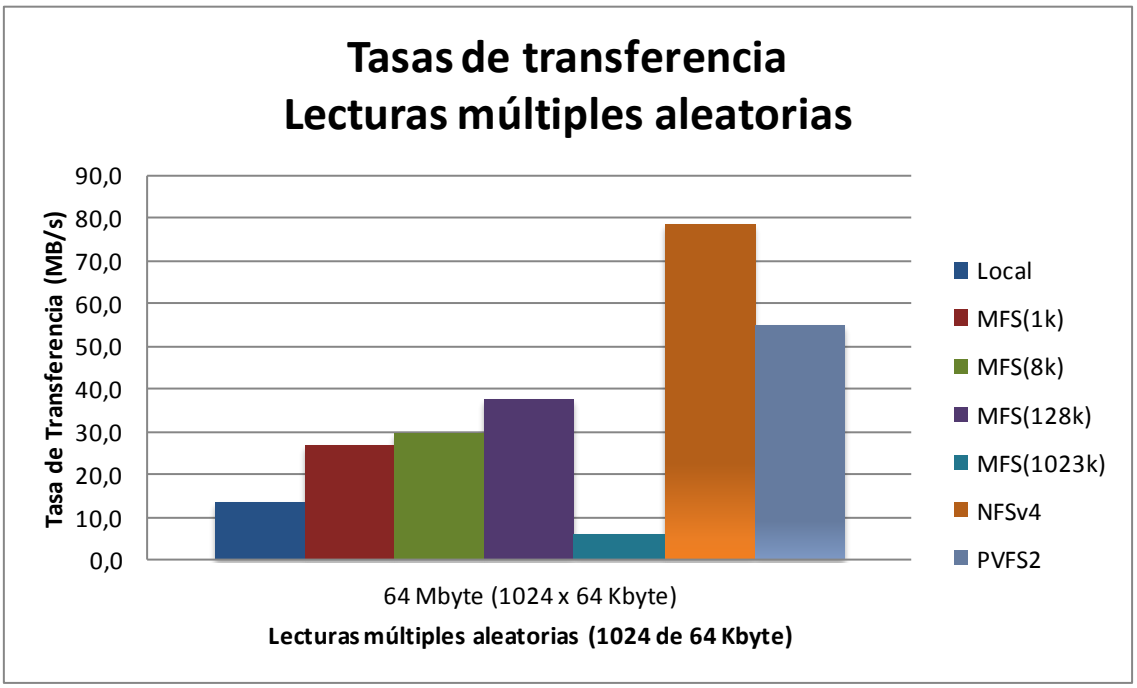

Figura 50: Comparativa de tasas de transferencia de lecturas múltiples aleatorias (1024 de 64 Kbyte)

Los resultados no cambian en exceso. El aprovechamiento de la red de MemcachedFS es exactamente el mismo para la mayoría de los casos. 128 Kbyte y 8 Kbyte se adaptan bastante bien al tamaño de las lecturas, 1023 Kbyte se queda completamente fuera de rango (tiene que transferir casi 1 Gbyte de datos para realizar las 1024 lecturas) y es el caso de 1 Kbyte el que realmente mejora con las lecturas múltiples. Al reducirse el número de conexiones necesarias, se optimiza el rendimiento de la red, y se consigue un resultado comparable al resto de casos y competitivo, al menos, con el modo local.

El resto de consideraciones son exactamente iguales al caso de lecturas simples, ya que, los resultados no han cambiado en exceso.

A continuación, se muestran los resultados obtenidos con 64 lecturas aleatorias de 1 Mbyte de tamaño:

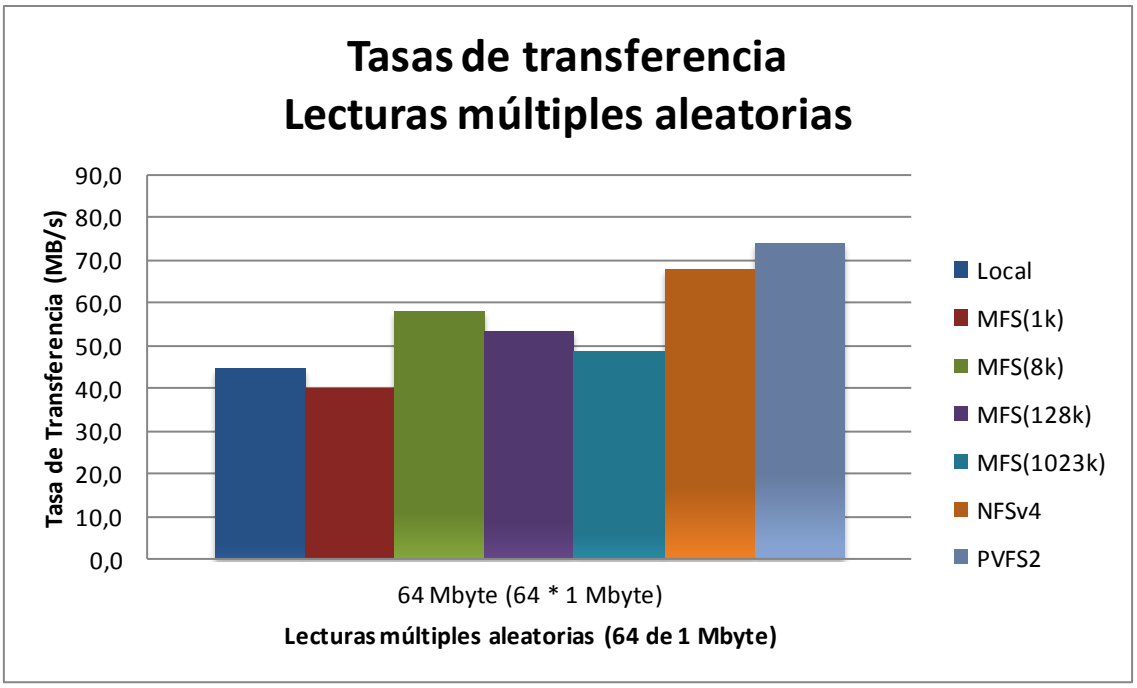

Figura 51: Comparativa de tasas de transferencia de lecturas múltiples aleatorias (64 de 1 Mbyte)

En este caso, el cambio es más pronunciado que en el anterior. Gracias a las lecturas múltiples, todos los resultados de MemcachedFS se homogenizan, no destacando ninguno por encima del resto (el caso de 1023 Kbyte se mantiene exactamente igual, ya que, el funcionamiento de las lecturas múltiples solo cambia cuando se leen dos o más bloques completos).

Gracias a las lecturas múltiples, se garantiza un resultado bastante homogéneo sea cual sea el tamaño de bloque en comparación con el tamaño de lectura. Es decir, si los tamaños de lectura van a ser muy heterogéneos, no es necesario preocuparse de configurar el sistema con un tamaño de bloque muy concreto, ya que, mediante las lecturas múltiples, se consigue un rendimiento similar en todos los casos (salvo cuando el tamaño de bloque supera con creces el tamaño de lectura, ya que, se deben transferir por la red muchos más datos de los necesarios, aprovechando peor la red).

Además, todos los casos son competitivos, situándose ligeramente por encima de las lecturas locales y ligeramente por debajo de los sistemas de ficheros distribuidos más maduros que, sin embargo, no dependen sólo de la velocidad de la red como MemcachedFS.

Con respecto a los modos de caché, viendo que en la mayoría de los casos, principalmente lecturas, superan al rendimiento de un disco duro local, pueden suponer una mejora interesante para procesos que requieran muchas operaciones de lectura/escritura en una sola sesión, es decir, las suficientes para que se compensen las latencias de apertura y cerrado con la mejoría de rendimiento en el resto de operaciones.

8.7 Evaluación del sistema bajo condiciones de carga (stress-test)

Para evaluar el sistema bajo condiciones de mucha carga de trabajo, se ha decidido realizar lecturas y escrituras simultáneas de dieciséis clientes de MemcachedFS. Para ello, los dieciséis clientes son lanzados de manera consecutiva en *background*, de modo que todos queden trabajando de manera simultánea.

Para asegurar esta condición de peticiones simultáneas a los servidores, se realizan en cada uno de los clientes, en este primer caso de escritura, diez escrituras de un fichero de 100 Mbyte de datos aleatorios, a continuación una escritura cronometrada de 100 Mbyte (open + write + close) y, por último, otras diez escrituras también de 100 Mbyte. De este modo, se puede tener la certeza de que en el momento de cronometrar la escritura de cada cliente, el resto de clientes aún estarán trabajando, bien sea en sus 20 escrituras de inicio o en las de fin y, por tanto, estresando el sistema.

La evaluación ha sido realizada con 1, 2, 4 y 8 servidores, para comprobar si los efectos negativos de una posible sobrecarga de trabajo pueden ser mitigados mediante el balanceo de dicha carga.

En primer lugar, se ha realizado la evaluación con tamaños de bloque de 1023 Kbytes, que fueron el mejor de los estudiados en las lecturas y escrituras de ficheros completos sin carga. El procedimiento ha arrojado los siguientes resultados:

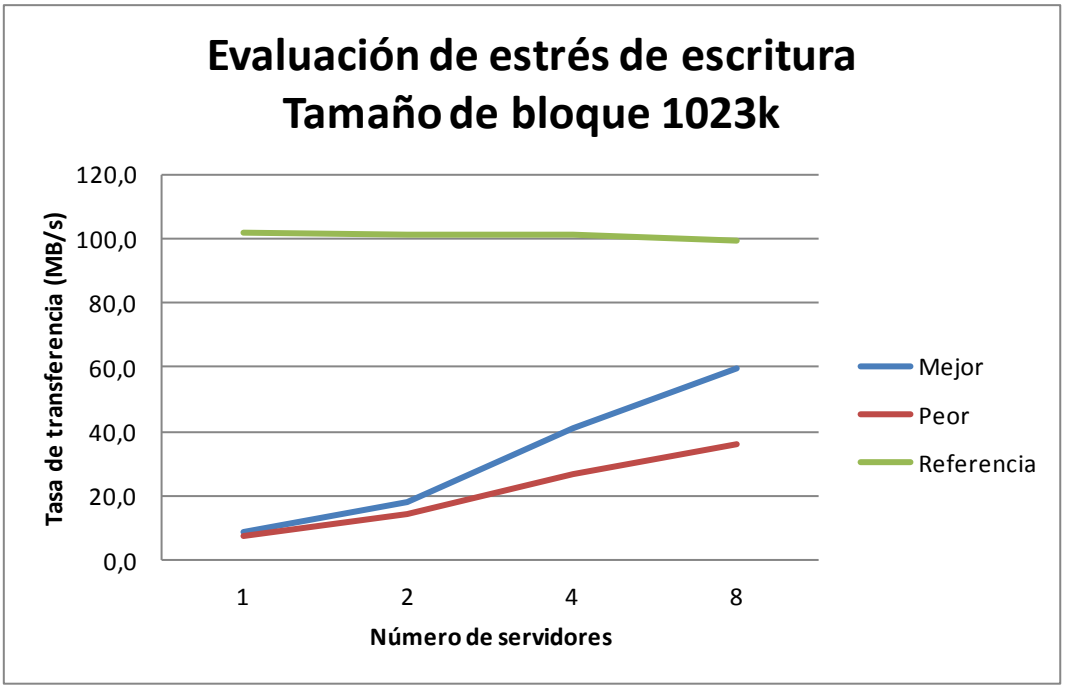

Figura 52: Gráfico de evaluación de estrés de escritura de MemcachedFS (1023k)

El resultado obtenido es contundente, el sistema pierde rendimiento de forma muy notable en el momento en que se empieza a someter a cargas de trabajo importantes, sobre todo si se compara con la medición de referencia en la que se escribía un fichero de 64 Mbyte sin ningún tipo de carga en el sistema.

Es interesante observar cómo el rendimiento bajo carga aumenta a medida que lo hace el número de servidores utilizados para distribuir los datos de MemcachedFS. De esta observación se puede extraer la siguiente conclusión:

La escalabilidad mejora el rendimiento. A pesar de haberse observado ligeras pérdidas de rendimiento en los casos estudiados sin carga (en este ejemplo se ve cómo el gráfico de referencia cae ligeramente a medida que aumenta el número de servidores) ésta pérdida no era significativa en ningún caso. Sin embargo, a medida que aumenta la carga en el sistema, la mejora de rendimiento que aporta la escalabilidad es muy sustancial, haciéndola especialmente recomendable para sistemas con alta carga de trabajo, es decir, es preferible tener varios servidores utilizando poca RAM para *Memcached* que un solo sistema que dedique toda su memoria exclusivamente a esta tarea.

Con respecto a las causas de la mejora, surgen dos teorías en torno a la saturación de la red. Por un lado, puede entenderse que la carga de la red es exactamente la misma en todos los casos estudiados (se transmite la misma cantidad de datos en todos los procedimientos de evaluación) y, por tanto, la mejora del rendimiento es cuestión exclusiva de la saturación de los servidores. Sin embargo, al aumentar el número de servidores, se aumenta también el número de tarjetas de red y de hilos de la red, pudiendo conllevar menos colisiones y, con ello, aumento del rendimiento de la propia red.

El siguiente caso estudiado es exactamente el mismo, es decir, escrituras simultáneas de diez clientes de MemcachedFS pero, en este caso, con un tamaño de bloque de 128 Kbyte.

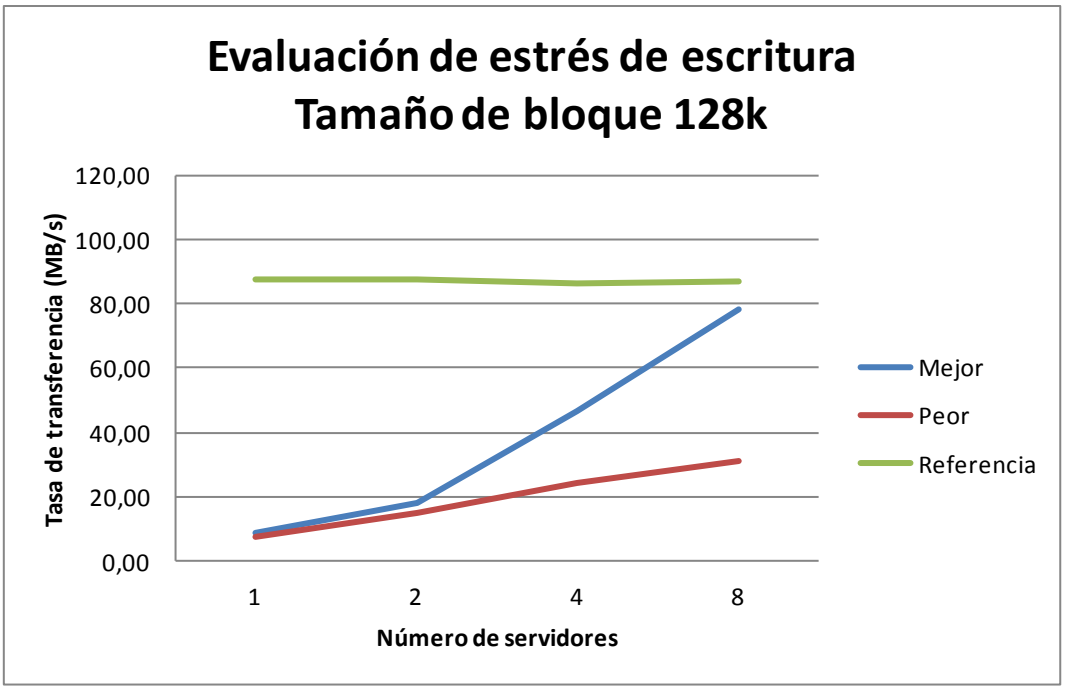

Figura 53: Gráfico de evaluación de estrés de escritura de MemcachedFS (128k)

Los resultados obtenidos son similares a los anteriores pero aún más acusados. Por un lado, a partir de tamaños de bloque de 128 Kbyte, se nota una reducción del rendimiento del sistema, que en este caso hace que las tasas de transferencia no lleguen a 85 MB/s.

Por otro lado, se observa cómo en el peor caso el rendimiento es similar al caso anterior, sin embargo, la mejora experimentada por el sistema a medida que se mejora su escalabilidad es aún más pronunciada que con tamaños de bloque de 1023 Kbyte, acercándose mucho al mejor rendimiento obtenido en los casos sin carga.

La siguiente evaluación realizada es equivalente a la anterior pero sustituyendo las escrituras por lecturas simples. Es decir, se lanzan cuatro clientes en cuatro nodos del clúster (hasta un total de dieciséis), escriben un fichero aleatorio de 100 Mbyte en MemcachedFS configurado con tamaño de bloque de 1023 Kbyte y, acto seguido, realizan diez lecturas de estrés, una lectura cronometrada y, para finalizar, otras diez lecturas de estrés. Los resultados obtenidos son los siguientes:

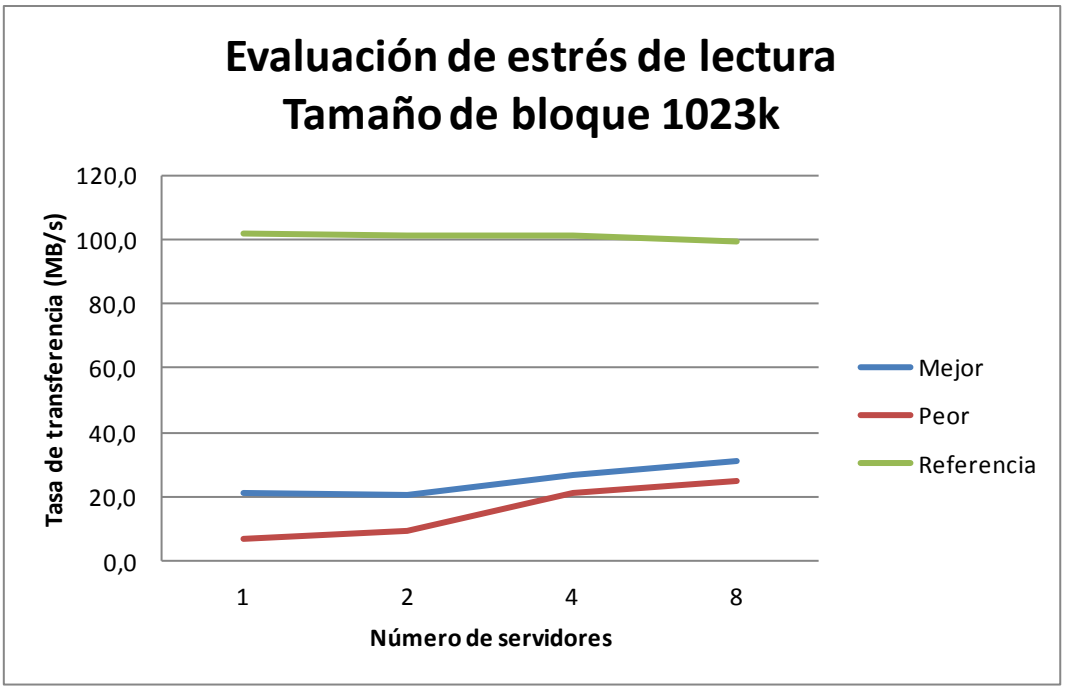

Figura 54: Gráfico de evaluación de estrés de lectura de MemcachedFS (1023k)

De nuevo se utiliza como referencia el caso de lecturas sin carga de un fichero de 64 Mbyte. Como primera observación, se nota claramente la pérdida de rendimiento causada por la carga del sistema. Las tasas de transferencia se reducen mucho y, a pesar de que mejoran a medida que se aumenta el número de servidores de *Memcached* utilizados, el rendimiento en ningún momento se acerca a los valores que el sistema consigue sin carga ni el crecimiento del rendimiento es proporcional al aumento de servidores.

Si se reduce el tamaño de bloque de MemcachedFS a 128 Kbyte, los valores medidos son los siguientes:

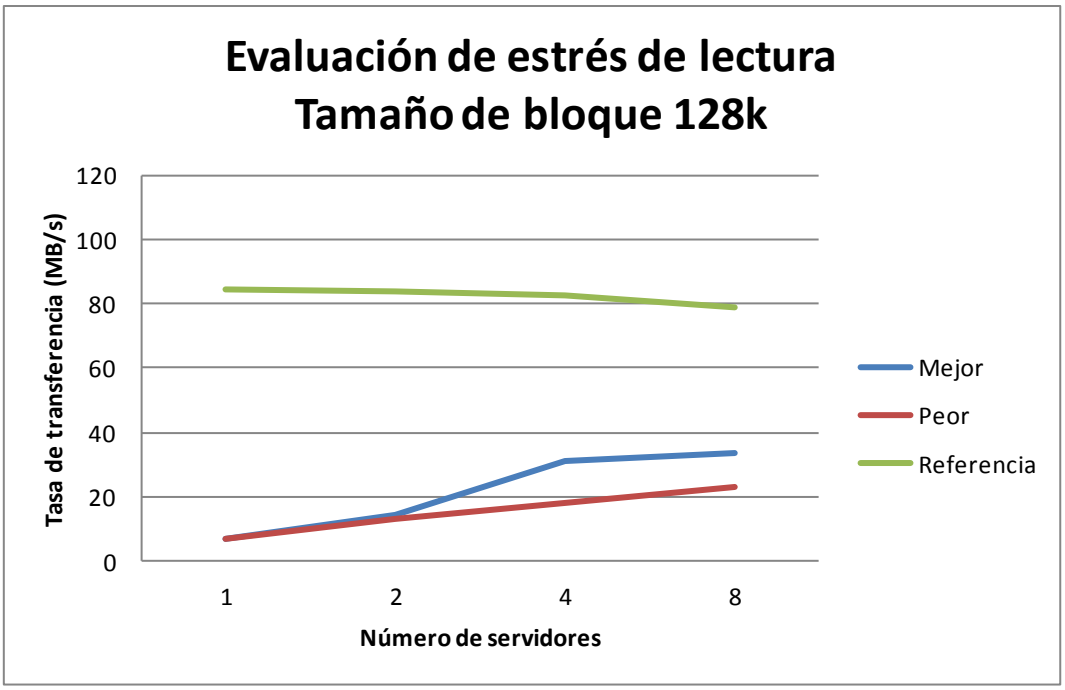

Figura 55: Gráfico de evaluación de estrés de lectura de MemcachedFS (128k)

Los resultados de la evaluación son muy similares a los anteriores. En este caso podrían considerarse algo mejores porque la lectura sin carga es menos eficiente, pero lo cierto es que en el momento en que aparece un nivel de carga en el sistema, las tasas de transferencia son muy similares, así como la evolución del rendimiento conforme aumenta el escalado.

La conclusión es similar al caso de las escrituras, sin embargo, en las lecturas la pérdida de rendimiento es notable en todos los casos, quedando muy lejos del rendimiento óptimo del sistema y con una mejora menos notable del rendimiento mediante escalabilidad.

También se ha realizado la misma prueba, utilizando la función de lecturas múltiples (`read_mc_mult`) en lugar de las lecturas simples (`read_mc`). Las condiciones del procedimiento son exactamente las mismas que en los casos anteriores, midiéndose estos resultados:

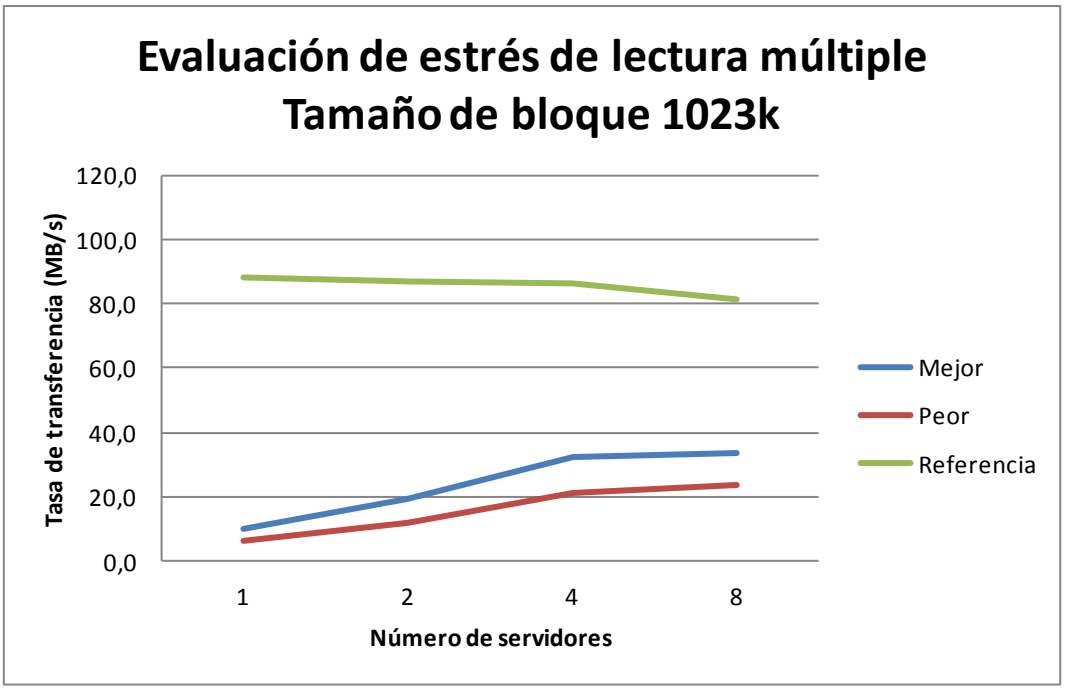

Figura 56: Gráfico de evaluación de estrés de lectura múltiple de MemcachedFS (1023k)

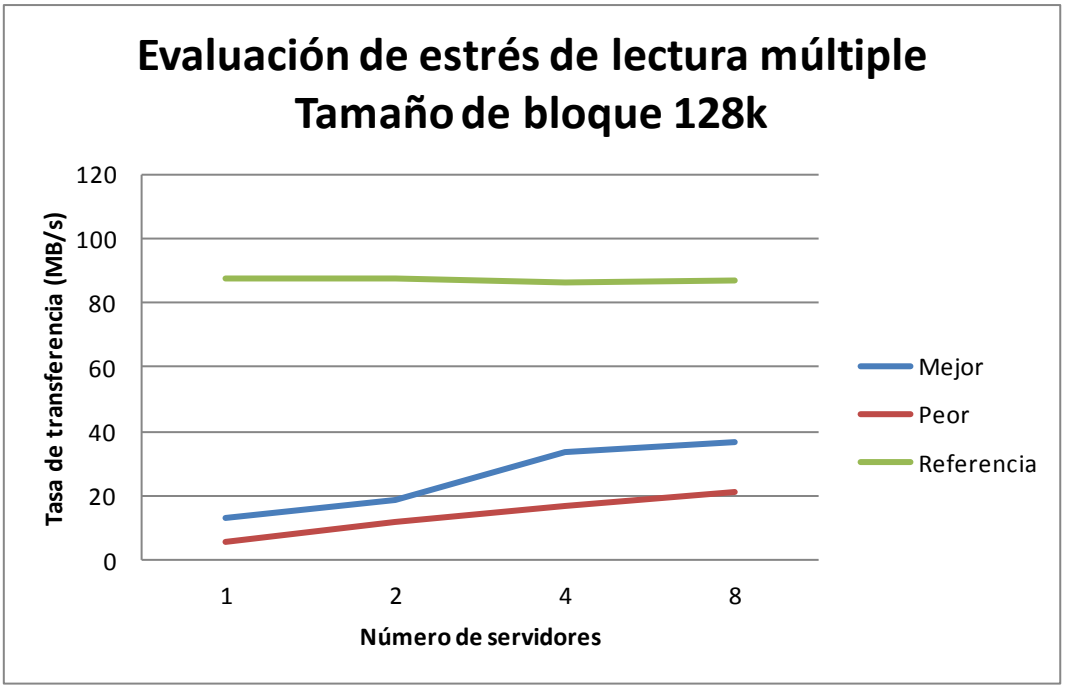

Figura 57: Gráfico de evaluación de estrés de lectura múltiple de MemcachedFS (128k)

Los resultados obtenidos son muy similares a los anteriores. Gran pérdida de rendimiento y mejora con la escalabilidad. Sin embargo, se deben desatacar dos puntos importantes.

En primer lugar, *Memcached* está muy orientado a las lecturas, por lo que no parece que tenga mucho sentido que el rendimiento decaiga más en las lecturas que en las escrituras. Además, no solo es así, sino que la situación no mejora al utilizar lecturas múltiples (supuestamente, más optimizadas). Mediante el uso de lecturas múltiples las conexiones con *Memcached* se reducen, por tanto, la carga del servidor debería también reducirse. Al no mejorarse los resultados, se puede deducir que el punto flaco del sistema vuelve a ser la red, que en el momento en que se satura, hace caer el rendimiento de las tasas de transferencia. Es un problema habitual y conocido la pérdida de rendimiento de las redes Ethernet cuando intentan trabajar sobre una misma red varios clientes de forma simultánea, debido al sistema de gestión de las colisiones.

Como segundo punto interesante, se observa que la mejora de rendimiento se estanca a partir de cuatro servidores, mejorando de forma muy escasa tanto en el peor caso como en el mejor, a pesar de doblarse el número de servidores.

Cabe destacar, que de todas las pruebas realizadas, en ningún caso se ha producido un error de lectura o escritura. Tan solo se han producido un par de errores de conexión (en la inicialización de *Memcached*) en la prueba de lecturas múltiples con tamaño de bloque de 1023 Kbyte y un solo servidor de *Memcached*. Este error puede ser debido a que, con la saturación de la red se haya perdido algún paquete. Se reintentó la conexión y fue satisfactoria en ambos casos (aunque los resultados obtenidos por ambos clientes fueron invalidados, ya que, habían realizado su prueba cronometrada después del resto, probablemente sin carga, obteniendo tasas de transferencia muy altas). Se puede concluir, por tanto, que el sistema es estable y fiable en casos de alta carga de trabajo, a pesar de su pérdida de rendimiento.

9 CONCLUSIONES Y PRESUPUESTO

9.1 Conclusiones

En este apartado se resume el trabajo global llevado a cabo durante el desarrollo de este Proyecto Fin de Carrera, valorando los objetivos que se han conseguido, tanto en comparación con los propuestos al inicio como los adicionales a éstos.

Se puede concluir sin ningún tipo de duda, que se ha conseguido el objetivo principal del Proyecto Fin de Carrera. Se ha desarrollado una librería para utilizar un sistema de ficheros distribuido basado en una caché distribuida, en este caso *Memcached*, dando como resultado el sistema de ficheros distribuido llamado MemcachedFS.

El primer objetivo perseguido se trataba de diseñar un sistema de ficheros distribuido utilizando *Memcached* como base, y aprovechando al máximo sus características para optimizar su rendimiento. Se ha conseguido minimizar al máximo el número de accesos necesarios para realizar cada operación de lectura/escritura mediante la eliminación de los punteros a bloques de los metadatos, también gracias a esta decisión se ha conseguido un tamaño máximo de fichero suficiente para cualquier uso actual, se han utilizado características avanzadas de *Memcached* como las lecturas múltiples adaptadas a la lectura de bloques consecutivos de ficheros, etc.

El segundo objetivo era la implementación de la librería, que debía ser lo más portable posible y de flexible configuración. Con respecto a la portabilidad, el funcionamiento de la librería es extraordinariamente parecido al de la librería estándar de entrada y salida de POSIX. Su sintaxis es idéntica en todas las funciones: `open`, `read`, `write`, `close`, `lseek`, `unlink` y `rename`. Y su funcionamiento también lo es, hasta el punto de que la verificación de funcionalidades ha sido realizada comparando los resultados de las llamadas a la librería estándar con los ofrecidos por la librería desarrollada.

En relación a la flexibilidad de configuración, la principal aportación conseguida de cara al usuario es la posibilidad de modificar el tamaño de bloque de MemcachedFS y la utilización de otras constantes en los ficheros de cabecera para poder adaptar fácilmente el funcionamiento al sistema sobre el que se utilice. Además, de cara a los usuarios más avanzados que quieran modificar el código de la librería, se ha buscado en todo momento facilitar su trabajo. Como característica principal, la librería está estructurada en dos capas, la primera capa es la que ofrece al usuario el API de MemcachedFS, mientras que la segunda capa es la encargada de realizar todas las llamadas a la librería libMemcached y otras funciones auxiliares. De este modo, se facilitan los posibles cambios tanto de la librería utilizada para comunicarse con

Memcached, como el cambio del propio sistema de caché distribuida *Memcached* por otro de similares características, tan solo modificando la capa interna, sin modificar en absoluto el núcleo de MemcachedFS.

También se han tenido en cuenta posibles ampliaciones de la funcionalidad, dejando en las estructuras de datos espacios, por ejemplo, para la administración avanzada de permisos de usuarios. Como característica añadida, todo el código está fuertemente documentado en inglés de cara a facilitar en la medida de lo posible estos cambios.

Aprovechando la funcionalidad desarrollada como sistema de ficheros distribuido y, teniendo presente que *Memcached* está pensado como sistema de caché distribuida, se han desarrollado dos modos de funcionamiento adicionales: el modo pre-caché que abre un fichero del disco local en que se ejecute el cliente, lo copia a la caché distribuida para trabajar con él, y lo devuelve actualizado al disco local cuando se cierra, y el modo caché, que copia a la caché distribuida los bloques de datos de un fichero local según van siendo necesitados, y devuelve a disco local todos los que han sido modificados al cerrar el fichero. Estos modos de funcionamiento, solo difieren del funcionamiento de la librería de E/S estándar en el momento de la apertura, el resto de operaciones se realizan de forma totalmente transparente al usuario. Se pueden utilizar estos modos tanto para utilizar MemcachedFS como caché de un sistema de ficheros local, como para utilizar ficheros locales en MemcachedFS facilitando las operaciones al usuario y optimizando el rendimiento.

Por último, se ha evaluado el rendimiento de MemcachedFS, y ha sido comparado con el de un sistema de ficheros local, así como con el de los sistemas de ficheros distribuidos NFSv4 y PVFS2. 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.

En primer lugar se ha evaluado el rendimiento “puro” de *Memcached*, antes de utilizarlo como sistema de ficheros distribuido, en función del tamaño de las inserciones y extracciones de valores de la caché distribuida.

Como segundo procedimiento, se ha evaluado el rendimiento del sistema de ficheros distribuido desarrollado tanto en lecturas y escrituras secuenciales como aleatorias, obteniendo unos resultados satisfactorios. El rendimiento de MemcachedFS es bastante competitivo con respecto a todos los casos estudiados, sobre todo con respecto a un sistema de ficheros local. En ocasiones se muestra más fuerte que la competencia y en ocasiones más débil, pero salvo casos muy concretos, obtiene unos resultados cercanos al resto de los evaluados. Además, configurando de forma adecuada su tamaño de bloque en función de las necesidades, se puede conseguir aprovechar al máximo su rendimiento.

La última evaluación de rendimiento realizada ha sido una prueba de estrés, es decir, se ha evaluado el rendimiento de MemcachedFS bajo grandes cargas de trabajo.

A pesar de que el rendimiento cae cuando la carga es muy alta (algo completamente normal debido al sistema de gestión de colisiones de Ethernet) se ha cumplido el objetivo de la prueba, que era asegurar su fiabilidad, manteniéndose un funcionamiento correcto a pesar de la pérdida de rendimiento y mejorando el rendimiento a medida que aumentaba la escalabilidad del sistema.

Como puntos fuertes de MemcachedFS con respecto a la competencia, se ha observado que su rendimiento depende exclusivamente de las características de la red sobre la que esté trabajando, mientras que en el resto de casos influye el dispositivo de almacenamiento sobre el que se gestione la persistencia, y la enorme capacidad de escalabilidad que ofrece *Memcached* de cara a su utilización en un sistema de ficheros distribuidos. En contrapartida, MemcachedFS es aún un sistema de ficheros experimental que carece de muchas funcionalidades como permisos de usuarios, directorios, etc. y, sobre todo, aún no está pulida la persistencia de datos.

En definitiva, se trata de un Proyecto Fin de Carrera muy completo. Se ha investigado y comprendido el funcionamiento de una de las plataformas más en auge en la actualidad, *Memcached*. Se ha diseñado desde cero un sistema de ficheros distribuido utilizando como base *Memcached* y aportando ciertas características novedosas al mundo de los sistemas de entrada y salida de alto rendimiento, aplicando un concepto como es el de utilizar memoria principal como espacio de almacenamiento. Se ha implementado una versión de este concepto novedoso, básica, pero completamente funcional dando lugar al sistema de ficheros distribuido MemcachedFS y se ha verificado su correcto funcionamiento. Por último, se ha realizado un estudio completo de evaluación de su rendimiento, utilizando un clúster como sistema de alto rendimiento, y se han obtenido unos resultados bastante positivos, de modo que se puede concluir que es viable utilizar *Memcached* como base para un sistema de ficheros distribuido.

9.2 Trabajos futuros

Debido al estado experimental del sistema desarrollado, ha habido una serie de funcionalidades que no han sido implementadas por motivos de coste de tiempo o porque pudieran penalizar el rendimiento dificultando la evaluación. Algunas de estas funcionalidades podrán ser desarrolladas en el futuro.

En primer lugar, **mejora de la persistencia**. Especialmente importante en los modos de caché. Para ello, sería interesante implementar algún tipo de método de *write-back* para que, de forma periódica, los datos de la caché sean devueltos a disco para evitar que se pierdan si son expulsados de la caché. De este modo, en caso de dar fallo caché alguna lectura, se podría recurrir al disco duro para recuperar la última versión del bloque. Para ello, se podría utilizar un sistema similar al "bit *dirty*" de las cachés,

implementando una cola de operaciones realizadas que se pueda recorrer periódicamente para sincronizar el contenido de MemcachedFS con el del disco.

Relacionado con los modos de caché, para mejorar el rendimiento, se podría implementar algún tipo de ***prefetching***, de modo que, cuando un bloque diese fallo caché y hubiese que llevarlo del disco a la caché distribuida, aprovechar la operación para transferir a caché distribuida los bloques contiguos que, previsiblemente, se utilizarán. Esta funcionalidad podría mejorar el rendimiento en situaciones concretas.

Para facilitar la organización y jerarquización de ficheros, conviene implementar un sistema arbóreo de **directorios** como el que tienen la mayoría de sistemas de ficheros. Asimismo, otra funcionalidad dejada de lado ha sido una implementación exhaustiva de la propiedad y los **permisos** de los ficheros. Las estructuras de MemcachedFS están preparadas para esta funcionalidad, tan solo habría que adaptar el código para añadir este tipo de comprobaciones.

Otras funcionalidades cuyo desarrollo puede ser interesante para ampliar la funcionalidad del sistema más allá de la concepción actual de MemcachedFS (y no dejadas de lado solo por ser experimental, sino porque cambiarían la filosofía de diseño del sistema) son las siguientes:

Sería muy interesante integrar el 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. Para conseguirlo, sería necesaria la implementación de un **servicio de directorio** que ofreciese un espacio de nombres único en todo el sistema.

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.

Teniendo en cuenta los resultados positivos que se han conseguido en la evaluación del sistema, también se puede plantear como trabajo futuro, la utilización de *Memcached* o, incluso, 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 podría intentar mantener el bloque de metadatos en memoria del cliente desde la apertura hasta el cerrado de un fichero y **actualizar el bloque de metadatos de forma retardada**, o bien cada cierto tiempo (o número de operaciones) o bien tan sólo en el momento de realizar el cierre.

Por último, podría resultar interesante trabajar para implementar funcionalidades que permitan la **utilización de los ficheros de forma concurrente**, de cara a conseguir un sistema de ficheros distribuido más versátil.

9.3 Método de trabajo

Se ha utilizado una metodología de trabajo basada en un ciclo de vida incremental. El objetivo es ir realizando una serie de iteraciones que producen versiones funcionales del sistema, aumentando la funcionalidad en cada una de estas iteraciones.

Figura 58: Diagrama de ciclo de vida incremental

El funcionamiento de este ciclo de vida es especialmente adecuado para este proyecto de fin de carrera, ya que, el funcionamiento de ciertas partes del sistema se basa ampliamente en otras zonas que deben funcionar a la perfección. Por tanto, es lógico dividir el desarrollo en fases, en que la funcionalidad básica sea desarrollada y probada a conciencia antes de continuar con funcionalidades más complejas.

9.3.1 Fases

Fase 1

Operaciones de entrada y salida de datos en la caché distribuida. Lectura y escritura de bloques en *Memcached*, tanto bloques de datos como bloques de metadatos. En este caso, el producto no es entregable, sólo sirve de infraestructura para el resto de funciones y como toma de contacto con la librería libMemcached.

Fase 2

Funcionalidad básica del sistema de ficheros distribuido. Funciones de apertura y cerrado de ficheros, leer, escribir, mover el puntero de datos, borrar ficheros, renombrarlos y acceder a sus metadatos. Este sería un primer producto entregable, con funcionalidad completa como sistema de ficheros distribuido.

Fase 3

Modos de cacheo de ficheros locales (pre-caché y caché). Aprovechando la funcionalidad básica se implementan funciones para utilizar el sistema de ficheros distribuido como una caché distribuida de un fichero local. Se amplía la funcionalidad del sistema y se consigue otro producto completamente entregable.

Fase 4

Utilización de la función de lectura simultánea de bloques de libMemcached (`memcached_mget`). Aprovechando que se tiene un sistema completamente estable y completo en la iteración anterior, se puede arriesgar para mejorar el rendimiento. En caso de que exista algún error con la nueva funcionalidad, se puede volver en cualquier momento al anterior producto entregado.

9.4 Presupuesto

Los siguientes datos muestran algunos datos de interés para la comprensión del presupuesto:

- **Fecha de inicio:** se establece como fecha de inicio la primera reunión de objetivos con el tutor, D. Francisco Javier García Blas, el día 14 de febrero de 2011 y la fecha de finalización prevista es el 30 de septiembre de 2011.
- **Periodo vacacional:** el periodo vacacional, se estableció entre los días 1 y 31 julio de 2011 en los que no se trabajó en el proyecto.
- **Días festivos:** también se consideraron días festivos, los días del 18 al 25 de abril (Semana Santa), el 2 de mayo, el 23 de junio y los días 15 y 16 de agosto.
- **Jornada laboral:** la jornada laboral comprende de lunes a viernes, 4 horas diarias.

Teniendo en cuenta todos estos datos, se obtienen 139 días de trabajo, que conforman un total de 556 horas.

9.4.1 Costes de personal

Para el desarrollo de todo el Proyecto Fin de Carrera ha habido un solo trabajador: el autor del mismo. Sin embargo, son varios los roles interpretados. En la siguiente tabla se pueden ver las distintas tareas, con su rol asociado y su coste desglosado.

Tarea	Rol	Coste por hora (€/h)	Total horas	Total coste (€)
Análisis	Analista	50 €/ hora	34	1.700
Diseño	Analista	50 €/ hora	54	2.700
Fase 1	Programador	30 €/ hora	30	900
Fase 2	Programador	30 €/ hora	128	3.840
Fase 3	Programador	30 €/ hora	96	2.880
Fase 4	Programador	30 €/ hora	28	840
Evaluación	Analista	50 €/ hora	86	4.300
Documentación	Analista	50 €/ hora	100	5.000
TOTAL			**556**	**22.160**

Tabla 48: Desglose presupuesto personal

9.4.2 Costes de equipos

Para el desarrollo completo del sistema, han sido necesarios dos tipos de equipo. Por un lado, equipos de desarrollo y, por otro, equipos de pruebas.

Como equipos de desarrollo se han utilizado los siguientes, de forma alterna según las necesidades:

Equipo de sobremesa

Phenom II x4 955 (@3.2 GHz 4 núcleos)

4 GB RAM DDR3 1333

Western Digital Caviar Black 640 GB 7200 rpm

Monitor Samsung 940BW

Coste total: 700 €

Equipo portátil

Sony VAIO VGN-NR10M/S

Intel Core 2 Duo T5250 (@1.5 GHz dos núcleos)

2 GB RAM DDR2

160 GB 5400 rpm

Coste total: 700 €

Los equipos nunca han sido utilizados de forma simultánea, por tanto, sólo se imputará el coste de uno de ellos, durante los siete meses que han sido utilizados (o un 50% de uso a cada uno). Por lo tanto, contando una amortización de 60 meses (unos cinco años de duración del equipo) el coste de utilizarlos durante el proyecto asciende a 40,83 € por cada equipo, 81,67 € en total.

Clúster

El clúster sólo ha sido utilizado durante el mes de pruebas, sin embargo, ha sido cedido de forma totalmente gratuita por la universidad. Las características principales de los equipos del clúster son las siguientes:

24 Intel(R) Xeon(R) CPU E5405 @ 2.00GHz

4 GByte de memoria RAM

Tarjeta Gigabit Ethernet

Ubuntu 10.10 Maverick Versión server

Disco duro de 500GByte

Equipo	Coste	Uso	Coste uso
Ordenador portátil	700,00	7 meses (50%)	40,83
Ordenador sobremesa	700,00	7 meses (50%)	40,83
Clúster	0,00	1 mes (100%)	0,00
TOTAL			**81,66**

Tabla 49: Desglose coste equipos

9.4.3 Costes de Software

El desarrollo de la librería ha sido realizado completamente mediante software libre y gratuito. Utilizando Geany como entorno de desarrollo y gcc como compilador corriendo sobre Ubuntu 11.04, tanto en el ordenador portátil como en el de sobremesa. El clúster de pruebas también corría sobre una versión gratuita de Linux, Ubuntu 10.10 Maverick Meerkat, en su versión para servidores.

Con respecto Memcached y libMemcached, ambos son software de código abierto (open source), por lo tanto, su utilización no supone ningún coste, así como la librería de SHA2 utilizada.

De cara a la documentación, se ha utilizado Doxygen para documentar el código (gratuito) y Office 2007 para llevar a cabo la memoria (tanto el documento de texto con Word, como los gráficos con Excel) corriendo sobre una versión de estudiante de Windows 7 Professional.

Por tanto, el único coste de software imputable a este proyecto es el de la licencia de Office 2007 Hogar y Estudiantes, de **79,95 €.**

9.4.4 Material Fungible

El coste de impresión de la documentación, así como otras impresiones necesarias para el proyecto (borradores, bocetos, comunicaciones con el tutor) han supuesto un coste en torno a los 200 euros.

9.4.5 Costes indirectos

Por último, se debe tener en cuenta que existen una serie de costes derivados del uso de instalaciones: electricidad, internet, conexiones, instalación de software específico en ordenadores de la universidad, utilización del clúster para pruebas, docencia, etc.

La estimación de estos costes se encuentra en torno al 20% del coste total del proyecto.

9.4.6 Coste total del proyecto

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor: Francisco José Rodrigo Duro

2.- Departamento:

3.- Descripción del Proyecto:

- Titulo: **Diseño e implementación de un sistema de ficheros distribuido basado en Memcached**
- Duración (meses): **6,5**
- Tasa de costes Indirectos: **20%**

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

Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Rodrigo Duro, Francisco José	NIF	Analista	2,0876	6.562,50	13.699,88	
Rodrigo Duro, Francisco José	NIF	Programador	2,1485	3.937,50	8.459,72	
		Hombres mes	**4,2361**	**Total**	22.159,59	

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

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Ordenador Portátil	700,00	50	7	60	40,83
Odenador Sobremesa	700,00	50	7	60	40,83
Clúster	0,00	100	1	60	0,00
				Total	81,67

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} \text{xCxD}$$

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

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Licencia Office 2007 Hogar y Estudia		79,95
Material Fungible		200,00
	Total	279,95

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

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	22.160
Amortización	82
Subcontratación de tareas	0
Costes de funcionamiento	280
Costes Indirectos	4.504
Total	27.025

Figura 59: Presupuesto inicial atendiendo a la plantilla de Rúbrica

El coste total del proyecto, por tanto asciende a **veintisiete mil veinticinco euros con noventa y tres céntimos (27.025,93)**.

Concepto	Coste
Recursos Humanos	22.160,00
Equipos	81,66
Software	79,95
Material Fungible	200,00
Costes indirectos	+20%
TOTAL	**27.025,93**

Tabla 50: Desglose coste total del proyecto

Leganés, a 4 de Septiembre de 2011

El ingeniero proyectista

Fdo. Francisco José Rodrigo Duro

10 BIBLIOGRAFÍA Y REFERENCIAS

[1] *Conferencia: Youtube Scalability*. **Seattle Google Tech Talks.** http://video.google.com/videoplay?docid=-6304964351441328559 Vídeo online (min. 26) consultado a 3 de agosto de 2011.

[2] *Scaling Memcached at Facebook (Escalando Memcached en Facebook).* **Paul Saab (Facebook).** http://www.facebook.com/note.php?note_id=39391378919&ref=mf Recurso online consultado a 3 de agosto de 2011.

[3] *It's not rocket science, but it's our work (No es ciencia de cohetes, pero es nuestro trabajo).* **Jack Dorsey and Biz Stone.** http://blog.twitter.com/2008/05/its-not-rocket-science-but-its-our-work.html Recurso online consultado a 3 de agosto de 2011.

[4] *Using UDP in Memcached (Utilizando UDP en Memcached).* **Jaime Medrano.** http://blog.tuenti.com/dev/using-udp-in-memcached/ Recurso online consultado a 3 de agosto de 2011.

[5] *Linux Programmer's Manual*. **Comando 'man' Linux.**

[6] *Sistemas Operativos: Una visión aplicada.* **Jesús Carretero, Félix García Carballeira, Pedro de Miguel, Fernando Pérez.** Editorial McGraw-Hill. 2001.

[7] *Crucial m4 SSD Review 256GB (Análisis del SSD Crucial m4 256GB).* **Kevin O'Brien (Storage Review).** http://www.storagereview.com/crucial_m4_ssd_review_256gb Recurso online consultado a 9 de agosto de 2011.

[8] *Western Digital Caviar Black Review 2TB (Análisis del Western Digital Caviar Black 2TB).* **Storage Review.** http://www.storagereview.com/western_digital_caviar_black_review_2tb Recurso online consultado a 9 de agosto de 2011.

[9] *Memcached project website.* http://www.memcached.org Recurso online consultado a 10 de agosto de 2011.

[10] *PHP Accelerators.* **Marius Ducea.** http://www.ducea.com/2006/10/30/php-accelerators/ Recurso online consultado a 10 de agosto de 2011.

[11] *Memcached FAQ.* **Contribuidores múltiples.** http://code.google.com/p/memcached/wiki/FAQ Recurso online consultado a 10 de agosto de 2011.

[12] *Memcached on membase website.* http://www.couchbase.com/products-and-services/memcached Recurso online consultado a 10 de agosto de 2011.

[13] *Redis vs Memcached (slightly better bench).* **dormando.** http://dormando.livejournal.com/525147.html Recurso online consultado a 10 de agosto de 2011.

[14] *Redis vs Memcached.* http://systoilet.wordpress.com/2010/08/09/redis-vs-memcached/ Recurso online consultado a 10 de agosto de 2011.

[15] *Sitio web oficial de libMemcached.* **Brian Aker.** http://libmemcached.org/libMemcached.html Recurso online consultado a 11 de agosto de 2011.

[16] Implementations of SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. **Aaron Gifford.** http://www.aarongifford.com/computers/sha.html Recurso online consultado a 26 de agosto de 2011.

ANEXO I: MANUAL DE INSTALACIÓN

En este anexo se detallarán los pasos necesarios para utilizar el sistema de ficheros distribuido que se ha implementado en este proyecto fin de carrera.

Paso 1: Instalación de Memcached

Memcached debe ser instalado en todos los nodos del sistema que quieran ser utilizados como servidores del sistema de ficheros distribuido.

Método 1: Synaptic

En todos los sistemas que han sido utilizados para llevar a cabo este proyecto fin de carrera, se disponía de Synaptic (viene por defecto con todas las distribuciones de Debian y derivados como Ubuntu), por lo tanto, se recomienda el uso de este método de instalación de *Memcached*.

En primer lugar se arranca el "Gestor de paquetes Synaptic", que se puede encontrar en la zona de "Administración", dentro del menú de "Sistema".

Acto seguido, se busca "memcached" mediante el cuadro de la esquina superior derecha y se selecciona para instalar el paquete "memcached". Por último, se pincha sobre el botón "Aplicar".

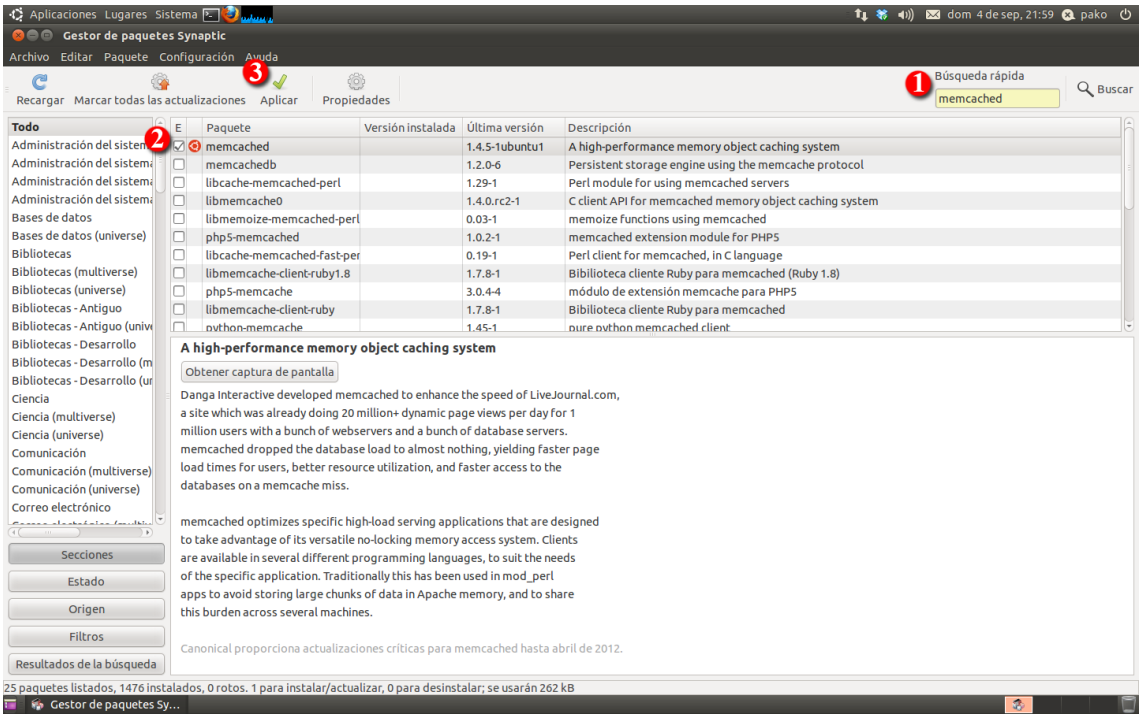

Figura 60: Captura de pantalla instalación de Memcached

El instalador hará una serie de comprobaciones y requerirá acciones del usuario: introducción de contraseña de administrador (‘root’), aceptación de la instalación y cerrado de la ventana en la que se informa de una instalación satisfactoria.

A partir de este momento, *Memcached* podrá ser lanzado en línea de comandos desde cualquier directorio del sistema, tan solo escribiendo “`memcached`” y las opciones que se deseen.

Método 2: manual (línea de comandos)

Para poder instalar *Memcached*, en primer lugar es necesario tener instalada y configurada la librería libevent, que se puede encontrar en el directorio ‘essentials’.

Para conocer si está instalada, bastará con ejecutar el siguiente comando:

```
whereis libevent
```

En caso de utilizar la contenida en dicho directorio, se puede obviar el primer paso (`wget`) del siguiente manual de instalación:

```
wget http://monkey.org/~provos/libevent-2.0.13-stable.tar.gz
tar xfz libevent-2.0.13-stable.tar.gz
```

```
cd libevent-2.0.13-stable
./configure
make
sudo make install
```

El último paso para tener operativa la librería libevent es apuntarla mediante un enlace simbólico para que los compiladores puedan encontrarla.

```
sudo ln -s /usr/local/lib/libevent-2.0.so.5 /usr/lib
```

Con estos pasos, libevent está instalado y listo para ser utilizado. Se puede en este momento comenzar la instalación de Memcached, que será muy sencilla siguiendo los pasos (de nuevo, se puede saltar el primer paso si se opta por instalar la versión contenida en 'essentials'):

```
wget http://memcached.googlecode.com/files/memcached-1.4.7.tar.gz
tar xfz memcached-1.4.7
cd memcached-1.4.7
./configure
make
sudo make install
```

A partir de este momento, *Memcached* está instalado en el sistema y listo para ser lanzado desde cualquier ruta.

Paso 2: Instalación libMemcached

La librería para la comunicación con libMemcached, también puede ser instalada mediante el gestor de paquetes Synaptic, de forma muy similar al método expuesto para *Memcached*. De este modo, para compilar cualquier código fuente que incluya la librería libMemcached, tan sólo habría que poner al final de la línea de compilación la opción "`-lmemcached`". Este es el método utilizado en el código incluido con el proyecto, por tanto, se recomienda utilizar esta opción.

Para la instalación en modo línea de comandos, en primer lugar se debe descargar la última versión de libMemcached del sitio web oficial (https://launchpad.net/libmemcached/+download) o utilizar la contenida en el directorio

'essentials' de los materiales entregados junto con el proyecto fin de carrera (libmemcached-0.51.tar.gz).

Una vez descargada o copiada al lugar que se desee, se descomprime, utilizando el comando:

```
tar xfz libmemcached-0.51.tar.gz
```

El siguiente paso es configurarla y compilarla. Para ello, se debe elegir un directorio sobre el que se tengan permisos de lectura y escritura, en el que se instalará la librería. Por ejemplo, se va a utilizar un directorio típico con estos permisos como puede ser "$HOME/libmemcached". En caso de querer instalar la librería en cualquier otro directorio, tan solo habrá que cambiar esta ruta por cualquier otra en todos los lugares que aparezca.

Para configurar y compilar, el primer paso es situarse dentro del directorio en que se haya descomprimido la librería. Una vez situado en ese directorio, ejecutar los siguientes comandos:

```
./configure --prefix=$HOME/libmemcached --enable-static --with-
    memcached

make

make install
```

Por último, se indica al sistema operativo dónde encontrar la librería:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/libmemcached/lib
```

En este punto, la librería está instalada y configurada para ser utilizada.

Paso 3: Compilación de código cliente

En este paso se indica la forma correcta de incluir las librerías desarrolladas en el código fuente y enlazarlas en el proceso de compilación.

El primer paso es incluir las librerías en el código fuente, para ello, se debe utilizar la ruta relativa al directorio en que se encuentre el código fuente a compilar. Por ejemplo, si la librería se encuentra en la carpeta "lib", al inicio de todos los ficheros de código fuente del cliente debería aparecer la línea:

```
#include "lib/memcachedfs.h"
```

Las opciones de compilación a utilizar son las siguientes, en un ejemplo en que el código fuente del cliente se encontrase en un fichero client.c:

```
gcc -o client client.c lib/memcachedfs.c lib/utils.c
    lib/libs/sha2/sha2.c -L$HOME/libmemcached/lib -lmemcached -I
    $HOME/libmemcached/include
```

La explicación es sencilla, en primer lugar se debe compilar todo el código de la librería desarrollada, por eso se indica dónde se encuentran todos sus ficheros de código fuente, incluso el de la librería de SHA2. Justo después, se indica dónde puede encontrar el compilador la librería libMemcached para enlazarla, tanto los ficheros de librería (`-L$HOME/libmemcached/lib`) como los ficheros de cabecera y su código asociado que son incluidos (`-I $HOME/libmemcached/include`).

En caso de haber instalado la librería mediante el gestor de paquetes Synaptic, la línea de compilación se reduciría a:

```
gcc -o client client.c lib/memcachedfs.c lib/utils.c
    lib/libs/sha2/sha2.c -lmemcached
```

Por último, para utilizar cualquier cliente, *Memcached* debe ser lanzado en uno o varios servidores y, éstos, deben ser conocidos y accesibles mediante red por los clientes.

Si se desea cambiar el tamaño de bloque del sistema de ficheros distribuido, es tan sencillo como abrir el fichero `utils.h` y modificar la constante **`BLOCK_SIZE`** con el valor en Kbyte deseado (entre 1 y 1023). Para que los cambios tengan efecto se debe recompilar el código.

AVISO: los ficheros de MemcachedFS no podrán ser accedidos con un tamaño de bloque distinto de aquel con el que fueron creados.

ANEXO II: CREACIÓN DE SCRIPTS PARA CLÚSTER

Dado que el uso habitual del sistema de ficheros distribuido desarrollado será su utilización en sistemas de alto rendimiento como clústeres, en este anexo se explica cómo generar scripts para la ejecución de programas en un clúster.

Para entender mejor el funcionamiento, se explicará paso por paso cómo se ha escrito uno de los scripts utilizados para la evaluación del sistema, en concreto, el utilizado para probar las lecturas y escrituras de ficheros con cuatro servidores. A continuación, el contenido del fichero de script (la numeración de las líneas ha sido incluida para facilitar la explicación del script, no debe incluirse en el fichero de script final):

```
001 #PBS -N test_write_read_4
002 #PBS -o test_write_read_4.o.$PBS_JOBID
003 #PBS -e test_write_read_4.e.$PBS_JOBID
004 #PBS -q normal
005 #PBS -l nodes=5
006 #PBS -l walltime=00:15:00
007
008 cd $PBS_O_WORKDIR
009 srv=`cat $PBS_NODEFILE`
010
011 i=0
012
013 for x in $srv
014 do
015   arr[$i]="$x"
016   i=$i+1
017 done
018
019 ssh ${arr[0]} "memcached -m 512 -d -p 11212"
020 ssh ${arr[1]} "memcached -m 512 -d -p 11212"
021 ssh ${arr[2]} "memcached -m 512 -d -p 11212"
022 ssh ${arr[3]} "memcached -m 512 -d -p 11212"
023
024 ssh ${arr[4]} "$HOME/codigo/stats/test_write_read

      {arr[0]}:11212,${arr[1]}:11212,${arr[2]}:11212,${arr[3]}:1
      1212 04srv_128k"
025
026 ssh ${arr[0]} "killall -9 memcached"
027 ssh ${arr[1]} "killall -9 memcached"
028 ssh ${arr[2]} "killall -9 memcached"
029 ssh ${arr[3]} "killall -9 memcached"
```

En las primeras seis líneas, que van precedidas por el parámetro '#PBS', se solicita al sistema de colas del clúster una serie de nodos para la ejecución del programa y se configura cómo debe ser realizada la ejecución del trabajo.

En la línea 001, mediante el parámetro **`-N`** se indica el nombre del trabajo, para que sea más fácilmente reconocible por el usuario en la cola de ejecución.

En la línea 002, mediante el parámetro **`-o`** se indica al clúster dónde debe volcar la salida por pantalla (`stdout`) del trabajo. De este modo, es más sencillo analizar a posteriori las trazas. Como nombre del fichero de salida se ha seleccionado el nombre del trabajo ".o" y, a continuación aparece la cadena '`$PBS_JOBID`'. Las cadenas precedidas por '$' en el lenguaje `bash` se refieren al contenido de una variable, por tanto, el nombre del fichero terminará con el contenido de la variable **`PBS_JOBID`**, que será el identificador que el sistema de colas haya concedido a este trabajo. La motivación de incluirlo en el nombre de fichero, no es otra que evitar sobrescribir trazas antiguas o trazas de trabajos que se ejecuten simultáneamente. El contenido de la tercera línea es equivalente a ésta, pero utilizando el parámetro **`-e`** para redirigir la salida de errores (`stderr`).

En la línea 004, se utiliza el parámetro **`-q`** para indicar la cola a la que queremos añadir el trabajo. En este caso, el trabajo no tiene unas características específicas, por lo que se añade a la cola 'normal'.

En las líneas 005 y 006 se especifican características de los nodos seleccionados mediante el parámetro **`-l`**, con `nodes=5` se seleccionan cinco nodos (los cuatro servidores de *Memcached* y el nodo que ejecutará la prueba), mientras que con `walltime` se indica el tiempo máximo que se espera que dure el trabajo. Este parámetro se introduce para evitar sobrecargar el clúster si alguna prueba tiene errores como bucles infinitos.

A partir de este punto, todas las líneas se dirigen a la ejecución del trabajo en sí misma, los nodos ya deberían estar disponibles para ser utilizados y el clúster configurado para el trabajo.

En primer lugar, en la línea 008 se utiliza el conocido comando '`cd`' para situarse en el directorio por defecto de trabajo del clúster y, en la siguiente línea, se guarda el contenido del fichero `PBS_NODEFILE` en la variable `srv`. Este fichero contiene el nombre de los nodos reservados por el clúster para el trabajo, para que se pueda conectar con ellos. Para poder guardarlo en la variable, se utiliza el comando '`cat`' para imprimir el fichero por la salida estándar y las comillas especiales ´valor´ (acento agudo) para capturar dicha salida estándar y guardarla en la variable.

Entre las líneas 011 y 017 se procesa la variable srv en la que se encuentran los servidores escritos en una sola cadena de caracteres para dividirla en un array (arr) en el que cada servidor ocupe una de las posiciones para poder ser accedido más fácilmente.

Una vez terminado el procesamiento, se podrá acceder al nombre de cada uno de los servidores tan solo utilizando la expresión `${arr[x]}` donde x es una de las posiciones del array (en este caso, dado que se han solicitado cinco servidores, los valores de x irán desde 0 para el primer servidor hasta 4 para el último).

A partir de este punto, se pueden lanzar aquellos comandos que se deseen y en el servidor que se prefiera de los reservados, para ello, se utiliza el comando **`ssh`** (Secure Shell) para conectar con el servidor, seguido del comando que se quiera utilizar.

En este caso, se requiere lanzar cuatro servidores de *Memcached* (lanzados entre las líneas 019 y 022). Cabe destacar que se lanzan como demonios (`-d`), ya que, en caso contrario la prueba se quedaría parada en la primera ejecución porque no se ejecuta la siguiente línea hasta que no termina la anterior (en modo normal, `memcached` se queda parado esperando a recibir peticiones, por tanto, hay que lanzarlo en *background* para no ocupar el terminal).

En la línea 024 se lanza la prueba indicando en qué servidores quiere lanzarse (que deben coincidir con aquellos en los que se está ejecutando *Memcached* y en los puertos de escucha configurados, en este caso, 11212).

Por último, se debe matar los procesos de demonio de `memcached` para dejar los nodos del clúster en el mismo estado que estaban antes de iniciarse la prueba y para evitar problemas a la hora de lanzar otra prueba en los mismos servidores. Para ello, se utiliza el comando `kill -9` del proceso `memcached`. El archivo debe guardarse en el disco con la extensión '`.sh`' y el nombre que se desee.

Para la ejecución del script, se debe utilizar el sistema de colas.

`qsub <nombre_fichero>` introduce el trabajo en la cola. El nombre del fichero será el script (`.sh`) creado siguiendo los pasos anteriores.

`qstat` muestra por pantalla el estado de la cola.

`qdel <id_trabajo>` borra de la cola el trabajo con identificador `<id_trabajo>`.

Una vez que termine el trabajo, se podrá consultar el resultado imprimiendo por pantalla mediante el comando '`cat`' los ficheros `<nombre_trabajo>.e.<id_trabajo>` y `<nombre_trabajo>.o.<id_trabajo>`.