

Índice de Contenidos

Índice de contenidos	i
Relación de Acrónimos	v
Índice de Figuras	vii
Índice de Tablas	x
CAPÍTULO 1: Introducción	1
CAPÍTULO 2: Fundamentos de <i>cache</i> Web	6
2.1. CONCEPTO DE SERVIDOR <i>PROXY</i>	6
2.2. CONCEPTO DE <i>CACHE</i>	9
2.3. HTTP Y LA <i>CACHE</i> WEB	17
2.3.1. PETICIONES HTTP	19
2.3.1.1. Peticiones al servidor origen	20

2.3.1.2. Peticiones a un servidor <i>proxy</i>	20
2.3.1.3. Peticiones no-HTTP a un servidor <i>proxy</i>	21
2.3.2. DOCUMENTOS ALMACENABLES EN <i>CACHE</i>	22
2.3.2.1. Códigos de estado	23
2.3.2.2. Métodos de petición	24
2.3.2.3. Expiración y validación	25
2.3.2.4. La cabecera <i>Cache-control</i>	26
2.3.2.5. Autenticación	29
2.3.2.6. Cookies	30
2.3.2.7. Contenido dinámico	31
2.3.3 GESTIÓN DE ACIERTOS Y FALLOS EN <i>CACHE</i> Y SU RELACIÓN CON LA FRESCURA Y VALIDACIÓN DE LOS DOCUMENTOS	32
2.4. POLÍTICAS DE REEMPLAZO PARA <i>PROXYS</i> WEB	36
2.4.1. LEAST RECENTLY USED (LRU)	37
2.4.2. LEAST FREQUENTLY USED (LFU)	38
2.4.3. GREEDY DUAL SIZE (GDS)	38
2.4.4. GREEDY DUAL SIZE FREQUENCY (GDSF)	40
2.4.5. GREEDY DUAL* (GD*)	41
CAPÍTULO 3: Emulador <i>Proxy</i>	44
3.1. INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS	44
3.1.1. PRINCIPIOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS	46
3.1.1.1. Principio de abstracción	46
3.1.1.2. Principio de encapsulamiento	47
3.1.1.3. Principio de modularidad	47
3.1.1.4. Principio de jerarquía	48
3.1.1.5. Principio de paso de mensajes	48
3.1.1.6. Principio de polimorfismo	48
3.2. EL LENGUAJE DE PROGRAMACIÓN JAVA	49
3.2.1. CARACTERÍSTICAS DEL LENGUAJE JAVA	49
3.2.1.1. Simplicidad	49
3.2.1.2. Orientado a objetos	49
3.2.1.3. Distribuido	50
3.2.1.4. Robusto	50

3.2.1.5. Interpretado y compilado a la vez	50
3.2.1.6. Indiferente de la arquitectura	51
3.2.1.7. Portable	51
3.2.1.8. Multihebra	51
3.3. ESTRUCTURA DEL EMULADOR DISEÑADO	52
3.3.1. DESCRIPCIÓN DEL INTERFAZ CON EL USUARIO	52
3.3.1.1. Botones para el manejo de los archivos con muestras de tráfico	53
3.3.1.2. Panel de trazas seleccionadas	54
3.3.1.3. Área para la selección del tamaño de <i>cache</i>	54
3.3.1.4. Área para la selección de la función de coste	55
3.3.1.5. Botones para el control del funcionamiento del emulador	56
3.3.1.6. Botón de ayuda	57
3.3.1.7. Área para la selección de la política de reemplazo	57
3.3.1.8. Área para la selección del tipo y subtipo/s de documentos a almacenar en <i>cache</i>	57
3.3.1.9. Mensaje de final de ejecución	58
3.3.2. IMPLEMENTACIÓN DEL EMULADOR	59
3.3.3. PRUEBAS REALIZADAS PARA VERIFICAR EL CORRECTO FUNCIONAMIENTO DEL EMULADOR	64
3.3.4. RECOMENDACIONES PARA EL USO DEL EMULADOR	67
CAPÍTULO 4: Estudio comparativo de rendimiento según el tipo de documento	69
4.1. METODOLOGÍA DE ESTUDIO Y MÉTRICAS EMPLEADAS	69
4.2. CARACTERIZACIÓN DE LAS MUESTRAS DE TRÁFICO	73
4.3. COMPARACIÓN DEL RENDIMIENTO PARA CADA UNO DE LOS TIPOS DE DOCUMENTOS	76
4.3.1. APLICACIONES	77
4.3.1.1. Resultados para todos los subtipos	77
4.3.1.2. Resultados para los subtipos más habituales	79
4.3.1.3. Resultados para subtipos con mayor HR y BHR	81
4.3.2. AUDIO	85
4.3.2.1. Resultados para todos los subtipos	85
4.3.2.2. Resultados para los subtipos más habituales	87

4.3.2.3. Resultados para subtipos con mayor HR y BHR	90
4.3.3. IMÁGENES	94
4.3.3.1. Resultados para todos los subtipos	94
4.3.3.2. Resultados para los subtipos más habituales	96
4.3.4. TEXTO	98
4.3.4.1. Resultados para todos los subtipos	98
4.3.4.2. Resultados para los subtipos más habituales	100
4.3.4.3. Resultados para subtipos con mayor HR y BHR	102
4.3.5. VÍDEO	104
4.3.5.1. Resultados para todos los subtipos	104
4.3.5.2. Resultados para los subtipos más habituales	106
4.3.5.3. Resultados para subtipos con mayor HR y BHR	108
CAPÍTULO 5: Conclusiones y líneas futuras	113
5.1. CONCLUSIONES FINALES	113
5.2. LÍNEAS FUTURAS	116
Bibliografía	117
Apéndice A. Diagramas UML	121
Apéndice B. Ejemplo de verificación del funcionamiento del emulador	126
Apéndice C. Codificación de tipos y subtipos de documentos	133

Relación de Acrónimos

ASCII	<i>American Standard Code for Information Interchange</i>
ASP	<i>Active Server Pages</i>
BHR	<i>Byte Hit Rate</i>
CGI	<i>Common Gateway Interface</i>
CPU	<i>Central Process Unit</i>
DNS	<i>Domain Name Service</i>
FTP	<i>File Transfer Protocol</i>
GD*	<i>Greedy Dual*</i>
GDS	<i>Greedy Dual Size</i>
GDSF	<i>Greedy Dual Size Frequency</i>
GIF	<i>Graphics Interchange Format</i>
GMT	<i>Greenwich Mean Time</i>
HR	<i>Hit Rate</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>HyperText Transfer Protocol</i>
IP	<i>Internet Protocol</i>

JPEG	<i>Joint Photographic Experts Group</i>
JRE	<i>Java Runtime Enviroment</i>
JVM	<i>Java Virtual Machine</i>
LFU	<i>Least Frequently Used</i>
LRU	<i>Least Recently Used</i>
MTU	<i>Maximum Transfer Unit</i>
NLANR	<i>National Laboratory for Applied Network Research</i>
RAM	<i>Random Access Memory</i>
RFC	<i>Request For Comments</i>
TCP	<i>Transfer Control Protocol</i>
UML	<i>Unified Modeling Lenguaje</i>
URI	<i>Universal Resource Identifier</i>
URL	<i>Uniform Resource Locator</i>
WAIS	<i>Wide Area Information Service</i>
WWW	<i>World Wide Web</i>

Índice de Figuras

Figura 2.1. Ejemplo de acceso a Internet a través de servidor <i>proxy</i> _____	7
Figura 2.2. Ejemplo de transacción a través de <i>proxy</i> _____	8
Figura 2.3. Ejemplo de uso de <i>cache proxy</i> _____	9
Figura 2.4. Ejemplo de acierto en <i>cache</i> _____	12
Figura 2.5. Estructura de los mensajes de petición y respuesta HTTP _____	17
Figura 2.6. Algoritmo Greedy Dual Size _____	40
Figura 3.1. Ejemplo de árbol de herencia _____	48
Figura 3.2. Pantalla principal de la aplicación _____	52
Figura 3.3. Botones para manejar los archivos con muestras de tráfico _____	53
Figura 3.4. Ventana para cargar archivos _____	53
Figura 3.5. Ejemplo de panel de trazas seleccionadas con un ejemplo de selección _____	54
Figura 3.6. Área para la selección del tamaño de <i>cache</i> _____	55

Figura 3.7. Área para la selección de la función de coste	55
Figura 3.8. Botones para el control del funcionamiento del emulador	56
Figura 3.9. Ejemplo de mensaje de error	56
Figura 3.10. Botón de ayuda	57
Figura 3.11. Menú desplegable para la elección de la política de reemplazo	57
Figura 3.12. Pestañas correspondientes a los distintos tipos de documentos	58
Figura 3.13. Mensaje de fin de ejecución	58
Figura 3.14. Diagrama de flujo de “Emulador”	62
Figura 4.1. HR (a) y BHR (b) para todos los subtipos de Aplicación	78
Figura 4.2. HR (a) y BHR (b) para los subtipos más habituales de Aplicación	80
Figura 4.3. HR (a) y BHR (b) para los subtipos para mayor HR de Aplicación	82
Figura 4.4. HR (a) y BHR (b) para los subtipos para mayor BHR de Aplicación	84
Figura 4.5. HR (a) y BHR (b) para todos los subtipos de Audio	86
Figura 4.6. HR (a) y BHR (b) para los subtipos más habituales de Audio	89
Figura 4.7. HR (a) y BHR (b) para los subtipos para mayor HR de Audio	91
Figura 4.8. HR (a) y BHR (b) para los subtipos para mayor BHR de Audio	93
Figura 4.9. HR (a) y BHR (b) para todos los subtipos de Imágenes	95
Figura 4.10. HR (a) y BHR (b) para los subtipos más habituales de Imágenes	97
Figura 4.11. HR (a) y BHR (b) para todos los subtipos de Texto	99
Figura 4.12. HR (a) y BHR (b) para los subtipos más habituales de Texto	101
Figura 4.13. HR (a) y BHR (b) para los subtipos para mayor HR y BHR de Texto	103
Figura 4.14. HR (a) y BHR (b) para todos los subtipos de Vídeo	105
Figura 4.15. HR (a) y BHR (b) para los subtipos más habituales de Vídeo	107
Figura 4.16. HR (a) y BHR (b) para los subtipos para mayor HR de Vídeo	110
Figura 4.17. HR (a) y BHR (b) para los subtipos para mayor BHR de Vídeo	112

Figura A.1. Diagrama UML de “emulcacheproxy”	122
Figura A.2. Diagrama UML de “Emulador”	123
Figura A.3. Diagrama UML de “MiLista”	124
Figura A.4. Diagrama UML de “ListaGDS”	125

Índice de Tablas

Tabla 2.1. Métodos de petición HTTP definidos por la RFC 2616 _____	19
Tabla 2.2. Grupos en los que se dividen los códigos de estado de las respuestas HTTP __	23
Tabla 2.3. Códigos de respuesta almacenables en <i>cache</i> _____	24
Tabla 4.1. Características de las muestras de tráfico según tipo de documento _____	75
Tabla 4.2. Características de los subtipos “más habituales” de aplicación _____	79
Tabla 4.3. Aportación al HR y BHR de los subtipos “más habituales” de aplicación ____	81
Tabla 4.4. Características de los subtipos “más habituales” de audio _____	88
Tabla 4.5. Aportación al HR y BHR de los subtipos “más habituales” de audio _____	90
Tabla 4.6. Características de los subtipos “más habituales” de imágenes _____	96
Tabla 4.7. Características de los subtipos “más habituales” de texto _____	100
Tabla 4.8. Aportación al HR y BHR de los subtipos “más habituales” de texto _____	102
Tabla 4.9. Características de los subtipos “más habituales” de vídeo _____	106
Tabla 4.10. Aportación al HR y BHR de los subtipos “más habituales” de vídeo _____	108

CAPÍTULO 1: Introducción

Cada día las páginas WWW (*World Wide Web*) tienen una mayor popularidad, lo que lleva consigo también un cada vez mayor tráfico debido a las peticiones de los usuarios.

La *cache* tiene la misión de almacenar aquellos documentos que son solicitados por los usuarios con mayor asiduidad. Ante una nueva petición de un determinado documento, ya sea por parte del mismo o de distintos usuarios, dicho documento podrá ser servido directamente de la *cache* con el consiguiente beneficio. Antes de procederse a ello tendrá que comprobarse que el documento no haya expirado y de ser así para poder ser servido, tendrá que ser validado porque en caso contrario requerirá la conexión con el servidor remoto para ser obtenido de nuevo tanto para la *cache* como para el usuario que lo ha solicitado.

Los recursos Web se pueden dividir en estáticos y dinámicos. Ante una petición de dichos recursos por parte de un usuario, las respuestas para recursos dinámicos se generan al vuelo y de forma personalizada para cada usuario. Las respuestas estáticas, sin embargo, son generadas previas e independientes de la petición del usuario. La distinción entre recursos estáticos y dinámicos es importante porque tiene influencia sobre la consistencia

de la *cache*. Dicha consistencia consiste, básicamente, en que la *cache* pueda disponer en todo momento de las copias actualizadas de los documentos que mantiene almacenados en ella así como en poder saber en qué momento dichos documentos pasaron a ser obsoletos y dejaron de ser frescos, porque no se han actualizado.

Muchos de los documentos que son accedidos en la Web son estáticos, como documentos de audio, vídeo o las propias páginas Web entre otros y se encuentran en servidores remotos, también denominados servidores origen de la respuesta (aquellos servidores donde se encuentra la copia original del documento al que se desea acceder). Si estos documentos estuviesen disponibles en una *cache* se podría reducir el tráfico originado por dichas peticiones. Esto se podría ver como una *cache* de primer nivel que estaría ubicada en el navegador del usuario.

Si además tenemos en cuenta que es muy habitual que un conjunto de usuarios se conecten a Internet a través de un servidor *proxy* y que en éste también se puede hacer uso de una *cache* (que en este caso sería equivalente a una *cache* de segundo nivel) los beneficios que se podrían conseguir serían aún mayores.

Los servidores *proxy* actúan como intermediarios entre los servidores Web de Internet y los usuarios. Suelen tener como misión el proporcionar a los usuarios de una determinada red, un acceso a Internet seguro, desde dentro de un cortafuegos, encargándose de recibir las peticiones y cursarlas a los servidores Web remotos para, una vez recibida la respuesta, reenviarla al usuario correspondiente.

Por tanto, a través de ellos fluyen todas las peticiones de los usuarios y haciendo uso en ellos de una *cache* (que en este caso estaría asociada a un conjunto de usuarios, mientras que la del navegador está sólo asociada a un usuario) se podrían conseguir unos importantes beneficios.

Algunos de estos beneficios pueden ser la reducción del tráfico de la red, la reducción de la latencia media de las comunicaciones, es decir, una reducción en el tiempo que transcurre entre la petición del documento y la recepción de la respuesta con dicho documento, y la reducción de la carga soportada por los servidores Web. De este modo se podría conseguir que con la adecuada política de reemplazo y el adecuado tamaño de

cache, buena parte de las peticiones pudiesen ser satisfechas sin necesidad de generar tráfico hacia y desde los servidores Web y además en un menor tiempo.

Por otro lado, las *caches* situadas en los servidores *proxy* presentan una problemática añadida pues a diferencia de las memorias *cache* de los ordenadores, los documentos a almacenar se caracterizan por tener un tamaño variable y por provenir de distintos flujos de distintos usuarios con lo que no suele ajustarse a un modelo concreto.

El objetivo de este Proyecto Fin de Carrera es la evaluación de distintas políticas de reemplazo asociadas a su comportamiento en servidores *proxy* Web, y más concretamente relacionado con el uso de la *cache* de dichos servidores. Para poder llevar a cabo dicho estudio, se implementará un emulador de *cache* de un servidor *proxy* con diferenciación de tipos de documento, en lenguaje Java.

A la hora de llevar a cabo el estudio, se podía haber optado por llevar a cabo una implementación real de una *cache* en un servidor, pero este método, además de ser el que más recursos consumiría, no sería el más adecuado para comparar el comportamiento de las distintas políticas de reemplazo a evaluar. De ahí que se optara por las emulaciones como medio más efectivo para realizar el estudio, y por tanto, fuese necesaria la implementación de un emulador de *cache proxy*.

En el emulador diseñado, se podrá elegir la política de reemplazo a emplear, así como el tamaño de la *cache*, el tipo de documento que se almacenará en la misma, el subtipo o subtipos de dicho tipo, la función de coste a usarse para realizar las métricas (caso de ser aplicable a la política de reemplazo elegida) y el archivo o los archivos a emplear para hacer las emulaciones.

Los tipos de documentos que se distinguirán serán aplicaciones, vídeo, audio, texto e imágenes. De entre dichos tipos se podrá elegir entre distintos subtipos, como se ha indicado anteriormente, siendo las emulaciones realizadas empleando siempre un único tipo de documento.

Una vez desarrollada la aplicación y comprobado su correcto funcionamiento, se procederá a la realización de un conjunto de emulaciones para cada uno de los tipos de

documentos existentes. En dichas emulaciones se irá variando tanto el tamaño de la *cache* como la política de reemplazo empleada, de forma que se puedan cotejar al final los resultados. La *cache* tomará los tamaños del 2, 5, 10, 30 y 50% respecto al tamaño de *cache* infinita asociado a cada tipo de documento concreto (que habrá tenido que ser obtenido previamente).

En primer lugar las simulaciones se realizarán seleccionando que se puedan introducir en *cache* todos los subtipos, una vez elegido un tipo de documento determinado. Tras estas emulaciones, se elegirán los subtipos de datos más frecuentes de entre todos los existentes para cada tipo, según las estadísticas de su aparición en las trazas empleadas y se repetirán todas las emulaciones. Por último se identificarán qué subtipos son los que más aportan a la consecución de una mejor tasa de acierto y tasa de acierto de byte y se repetirán las emulaciones de nuevo.

Los resultados de todas estas emulaciones se emplearán para la confección de una serie de gráficas mediante el uso de MATLAB. Con estas gráficas se harán comparativas del rendimiento proporcionado por las distintas políticas de reemplazo que facilitarán la extracción de conclusiones relativas a cuáles son las mejores opciones, en lo relativo a los parámetros a elegir, que permitan optimizar el rendimiento de la *cache* del servidor *proxy*, en función de la política de reemplazo, el tamaño y la función de coste asociados a un tipo de documento concreto.

La presente memoria se estructura en cinco capítulos y tres apéndices que se describen a continuación. En primer lugar el capítulo de introducción en el que se ha presentado el marco tecnológico en el que se encuadra este proyecto así como los objetivos perseguidos y la forma empleada para su consecución.

En el segundo capítulo, se presentarán los conceptos y conocimientos teóricos básicos para la comprensión del Proyecto Fin de Carrera. Se hará una introducción a lo que es un servidor *proxy*, una *cache* y a las distintas políticas de reemplazo que posteriormente serán evaluadas en este estudio. También se hará una introducción al protocolo HTTP (*HyperText Transfer Protocol*) y los aspectos de éste que están relacionados con la *cache*.

El tercer capítulo constará de una descripción del emulador diseñado para lo que previamente se hará una introducción a la programación orientada a objetos y al lenguaje de programación Java, elementos ambos, en los cuales se ha basado el diseño de la aplicación. En la descripción del emulador se describirá el interfaz gráfico que posee así como su funcionamiento interno y consideraciones tenidas en cuenta. También se describirán las pruebas realizadas para verificar el correcto funcionamiento del emulador y las recomendaciones para el uso del mismo.

En el cuarto capítulo se realizará el estudio comparativo de las distintas políticas de reemplazo, a la vista de los resultados obtenidos de las emulaciones. En él, además, se describirán las métricas empleadas y las características de las trazas.

El capítulo cinco recogerá las conclusiones finales del proyecto así como las posibles líneas futuras de investigación asociadas al mismo.

Por último, se incluirá el Apéndice A con diversos diagramas UML (*Unified Modeling Language*) representativos de la estructura del emulador diseñado. En el Apéndice B, se presenta un ejemplo de la verificación del funcionamiento del emulador que se llevó a cabo. Por último, en el Apéndice C, se incluirán los subtipos de documentos considerados en relación a cada tipo de documento y la codificación que se asoció a cada uno de ellos.

CAPÍTULO 2: Fundamentos de *cache* Web

En este capítulo, se presentarán los conceptos de servidor *proxy* y de *cache*. Tras esto, se hará una introducción a HTTP y sus aspectos relacionados con la *cache* y posteriormente se explicarán las distintas políticas de reemplazo empleadas para hacer la evaluación de la *cache proxy*.

2.1. CONCEPTO DE SERVIDOR *PROXY*

Un servidor *proxy* WWW proporciona acceso a la Web para personas en subredes cerradas que tan sólo pueden acceder a Internet a través de un determinado servidor, o de un cortafuegos (como se muestra en la figura 2.1.), siendo este su uso principal. Incluso usuarios que no dispongan de DNS (*Domain Name Service*) pueden acceder a Internet a través de él siempre que conozcan la dirección IP (*Internet Protocol*) del servidor *proxy*.

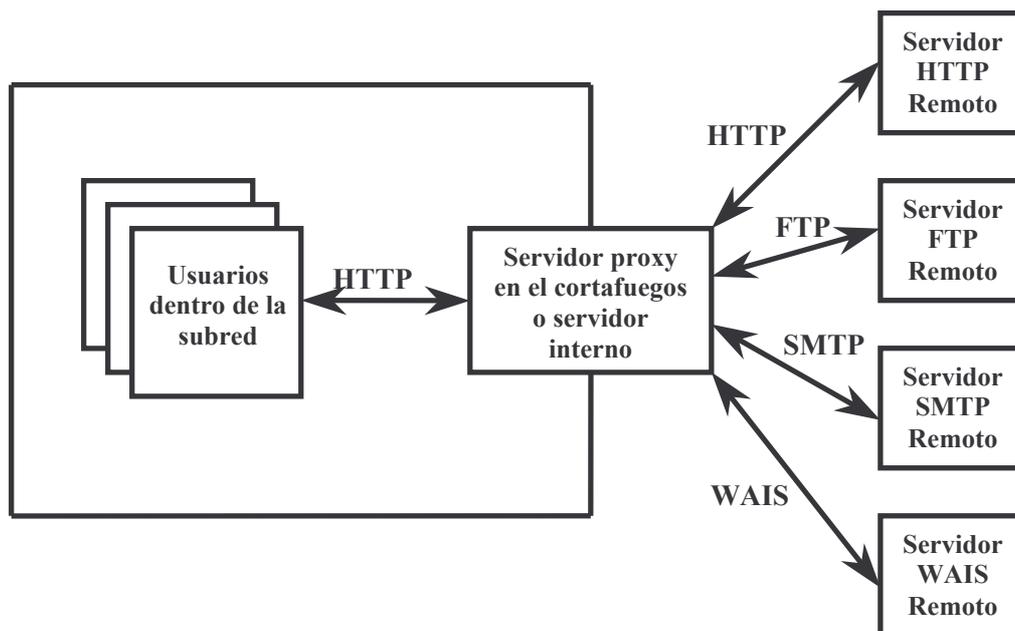


Figura 2.1. Ejemplo de acceso a Internet a través de servidor *proxy*

El servidor *proxy* espera la llegada de una petición procedente de un usuario desde dentro del cortafuegos. Una vez se produce, si el servidor *proxy* no dispone del documento solicitado, o del que dispone ha expirado, la dirige a un servidor remoto fuera del cortafuegos, lee la respuesta, la manda de vuelta al usuario, y la almacena en su memoria local. Actúa, por tanto, como intermediario entre el usuario y otro servidor de información al que se quiera acceder y suele ser usado por todos los usuarios de una subred. Los usuarios no pierden funcionalidad haciendo uso del *proxy* excepto si es necesario hacer algún procesamiento especial.

El servidor *proxy* permite el almacenamiento en *cache*, de forma local, de las páginas consultadas con mayor frecuencia y así, ante una petición de un usuario, si el documento solicitado está presente en *cache* y no ha alcanzado aún su tiempo de expiración (es decir, aún se considera como “fresco”) o bien ha sido validado, podrá ser servido directamente de *cache*. La diferencia entre ambos casos se encuentra en que si no ha expirado, se considera que el documento sigue siendo válido, mientras que si lo ha hecho, pero el servidor ha confirmado que aún no ha sido modificado (comprobando por ejemplo, la cabecera *Last-Modified* al realizar una petición *If-Modified-Since*) también se puede seguir usando el documento y no es necesario volver a traerlo desde el servidor remoto original.

De este modo se obtiene un notable incremento en la velocidad de transferencia de la información con el consiguiente uso eficiente del ancho de banda y un menor tiempo de respuesta, así como permite un ahorro en espacio de disco puesto que una sola copia es almacenada y usada por múltiples usuarios. Incluso puede permitir mostrar páginas Web aún cuando la red exterior no esté disponible, proporcionándolas de su propia *cache*. De este modo, el uso de un servidor *proxy* resulta interesante siempre a cualquier usuario.

El servidor *proxy* puede actuar como filtro de contenidos, como traductor de formato de archivos, adaptador de protocolos (sin pérdida de funcionalidad) o para verificar la seguridad (virus, accesos permitidos, etc.) incrementando la seguridad de la red interna y pudiendo ser tan efectivo como un cortafuegos.

A la hora de hacer transacciones con el servidor *proxy*, el cliente siempre usará HTTP aún cuando el recurso pedido se encuentre en un servidor remoto que use un protocolo distinto, como por ejemplo FTP (*File Transfer Protocol*), como se muestra en la figura 2.2. A la hora de hacer la petición al servidor *proxy*, se especificará la URL (*Uniform Resource Locator*) completa y no sólo el nombre de la ruta u otras claves de búsqueda adicionales como ocurre en HTTP.

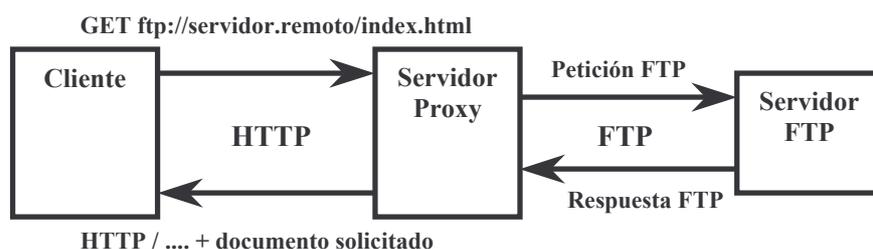


Figura 2.2. Ejemplo de transacción a través de *proxy*

Por tanto, el servidor *proxy* deberá realizar funciones tanto de servidor como de cliente. Actuará como servidor cuando acepte las peticiones HTTP de los usuarios conectados a él pero actuará como cliente cuando curse esas peticiones hacia los servidores remotos para poder obtener de ellos los documentos pedidos por los usuarios.

2.2. CONCEPTO DE *CACHE*

La *cache* es otro aspecto importante y sobre el que ya se ha hecho alguna reseña anteriormente. Se encarga del almacenamiento de los documentos más frecuentemente consultados por los usuarios, como se muestra en la figura 2.3. El uso de una *cache* sólo será beneficioso cuando el coste de almacenamiento de un documento sea menor al que supondría traer dicho documento directamente de un servidor origen.

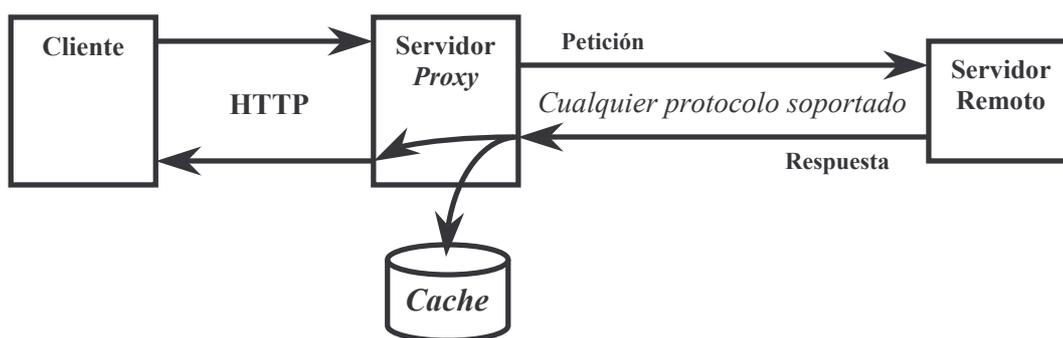


Figura 2.3. Ejemplo de uso de *cache proxy*

El concepto de *cache* es aplicable a casi todos los aspectos de la computación y los sistemas de red. Los procesadores de los computadores tienen *caches* tanto de datos como de instrucciones mientras que los sistemas operativos tienen *caches* para manejadores de disco y archivos de sistema.

El buen funcionamiento de las *caches* radica en el principio de localidad que siguen las referencias. Existen dos tipos de localidad, temporal y espacial. La localidad temporal hace referencia a la popularidad de los documentos (en un período de tiempo existe más probabilidad de que se soliciten ciertos documentos que otros), mientras que la localidad espacial tiene que ver con el hecho de que ciertos documentos tienen la tendencia de ser pedidos conjuntamente con otros, como por ejemplo las imágenes que pueda haber en una determinada página Web. Así, al solicitarse una página Web también existe una elevada probabilidad de que se soliciten las imágenes que contiene la misma.

Las *caches* usan la localidad de referencia para predecir accesos futuros basados en otros previos y en el caso de que la predicción sea correcta se producirá un incremento notable del rendimiento. Cuando un documento solicitado se encuentra en *cache*, se conoce a esto como un acierto en *cache* mientras que si no está se producirá un fallo en *cache*. La

mayoría de las tareas de procesamiento de datos exhiben localidad de referencia y por lo tanto se benefician del almacenamiento en *cache*.

Cualquier sistema que use una *cache* tiene que disponer de mecanismos de mantenimiento de consistencia en *cache*. Este es el procedimiento por el que los documentos almacenados en *cache* se mantienen actualizados respecto de los originales. Los documentos podrán ser frescos u obsoletos. Los frescos pueden ser utilizados inmediatamente pero los obsoletos requieren una validación previa para su uso. A la hora de llevar a cabo dicha validación, los algoritmos para mantener la consistencia pueden ser fuertes o débiles. En los algoritmos débiles, la *cache* a veces devuelve documentos obsoletos. Los fuertes sin embargo obligan siempre a una validación previa de los documentos antes de que puedan ser enviados al usuario. La CPU (*Central Process Unit*) y los archivos de sistema requieren consistencia fuerte mientras que otros sistemas como aquellos situados en encaminadores son efectivos usando consistencia débil. En el siguiente apartado se tratarán más en profundidad todos estos aspectos.

Sin el uso de las *caches*, las páginas Web habrían sido víctimas de su propio éxito ya que debido a su gran crecimiento en popularidad, el número de usuarios que accede a servidores Web populares también ha crecido y con ello el ancho de banda requerido para poder dar cobertura a dicha demanda.

El uso de las *caches* Web se justifica en base a tres beneficios principales como son la reducción en la latencia, en el uso de ancho de banda y en la carga de tráfico soportada por los servidores remotos, donde se encuentran originalmente los documentos a los que se desea acceder.

La latencia, básicamente, está referida a los retardos que sufre la transmisión de la información de un punto a otro. La transmisión de información sobre circuitos eléctricos y ópticos está limitada por la velocidad de la luz. De hecho, los pulsos eléctricos y ópticos viajan aproximadamente a dos tercios de la velocidad de la luz en cables y fibras, lo que genera un retardo que se incrementa con la distancia y que en conexiones transoceánicas es bastante considerable.

Por otro lado, si en las comunicaciones se emplean enlaces que están próximos a su límite de máximo uso, los documentos sufren largos retardos en las colas de los encaminadores y conmutadores. Esto puede ocurrir en un elevado número de puntos a lo largo del camino seguido por la información, lo que puede ocasionar grandes retardos e incluso la pérdida de la información en el caso de que sea descartada de la cola de un encaminador si dicha cola se llena. Con el uso de protocolos fiables como TCP, los paquetes perdidos se pueden retransmitir, pero una pequeña cantidad de paquetes perdidos puede dar lugar a una disminución importante del rendimiento de la comunicación a causa de las retransmisiones que se pueden originar.

Una *cache* Web situada cerca de sus usuarios reduce la latencia para los aciertos en *cache*. Los retardos de transmisión que se producen son mucho menores porque los sistemas se encuentran próximos. Además, los retardos originados por retransmisiones o por la introducción de los paquetes en colas son menos probables pues en la transmisión se ven involucrados menos encaminadores y enlaces.

Por otro lado, un fallo en *cache* no deberá sufrir un retardo mucho mayor del que supondría una transferencia directa entre el usuario y el servidor original. Por lo tanto, los aciertos en *cache* reducen la latencia media de todas las peticiones.

Otro de los beneficios principales del uso de *caches* Web es la reducción en el ancho de banda utilizado, puesto que cada petición que supone un acierto en *cache* permite un ahorro de dicho ancho de banda. Si una conexión de Internet está congestionada, instalando una *cache* se mejora el rendimiento de otras aplicaciones de red, ya que todas compiten por el mismo ancho de banda. Así, con la *cache* Web se consigue reducir el ancho de banda consumido por tráfico HTTP dejándolo libre para otras aplicaciones que también lo requieran.

El tercer beneficio principal del uso de *cache* Web es la reducción en la carga de tráfico soportado por los servidores origen de los documentos. El tiempo de respuesta de un servidor aumenta conforme la tasa de peticiones al mismo es mayor, mientras que, por el contrario, un servidor libre de peticiones es más rápido respondiendo a la llegada de alguna, por lo que, si haciendo uso de *caches* Web se consigue esa disminución de carga de trabajo se está favoreciendo la mayor efectividad en las conexiones con los servidores remotos.

Por todo ello, se buscó la alternativa del uso de *caches* Web para conseguir que los usuarios apreciaran unos menores retardos en sus peticiones de documentos, los gestores de las redes un menor tráfico y los servidores Web una menor tasa de peticiones y todo ello gracias a disponer de copias de los documentos más populares en servidores próximos a los usuarios.

En definitiva, el funcionamiento de la *cache* se basa en que, una vez la petición de un usuario es almacenada en la *cache* del servidor *proxy*, ante una segunda petición de la misma página por parte de otro usuario, el servidor *proxy* podrá servirla directamente de *cache*, siempre que la copia no haya expirado o de haber sido así, haya sido validada, evitando de este modo el acceso remoto, como se ilustra en la figura 2.4.

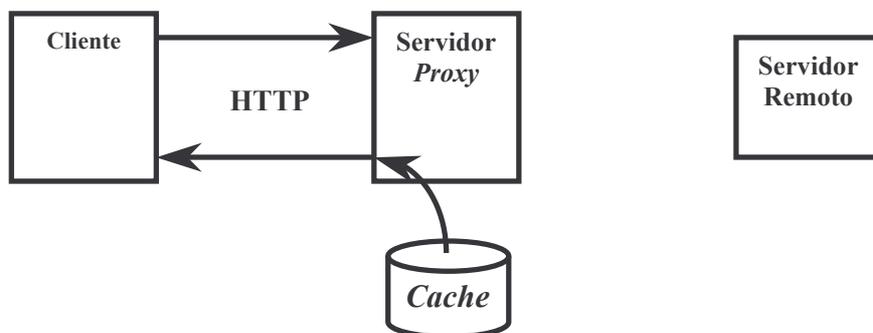


Figura 2.4. Ejemplo de acierto en *cache*

Por tanto la *cache proxy* suele actuar como una *cache* de segundo nivel ya que recibe sólo los fallos en *cache* procedentes de los usuarios Web que ya usan una *cache* a nivel de usuario. De los aciertos en *cache* de usuario tan sólo recibe peticiones para comprobar la frescura del documento existente en la *cache* de usuario y en el caso de comprobarse la validez del mismo, permite un ahorro en ancho de banda para ambas *caches* simultáneamente.

Existen distintos tipos de *caches* Web entre los que podemos distinguir:

- *Cache* de navegador (o de usuario): como su nombre indica está asociada al navegador que usa el usuario. Esta *cache* funciona siguiendo unas reglas bastante simples y se encarga de comprobar que los documentos existentes en la *cache* están actualizados, por lo general, una vez por sesión. Es una *cache* especialmente útil cuando se usa el botón de “atrás” de un navegador o se trata

de acceder a páginas recientemente visitadas. Trata de aprovechar la localidad en las peticiones del usuario para reducir el tráfico en la red y el tiempo de respuesta. Dentro de ellas se distinguen dos tipos:

- Persistentes: mantiene los documentos en memoria ante distintas invocaciones del navegador Web.
 - No persistentes: libera la memoria usada cuando el usuario cierra el navegador.
- *Cache proxy*: trabajan siguiendo el mismo principio que las *cache* de navegador pero a una escala mucho mayor puesto que se encuentran en servidores *proxy* que pueden dar servicio a cientos o miles de usuarios, de ahí que las *cache proxy* sean *caches* compartidas. Permiten ahorro de ancho de banda y latencia, pero tienen el riesgo de ocasionar cuellos de botella.
 - *Cache* de pasarela: al igual que las *caches proxy*, son intermediarias, pero en lugar de ser usadas por los administradores de las redes para ahorrar ancho de banda son típicamente usadas por los propios gestores Web para hacer sus páginas más escalables, fiables y que proporcionen un mejor servicio, para lo cual almacenan en memoria principal los documentos más solicitados para de esta forma, reducir los accesos a disco.

Por otro lado, las dos topologías principales a la hora de organizar las memorias *cache* son:

- Distribuida: el contenido de la *cache* se distribuye entre una serie de servidores *proxy* todos conectados entre sí en una red. El grupo de servidores coopera en la creación de un sistema de almacenamiento robusto y eficiente, con la capacidad de compartir la carga. Este sistema funciona bien incluso si alguno de los servidores falla y el flujo de datos hacia los distintos usuarios no se detiene. Con esta tecnología, la búsqueda de información en la *cache* es bastante rápida.

- Jerárquica: como su nombre indica, sigue una estructura jerárquica, es decir, las peticiones del usuario son primero procesadas en un *proxy* local y si no se encuentra la petición solicitada en éste, se dirige al siguiente servidor *proxy* de acuerdo a la jerarquía que se encuentre establecida y siendo, habitualmente, el flujo de peticiones, en sentido ascendente en la topología. El siguiente servidor al que se dirija la petición puede encontrarse a su mismo nivel de jerarquía o a un nivel superior, según la estructura existente, y en el caso de que exista más de un acierto hay formas de decidir qué servidor se prefiere, tomándose la información requerida de él. En el caso de no producirse acierto en *cache* en ninguna de las *caches* de la jerarquía, la petición se enviaría al servidor original

De los distintos tipos de *caches* Web, la que nos va a ocupar principalmente va a ser la *cache proxy*, de ahí que veamos algunas de sus características principales. La principal característica de la *cache* de un servidor *proxy* es su habilidad para almacenar documentos que puedan ser solicitados para un uso posterior, con lo que se consigue un importante ahorro en tiempo y ancho de banda. Las *caches proxy* además presentan otra amplia gama de características adicionales, muy valoradas por muchas organizaciones y la mayoría de las cuales están asociadas únicamente a su uso conjuntamente con un servidor *proxy* pero que tienen relativamente poco que ver con el propio almacenamiento en *cache*.

Algunas de estas otras características son descritas a continuación:

- Autenticación: un servidor *proxy* puede pedir a los usuarios que se autenticen antes de servirles documentos. Esto es especialmente útil para cortafuegos pues si cada usuario autorizado tiene un nombre de usuario y contraseña, sólo éstos podrán acceder a la Web desde dentro de esa red privada.
- Filtrado de peticiones: los servidores *proxy* que disponen de almacenamiento en *cache*, pueden emplearse para realizar el filtrado de las peticiones de los usuarios. Los servidores *proxy* pueden ser configurados para denegar peticiones de los usuarios a contenidos fuera de los permitidos por la política de cada organización.

- Filtrado de respuestas: de igual modo que con las peticiones, se puede hacer un filtrado de las respuestas recibidas. Ello habitualmente implica el control de los contenidos de los documentos descargados, como por ejemplo para realizar chequeos contra ataques de virus.
- Hacer descargas por adelantado: este proceso consiste en traerse imágenes o enlaces referenciados en un fichero HTML (*HyperText Markup Language*) antes de que sean pedidos, suponiendo que van a ser solicitados por el usuario en breve. Una predicción correcta supone una reducción de la latencia pero si es incorrecta, se malgasta ancho de banda.
- Traducción y *transcoding*: Ambos se refieren a procesos que cambian el contenido de algo sin que con ello exista un cambio significativo de su significado o apariencia. Un ejemplo de traducción puede ser una aplicación que realice la traducción de una página de texto de un idioma a otro conforme es descargada. El segundo proceso supone cambios a bajo nivel en los datos digitales. Un ejemplo de esto podría ser pasar una imagen de formato GIF (*Graphics Interchange Format*) a JPEG (*Joint Photographic Experts Group*) porque en el formato JPEG los documentos resultan más pequeños y con ello se puede llevar a cabo una transferencia de forma más rápida.
- Modelado de tráfico (*Traffic shaping*): Llevar a cabo un control del ancho de banda usado, empleando para ello el servidor *proxy*, pues de esta forma se puede conseguir información adicional que los administradores de red encuentran útil.

A pesar de todos los beneficios derivados del uso de *caches proxy*, existen una serie de aspectos y consecuencias a tener en cuenta a la hora de usar una *cache* Web. En primer lugar puede ser difícil garantizar la consistencia para una *cache* Web. La *cache* podría devolver documentos no actualizados a los usuarios ya que los servidores Web no suelen dar mucha información acerca de la frescura de los documentos y en algunos casos no se da información alguna.

Puesto que la *cache proxy* ha de devolver documentos actualizados a los usuarios, puede que la demanda de validación sea la única forma para conseguirlo y ello implica

unas pérdidas de tiempo relativamente grandes en comparación con otros sistemas. Además, la petición de validación puede no alcanzar al servidor remoto por un fallo en la red o el servidor, con lo que la *cache* no sabrá realmente si el documento almacenado en ella está actualizado o no.

Por otro lado, el uso de *cache* hace más complejo saber qué usuarios han visitado qué páginas y con qué frecuencia lo han hecho. Los aciertos en *cache* no se registran en el servidor origen y los servidores *proxy* tienden a ocultar la identidad de los usuarios, pues todos los usuarios conectados a un servidor *proxy* tendrán la misma dirección IP. De esta forma los suministradores de contenidos no pueden realizar un adecuado análisis de los accesos a sus páginas Web.

El copyright ha supuesto también, durante bastante tiempo, un problema al uso de las *caches*, porque éstas no permitían al autor de un determinado documento controlar la distribución de su trabajo. HTTP permite especificar a los autores cómo han de ser manejados y distribuidos sus documentos por los distintos tipos de *caches* aunque el protocolo no trata el copyright de una forma directa.

Alguna gente predice que la cantidad de contenido Web que será dinámico y personalizado crecerá en el futuro. Las respuestas dinámicas, normalmente no se deben almacenar en *cache* porque no se pueden reutilizar en peticiones futuras. Si la predicción es cierta, el uso de *cache* será menos importante con el paso del tiempo. Otros, por el contrario, creen que los contenidos cada vez son más estáticos y será más fácil diferenciar información estática y dinámica. En este caso el uso de *cache* será cada vez más efectivo.

Por tanto, existe una lucha por el control de los contenidos Web. Usuarios y proveedores de servicio quieren altos porcentajes de acierto en *cache* porque así ahorran tiempo y ancho de banda, mientras que los autores de contenidos quieren menos aciertos de las *cache proxy* para controlar mejor la distribución de sus documentos y llevar a cabo una cuenta más exacta de los accesos a sus páginas. A pesar de estos problemas, las ventajas proporcionadas por el uso de *caches* se presentan como más determinantes, sin ignorar, por supuesto, todas estas problemáticas asociadas.

2.3. HTTP Y LA *CACHE* WEB

Usuarios y servidores utilizan una serie de distintos protocolos de transporte para intercambiar información. Estos protocolos, implementados sobre TCP/IP (*Transfer Control Protocol/Internet Protocol*) constituyen una parte importante del tráfico de Internet hoy en día. HTTP es el más común porque está diseñado específicamente para la red, aunque existen otros como FTP que aún se usan pero de forma minoritaria.

Las transacciones hechas con HTTP usan una estructura de mensaje bien definida, como se presenta en la figura 2.5. Los mensajes, tanto peticiones como respuestas tienen una cabecera y un cuerpo, siendo la cabecera obligatoria y el cuerpo opcional. Las cabeceras son cadenas ASCII (*American Standard Code for Information Interchange*) terminadas por un carácter retorno de carro y otro de nueva línea. Una línea vacía indica el final de la cabecera y el inicio del cuerpo. Los cuerpos son tratados como datos binarios, siendo en las cabeceras donde se encuentra la información y directivas relevantes para el almacenamiento en *cache*.

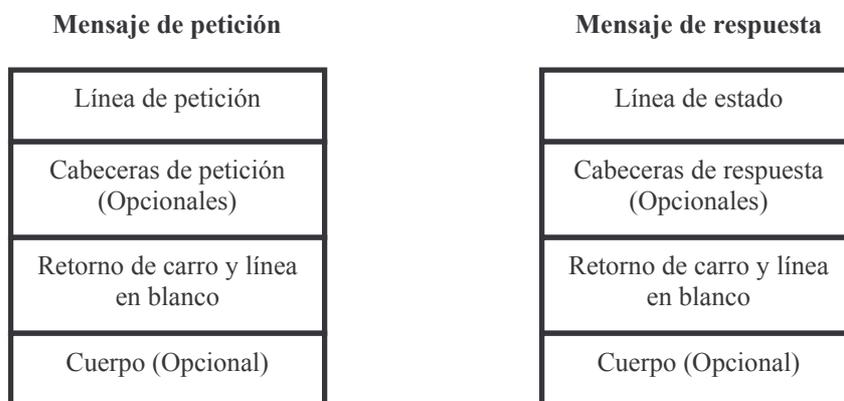


Figura 2.5. Estructura de los mensajes de petición y respuesta HTTP

Una cabecera de petición o de respuesta, consiste en el nombre de la cabecera seguido de dos puntos y después uno o más valores separados por comas. Los nombres de cabecera que están compuestos por más de una palabra se escriben con cada una de las palabras separadas por guiones. Tanto los nombres de cabecera como las palabras reservadas son insensibles a mayúsculas o minúsculas.

HTTP define cuatro categorías de cabeceras: entidad, petición, respuesta y general. Las cabeceras de entidad contienen meta-datos del cuerpo del mensaje, como por ejemplo, la longitud del cuerpo, descrito por la cabecera *Content-length*. Las cabeceras de petición sólo aparecen en peticiones HTTP pues para respuestas no tienen sentido. Algunas de estas cabeceras son *Host* o *If-modified-since*. Cabeceras de respuesta se dan sólo en respuestas, como por ejemplo *Age*. Por último, las cabeceras generales son de doble propósito, pues se pueden encontrar tanto en peticiones como en respuestas, como es el caso de *Cache-control*.

La primera línea de un mensaje HTTP es especial. Para las peticiones se llama línea de petición y contiene el método de petición, un URI (*Universal Resource Identifier*), y el número de versión HTTP (todos ellos separados por espacios en blanco). Un ejemplo de petición podría ser:

```
GET /indice.html HTTP/1.1
Host: www.miservidor.es
Accept: */*
```

Para las respuestas, la primera línea se llama línea de estado e incluye el número de versión HTTP y el código de estado que indica el éxito o fracaso de la petición. La mayoría de los mensajes de petición no poseen cuerpo pero la mayoría de las respuestas sí que lo incluyen. Un ejemplo de respuesta podría ser:

```
HTTP/1.1 200 OK
Date: Wed, 2 Sep 2002 22:00:00 GMT
Last-Modified: Wed, 1 Sep 2002 20:12:12 GMT
Server: Apache/1.2.5
Content-Length: 15
Content-Type: text/html
```

En la tabla 2.1 aparecen los métodos de petición definidos por la RFC (*Request For Comments*) 2616 [RFC 2616'99]. Existen también otras RFC como la 2518 [RFC 2518'99] que definen otros métodos de petición adicionales. Si un servidor *proxy* recibe una petición

que incluye un método desconocido o no soportado por dicho servidor, debe responder con el mensaje 405 (*Method Not Allowed*).

Tabla 2.1. Métodos de petición HTTP definidos por la RFC 2616

Método	Descripción
GET	Petición de la información identificada con el URI solicitado
HEAD	Idéntico a GET, excepto que la respuesta no incluye cuerpo del mensaje
POST	Petición al servidor para que procese la información del cuerpo del mensaje
PUT	Petición para que los datos incluidos en el cuerpo se guarden en el URI indicado
TRACE	Realiza un eco de la petición, de vuelta al cliente. Útil para descubrir y testear servidores <i>proxy</i> entre el usuario y el servidor
DELETE	Petición para eliminar el URI indicado del servidor origen
OPTIONS	Petición de información sobre las capacidades del servidor o la posibilidad de soportar características opcionales
CONNECT	Usado para crear túneles a través de servidores <i>proxy</i> para ciertos protocolos

Para el estudio del funcionamiento de la *cache*, sólo los métodos de petición GET, HEAD y POST serán interesantes, ya que son los únicos que pueden ser útiles para la *cache*.

A continuación se presentarán distintos aspectos del protocolo HTTP y su relación con el funcionamiento de una *cache proxy*. Entre ellos se presentará cómo son las peticiones HTTP y las diferencias que existen cuando éstas se hacen a servidores *proxy* y cómo son gestionadas las peticiones que no son HTTP. También se verá cómo las *caches* deciden si se va a almacenar una respuesta o no y como afecta la frescura de un documento y su validación a la hora de decidir si se va a producir un acierto o fallo en *cache*.

2.3.1. PETICIONES HTTP

Los usuarios siempre utilizan HTTP para comunicarse con un servidor *proxy* que use una *cache*, siendo esto cierto aún cuando las peticiones se realicen a una URL de otro tipo, como por ejemplo FTP. Sin embargo, el usuario realizará una petición algo distinta cuando

sabe que se comunica con un servidor *proxy* en lugar de comunicarse con un servidor origen remoto.

2.3.1.1. Peticiones al servidor origen

Si suponemos que se quiere hacer una petición a un servidor origen con URL <http://www.miservidor.es/indice.html>. Si el usuario no ha configurado su conexión para usar un servidor *proxy*, se conecta directamente con el servidor remoto (www.miservidor.es) y envía la petición:

```
GET /indice.html HTTP/1.1
Host: www.miservidor.es
Accept: */*
Connection: Keep-alive
```

La petición en realidad incluye muchas más cabeceras que las mostradas. La URL se divide en dos partes. La línea de petición incluye sólo el componente de la ruta de la URL mientras que el nombre del servidor origen se incluye en la cabecera *Host*.

2.3.1.2. Peticiones a un servidor *proxy*

Las peticiones hechas a un servidor *proxy* son ligeramente distintas. En la línea de petición se incluye la URI completa como se muestra a continuación:

```
GET http://www.miservidor.es/indice.html HTTP/1.1
Host: www.miservidor.es
Accept: */*
Proxy-connection: Keep-alive
```

El nombre del servidor origen se encontrará por tanto en dos lugares, en la URI completa incluida en la línea de petición y en la cabecera *Host*. Esto puede parecer algo redundante, pero cuando las técnicas de uso de servidores *proxy* se inventaron, la cabecera *Host* no existía. Puede ocurrir que los dos nombres de *Host* empleados no sean el mismo, en dicho caso, el de la URL tiene preferencia.

HTTP permite que las peticiones y respuestas puedan pasar a través de varios servidores *proxy* durante su transmisión entre el usuario y servidor origen. Algunas cabeceras HTTP se definen extremo a extremo y otras, salto a salto. Las cabeceras extremo a extremo, transportan información para los sistemas finales (usuario y servidor origen) y por lo general, no deben ser modificadas por los servidores *proxy*. Un ejemplo de dicho tipo de cabeceras es la cabecera *Cookie*. Por el contrario, la información de las cabeceras salto a salto, está pensada para sistemas intermedios y debe, a menudo, ser modificada o eliminada antes de remitirse hacia el siguiente destino. Las cabeceras *Proxy-connection* y *Proxy-authorization* son cabeceras salto a salto. En el caso de la primera, el cliente usa la cabecera *Proxy-connection* para pedir al servidor *proxy* que se establezca una conexión TCP persistente que pueda ser reutilizada en el futuro. La cabecera *Proxy-authorization* contiene credenciales para el acceso al servidor *proxy*, no al servidor origen.

2.3.1.3. Peticiones no-HTTP a un servidor *proxy*

La mayoría de los agentes de usuario soportan otros protocolos de transferencia distintos de HTTP, pero la mayoría de los servidores *proxy* no emulan a servidores de esos otros protocolos, de ahí, que no se pueda establecer una conexión FTP con un servidor *proxy*.

Por lo tanto, los URI que no son HTTP son gestionados por los servidores *proxy* como peticiones HTTP. Por ejemplo, una petición FTP de un agente de usuario presenta un aspecto como:

```
GET ftp://ftp.miservidor.org/pub/ HTTP/1.1
Host: ftp.miservidor.org
Accept: */*
```

Puesto que se trata de una petición HTTP, el servidor *proxy* genera una respuesta HTTP. El servidor *proxy* se encargará de convertir de FTP en un lado a HTTP en el otro. Actuará como un cliente FTP cuando se comunica con un servidor FTP y respecto al usuario actuará como un servidor HTTP pues la respuesta enviada al agente del usuario será HTTP.

Las peticiones que no son HTTP generan dificultades para el almacenamiento en *cache* de los servidores *proxy* en distintos aspectos. Los servicios FTP se basan en el uso de listados de directorios. Cuando un usuario Web se conecta directamente, por ejemplo, a un servidor FTP analiza los listados y genera la página HTML. Los servidores HTTP funcionan de forma distinta ya que es el propio servidor el que genera la página y no el usuario. Al ser gestionadas las peticiones FTP a través del servidor *proxy*, se convierte en responsabilidad del servidor *proxy* la generación de las páginas adecuadas para los listados de directorios.

Otro aspecto complejo es que HTTP posee ciertos atributos y características que protocolos más antiguos no proporcionan. En particular, puede ser difícil obtener valores de las cabeceras *Last-modified* y *Content-length*. Por supuesto, los servidores FTP no entienden nada sobre expiración o validación de documentos. Normalmente, será el servidor *proxy* el que elija un tiempo de frescura para los documentos no HTTP y, puesto que tiene poca información para realizar dicha elección, se producirán más respuestas con indicación de que un documento guardado en *cache* es obsoleto que para fuentes HTTP.

Por último, otro problema importante es determinar el tipo de contenido de una respuesta. Los servidores *proxy* y los servidores origen, suelen usar la extensión del documento para determinar el tipo de éste. Así, una URL que acaba en *.jpg* tendrá un *Content-type* del tipo *image/jpeg*. Para documentos FTP, el servidor *proxy* deberá adivinar el *Content-type* y es relativamente común que aparezcan extensiones desconocidas. La elección del *Content-type* que haga el servidor *proxy* afecta a cómo manejará la respuesta el navegador. Para documentos de tipo texto, el navegador mostrará el contenido en una ventana. Para otros como *application/octet-stream*, muestra una ventana de “Guardar como”. Con el uso de nuevas extensiones de documentos y tipos de contenidos los servidores *proxy* que usan *caches* deben ser actualizados para poder llevar a cabo el reconocimiento de los mismos.

2.3.2. DOCUMENTOS ALMACENABLES EN *CACHE*

El objetivo primordial de una *cache* es almacenar algunas de las respuestas que recibe del servidor origen. Una respuesta se considera apropiada para ser almacenada en *cache* si

se puede usar para responder peticiones futuras. En flujos de peticiones normales, alrededor del 75% de las respuestas se pueden guardar en *cache*.

Una *cache* decide si un documento es adecuado para su almacenamiento en ella comprobando distintos componentes de la petición y la respuesta. En particular se examina el código de estado de la respuesta, el método de la petición, las directivas *Cache-control* de la respuesta, el validador de la respuesta y la autenticación de la petición.

Estos factores interactúan de manera compleja pues algunos métodos de petición no se pueden guardar en *cache* a menos que ello sea permitido por la directiva *Cache-control*. Algunos códigos de estado se almacenan por defecto, pero la autenticación y la directiva *Cache-control* tienen precedencia.

Aún cuando un documento sea adecuado para almacenarse en *cache*, ésta puede elegir que no sea guardado. Algunos documentos tienen más valor que otros, como por ejemplo, un documento que es solicitado muy frecuentemente tiene más valor que otro que sólo es solicitado una vez. Muchas respuestas dinámicas pertenecen a estos últimos. Si la *cache* identifica los documentos que no tienen valor, puede ahorrar recursos e incrementar el rendimiento no guardándolos en ella.

2.3.2.1. Códigos de estado

Uno de los factores más importantes a la hora de determinar si un documento ha de ser guardado en *cache* o no es el código de respuesta o estado, del servidor HTTP. El código de estado, de tres dígitos, indica si la petición tuvo éxito o si por el contrario, ocurrió algún tipo de error. Los códigos de estado se dividen en los 5 grupos que se presentan en la tabla 2.2.

Tabla 2.2. Grupos en los que se dividen los códigos de estado de las respuestas HTTP

Código	Descripción del Grupo
1xx	Un estado intermedio, de información. Indica que la transacción está siendo procesada
2xx	La petición fue recibida y procesada con éxito
3xx	El servidor está redireccionando al usuario a una ubicación diferente
4xx	Existe un error o problema con la petición del usuario, como por ejemplo que se requiera autenticación o que el documento pedido no exista
5xx	Se produjo un error en el servidor para una petición válida

El código de estado más común es 200 (OK) que significa que la petición fue procesada por completo con éxito. Las respuestas con los códigos que aparecen en la tabla 2.3 son guardados en *cache* por defecto. Sin embargo, existen otros aspectos de las peticiones o respuestas que pueden hacer que las respuestas no sean almacenables en *cache*. El código de estado sólo, no es suficiente para hacer que una respuesta, y por tanto, un documento, sea guardada en *cache*.

Tabla 2.3. Códigos de respuesta almacenables en *cache*

Código	Descripción	Explicación
200	OK	La petición fue procesada satisfactoriamente
203	Non-Authoritative Information	Similar a 200 pero se usa cuando el usuario cree que las cabeceras son distintas de las que el servidor origen proporcionaría
206	Partial Content	Similar a la respuesta 200 pero es la respuesta a una petición de un subconjunto de un documento (petición de rango)
300	Multiple Choices	La respuesta incluye una lista de las elecciones apropiadas de las cuales el usuario hará una selección
301	Moved Permanently	El documento solicitado ha sido movido a una nueva ubicación siendo dada la nueva URL en la cabecera de respuesta
410	Gone	El documento solicitado ha sido borrado intencionada y permanentemente del servidor origen.

Todos los demás códigos de respuesta no se guardan en *cache* por defecto aunque se puede explícitamente permitir que se guarden. Por ejemplo, si una respuesta con el código 302 (*Moved Temporarily*) tiene además una cabecera *Expires*, también se puede guardar en *cache*. Realmente, se consigue poca mejora con el almacenamiento en *cache* de las respuestas que por defecto no lo son, pues dichos códigos de estado ocurren con poca frecuencia y es aún más infrecuente que las respuestas incluyan las cabeceras *Expires* o *Cache-control* para que dichas respuestas se puedan guardar en *cache*. Por tanto, es más simple y seguro para una *cache* no almacenar nunca respuestas de este tipo.

2.3.2.2. Métodos de petición

Otro factor significativo a la hora de determinar si un documento se ha de almacenar en *cache* o no, es el método de petición. Nos centraremos principalmente en 3 métodos, GET,

HEAD y POST pues otros como PUT, DELETE, OPTIONS o TRACE nunca darán lugar al almacenamiento en *cache* de un documento.

GET es el método de petición más popular y las respuestas a las peticiones hechas con GET son guardadas en *cache* por defecto. Las respuestas a las peticiones HEAD son tratadas de forma especial, pues no contienen cuerpo, de ahí que realmente no haya nada que guardar en *cache*. Sin embargo, se pueden usar las cabeceras de las respuestas para actualizar las cabeceras de respuestas previamente almacenadas en *cache*. Así, una respuesta a HEAD puede contener un nuevo tiempo de expiración o indicar que un determinado documento ha cambiado, por lo que la copia en *cache* ya no será válida. La respuesta a una petición POST sólo se introducirá en *cache* si incluye un tiempo de expiración o una de las directivas de *Cache-control* que deje sin efecto el funcionamiento por defecto.

Las respuestas para el resto de métodos de petición nunca se guardarán en *cache*. Además las *caches* no deben reutilizar las respuestas de métodos desconocidos.

2.3.2.3. Expiración y Validación

HTTP proporciona dos maneras a las *caches* para mantener la consistencia con los servidores origen: los tiempos de expiración y los validadores. Ambos aseguran que los usuarios siempre reciban información actualizada. Idealmente, una respuesta almacenable en *cache* incluye una o ambas informaciones, aunque en realidad un pequeño aunque significativo porcentaje de respuestas no tiene ninguna de las dos.

La expiración y la validación afectan a si un documento se guardará en *cache* o no, de dos formas importantes. En primer lugar, la RFC 2616 dice que las respuestas que no contiene ni un tiempo de expiración ni un validador de *cache*, no deben ser guardadas en *cache* pues sin esta información la *cache* no puede saber nunca si una copia en *cache* sigue siendo válida o no. No obstante, hay que notar que esto es sólo una recomendación, no una norma de obligado cumplimiento.

En segundo lugar, un tiempo de expiración convierte normalmente respuestas que no se guardarían en *cache* en otras que sí lo serán. Así, respuestas a peticiones POST y mensajes de estado 302 se convierten en documentos a guardar en *cache* cuando el servidor de origen suministra el tiempo de expiración.

Es importante entender la diferencia entre expiración y que un documento sea conveniente almacenarlo en *cache*. En el segundo caso, la respuesta puede ser almacenada en la *cache* del servidor *proxy*. Para el primer caso, respuestas que hayan expirado pueden ser también guardadas en *cache*, pero tienen que ser validadas antes de ser usadas de nuevo para responder a la petición de un usuario pues, que un documento expire no quiere decir que dicho documento haya cambiado respecto del que se guarde en *cache*, pero tendrá que comprobarse si lo ha hecho o no.

De hecho, algunas páginas Web envían respuestas pre-expiradas. Esto quiere decir que la *cache* debe validar el documento la siguiente vez que alguien lo pida. Esto es útil para los servidores origen que quieren tener un control exhaustivo sobre los accesos a su página Web pero que también quieren que sus documentos puedan ser guardados en *cache*. La forma adecuada de hacer que un documento pre-expire sería establecer la cabecera *Expires* igual a la cabecera *Date*. Otro método alternativo es enviar una fecha no válida o usar el valor "0" en la cabecera *Expires*.

Como conclusión podríamos decir que, para ser almacenable en *cache*, un documento ha de tener un tiempo de expiración, un validador o ambos. Las respuestas que no poseen ninguno de los dos no deben ser guardadas en *cache*, aunque HTTP permite que sean guardadas de todos modos, sin que con esto se viole el protocolo.

2.3.2.4. La cabecera *Cache-control*

La cabecera *Cache-control* se utiliza para indicar a las *caches* cómo tienen que manejar las peticiones y las respuestas. El valor de dicha cabecera puede ser una o más directivas. Aunque esta cabecera puede aparecer tanto en peticiones como en respuestas, nos centraremos sólo en las directivas que aparecen en las respuestas.

Las directivas de *Cache-control* dejan sin efecto el funcionamiento definido por defecto para la mayoría de los códigos de estado y de los métodos de petición a la hora de determinar si un documento ha de guardarse en *cache* o no. Así, con algunas de las directivas de *Cache-control* una respuesta a un GET, que normalmente se guarda en *cache* puede determinarse que no se haga y por el contrario, que una respuesta POST sí sea almacenada en *cache*.

Las directivas *Cache-control*, que pueden aparecer en una respuesta HTTP y afectar a si el documento es guardado en *cache* o no, se muestran a continuación:

- *no-cache*: según indica la RFC 2616 esta directiva permite que las respuestas que la contienen pueden ser guardadas, pero no deben ser usadas de nuevo sin haber sido validadas previamente. De este modo, *no-cache* equivale a la directiva *must-revalidate* con *max-age=0*. Esta directiva también tiene un uso secundario, pues cuando especifica dos o más nombres de cabeceras, dichas cabeceras no deben ser enviadas a un usuario sin haber sido previamente validadas. Esto permite a los servidores origen evitar compartir ciertas cabeceras aunque sí permitir que el contenido de las respuestas, que suele ser la parte más importante y de mayor tamaño de la respuesta, sean guardados en *cache* y reutilizados.
- *private*: esta directiva da permiso a la *cache* del agente de usuario para almacenar respuestas aunque no así a las *caches* de servidores *proxy* compartidos. Esta directiva es útil si la respuesta contiene contenido adaptado para una única persona. El servidor origen podría usar esto para controlar el acceso individual y al mismo tiempo permitir que las respuestas puedan ser guardadas en *cache*. Para *caches* compartidas, la directiva *private*, hace que respuestas normalmente aptas para ser guardadas en *cache* pasen a no serlo. Sin embargo, para una *cache* no compartida, una respuesta que normalmente no sería guardada en *cache* se convierte en una respuesta a guardar por el uso de esta directiva.
- *public*: La directiva *public* hace que respuestas que normalmente no se guardan en *cache* pasen a ser guardadas en la misma tanto para *caches* compartidas como no compartidas. Esta directiva tiene mayor precedencia incluso que las credenciales de autorización. Así, si la petición incluye una cabecera de *Authorization* y las cabeceras de respuesta contienen *Cache-control: public*, la respuesta es almacenable en *cache*.
- *max-age*: esta directiva es una forma alternativa de especificar un tiempo de expiración. Se encuentra relacionada con el hecho de que respuestas que

normalmente no se guardarían en *cache*, pueden guardarse si incluyen un tiempo de expiración. La RFC 2616 recomienda que respuestas que no tengan ni un validador ni un tiempo de expiración no sean guardadas en *cache*. Cuando una respuesta contiene la directiva *max-age*, la directiva *public* está implícita igualmente.

- *s-maxage*: la directiva *s-maxage* es muy similar a la directiva *max-age* con la excepción de que sólo se aplica a *caches* compartidas. La directiva *s-maxage* permite que respuestas normalmente no almacenables en *cache*, pasen a poderse guardar en *cache*. Si tanto la directiva *s-maxage* como *max-age* están presentes en una respuesta, una *cache* compartida usa el valor *s-maxage* e ignora los demás valores de expiración. A diferencia de *max-age*, *s-maxage* no implica la directiva *public*, sin embargo, sí que implica la directiva *proxy-revalidate*.
- *must-revalidate*: esta directiva permite también a las *caches* almacenar una respuesta que normalmente no se guardaría. *must-revalidate* está relacionada con la validación y será tratada en el siguiente apartado.
- *proxy-revalidate*: esta directiva es similar a la directiva *must-revalidate*, con la excepción de que se aplica sólo en *caches* compartidas. También permite a las *caches* guardar respuestas que normalmente no son guardadas en *cache*.
- *no-store*: la directiva *no-store* da lugar a que cualquier respuesta no sea guardada en *cache*. Puede estar también presente en una petición en cuyo caso la correspondiente respuesta no se guardará en *cache*. En la sección 14.9.2 de la RFC 2616 se describe la directiva *no-store* diciendo que peticiones y respuestas que incluyan esta directiva no deben nunca ser escritas en dispositivos de almacenamiento no volátiles, incluso temporalmente. Es una forma para que distribuidores de contenidos muy celosos de los mismos disminuyan la probabilidad de que información especialmente sensible sea descubierta o hecha pública. Es una directiva muy importante, ya que es la única (a excepción de *private*) que evita que una respuesta sea almacenada en *cache*. Por ello se utiliza para muchos tipos de respuestas y no sólo para aquellas que contienen datos especialmente sensibles.

A modo de resumen, las directivas *public*, *max-age*, *s-maxage*, *must-revalidate* y *proxy-revalidate* hacen que respuestas que habitualmente no se guardan en *cache* pasen a guardarse. Las primitivas *no-store* y *private* hacen que respuestas que normalmente se guardan en *cache* pasen a no ser guardadas en una *cache proxy*. Al mismo tiempo *private* hace que una respuesta no almacenable en *cache* si se guarde para una *cache* de agente de usuario.

2.3.2.5. Autenticación

Las peticiones que requieren autenticación no suelen guardarse en *cache*. Sólo el servidor origen puede determinar quién está autorizado para acceder a sus documentos.

Cuando se trata de acceder a un documento protegido, el servidor devuelve un mensaje con el código de estado 401 (*Unauthorized*). Tras recibir esta respuesta, el agente de usuario solicita al usuario sus credenciales de autorización, que normalmente son el nombre de usuario y la clave. El agente de usuario entonces enviará la petición incluyendo esta vez la cabecera de *Authorization*. Cuando un servidor *proxy* con *cache* encuentra dicha cabecera en una petición sabe que la respuesta correspondiente no se guardará en *cache* a menos que el servidor origen explícitamente lo permita.

La sección 14.8 de la RFC 2616 trata de las condiciones bajo las cuales las *caches* compartidas pueden almacenar y reutilizar dicho tipo de respuestas. Sólo se podrán guardar en *cache*, si una de las siguientes cabeceras *Cache-control*, está presente: *s-maxage*, *must-revalidate* o *public*.

La directiva *public* sola, permite el almacenamiento y reutilización de cualquier petición posterior, sujeta al tiempo de expiración. La directiva *s-maxage* permite que la respuesta se reutilice hasta que se alcance el tiempo de expiración. Después de esto, la *cache proxy* debe revalidar el documento con el servidor origen. La directiva *must-revalidate* por su parte, indica a la *cache* que ha de revalidar la respuesta para cada petición realizada de dicho documento.

La RFC 2616 no dice cómo las *caches* de agente de usuario (no compartidas) han de manejar las respuestas autenticadas. Se puede asumir que los documentos pueden ser guardados en *cache* y reutilizados mientras la *cache* no sea compartida con otros usuarios.

Sin embargo, las *caches* no compartidas sí que pueden ser compartidas por distintos usuarios. Por ejemplo, las *caches* situadas en terminales de laboratorios de universidades que pueden ubicarse en equipos compartidos por distintos alumnos.

No es muy habitual que las respuestas autenticadas incluyan los controles de *cache public* o *s-maxage* aunque las directivas *must-revalidate* y *proxy-revalidate* sí que son bastante habituales. Con ellas se proporciona un mecanismo que permite a las respuestas ser almacenadas en *cache*, pero también dar al servidor origen un completo control sobre quién puede o no acceder a la información protegida.

Existe otra cuestión relacionada con la autenticación, aunque no con el almacenamiento en *cache*. Los servidores origen a veces utilizan autenticación basada en la dirección IP en lugar de claves. Es decir, la dirección IP del usuario determina si el acceso al documento pedido es permitido o no. Según esto, es posible para un servidor *proxy* proporcionar acceso a cualquiera que lo desee a dicha información pues mientras la dirección IP del *proxy* se encuentre dentro de las permitidas tendrá acceso y cualquiera que lo requiera podrá acceder a la información restringida a través del servidor *proxy*, pues todos los usuarios conectados a dicho servidor *proxy* hacen uso de la misma dirección IP.

2.3.2.6. Cookies.

Con el uso de las *cookies*, el servidor origen puede mantener información de sesión para usuarios individuales entre las distintas peticiones. Una respuesta puede incluir una cabecera *Set-cookie* que contiene la *cookie* que se presenta como una cadena de caracteres de aspecto aleatorio. Cuando el agente de usuario recibe la *cookie*, se supone que en futuras peticiones que realice a ese servidor, la usará, por lo que la utilizará como un identificador de sesión.

Las *cookies* se suelen usar típicamente para representar o referenciar información privada y no deben ser compartidas entre distintos usuarios. Sin embargo, en la mayoría de los casos es la *cookie* en sí es la única cosa realmente privada. El documento en el cuerpo del mensaje de respuesta puede ser público y guardable en *cache*. En lugar de hacer que toda la respuesta no se pueda guardar en *cache*, HTTP tiene una forma de hacer que sólo la

cookie no sea introducida en *cache*. Para ello se usa la directiva *no-cache* de *Cache-control* como por ejemplo:

Cache-control: no-cache="Set-cookie"

Los servidores origen pueden usar la directiva *no-cache* para evitar que otras cabeceras también se guarden en *cache*.

2.3.2.7. Contenido dinámico

La RFC 2616 no indica directamente si el contenido dinámico se guarda o no en *cache* pues, en términos del protocolo, sólo las cabeceras de peticiones y respuestas determinan el almacenamiento o no en *cache*, no importando cómo es generado dicho contenido. Las respuestas dinámicas, por tanto, se podrían guardar en *cache* siguiendo las reglas ya descritas. Los servidores origen, sin embargo, suelen marcar el contenido dinámico como no guardable en *cache*, por tanto no es algo por lo que nos tengamos que preocupar.

Los servidores de contenido se aseguran de que sus servidores marcan las respuestas dinámicas como no guardables en *cache* usando las cabeceras ya descritas. Sin embargo los administradores de *cache*, en lugar de confiar en los servidores origen, suelen optar por no guardar en *cache* contenido dinámico alguno.

La escasa popularidad es otro motivo para no almacenar en *cache* respuestas dinámicas. Las *caches* proporcionan beneficios cuando el flujo de peticiones contiene peticiones que se repiten habitualmente. Guardar objetos que sólo se solicitan una o dos veces hace que se consuman recursos que podrían ser usados más eficientemente para guardar otros documentos de más valía.

A la hora de identificar los contenidos dinámicos, las cabeceras de respuesta HTTP proporcionan una buena indicación. De forma específica, la cabecera *Last-modified* especifica cuándo fue actualizado un recurso por última vez. Si dicho valor es próximo a la fecha actual es probable que el contenido sea dinámico. Desafortunadamente, en torno al 35% de las respuestas no suelen incluir dicha cabecera.

Otra técnica útil es buscar cadenas específicas en las URL, como las siguientes:

- */cgi-bin/* o *.cgi*: la presencia de */cgi-bin/* o *.cgi* en una URL de la ruta, indica que la respuesta fue generada por un *script* CGI (*Common Gateway Interface*).
- */servlet/*: Cuando la ruta URL incluye */servlet/*, la respuesta fue muy probablemente generada por la ejecución de un *servlet* Java en el servidor origen.
- *.asp*: las páginas ASP (*Active Server Pages*) tienen normalmente la extensión *.asp* y presentan contenido dinámico.
- *.shtml*: la extensión *.shtml* indica una página HTML que contiene unas etiquetas especiales que serán interpretadas por el servidor origen y posteriormente sustituidas por cierto contenido generado dinámicamente. Los *servlets* de Java pueden ser también usados de esta forma dentro de documentos *.shtml*.
- Parámetros de petición (*Query terms*): Otra indicación de contenido dinámico es la presencia de “?” (signo de interrogación) en una URL. Este carácter delimita los parámetros opcionales de una petición o de una parte de la URL.

2.3.3. GESTIÓN DE ACIERTOS Y FALLOS EN *CACHE* Y SU RELACIÓN CON LA FRESCURA Y VALIDACIÓN DE LOS DOCUMENTOS

Cuando una *cache* recibe una petición, comprueba si la respuesta a dicha petición se encuentra ya almacenada en *cache*. Si no es así, se dice que se ha producido un fallo en *cache* y la petición se reenvía al servidor origen. Los fallos en *cache* ocurren para documentos que nunca antes han sido pedidos, para aquellos que no son almacenables en *cache* o para los que han sido borrados para hacer sitio a otros nuevos.

Si la respuesta se encuentra en *cache* es posible que tengamos un acierto en *cache*. Sin embargo, la *cache* debe decidir si la respuesta es fresca u obsoleta. Una respuesta almacenada en *cache* es fresca si su tiempo de expiración no ha sido alcanzado aún, siendo

en otro caso, obsoleta. Las respuestas frescas son siempre preferibles pues se pueden proporcionar a los usuarios directamente, no sufriendo la latencia ni el consumo del ancho de banda requerido para su obtención del servidor origen, como ocurre con las obsoletas, que han de ser validadas.

A la hora de solicitar la validación de un documento, se pueden utilizar validadores fuertes o débiles. Los validadores fuertes llevan a cabo una comprobación exacta, byte a byte, mientras que los débiles, se utilizan para comprobar la equivalencia semántica. Así, si se produce un cambio simple en un documento, como puede ser la corrección de una falta ortográfica, dicha corrección no afecta el significado del documento pero cambia los bits del documento. Ante esta situación, si se realiza una petición condicional con un validador fuerte, se devolvería el nuevo contenido, considerándolo como un nuevo documento, mientras que con un validador débil, indicaría que el documento no ha sido modificado.

El propósito de la petición de validación es preguntar al servidor origen si la respuesta existente en *cache* sigue siendo válida o por el contrario el documento ha cambiado y la copia en *cache* ha pasado a ser obsoleta, pues si el documento ha cambiado, no se deberá suministrar al usuario una copia obsoleta. HTTP llama a estas peticiones, peticiones condicionales pues ellas, en el caso de verificarse que el documento en *cache* es obsoleto, dan lugar a que en la respuesta se envíe el documento fresco. La respuesta a una petición condicional es un mensaje *Not Modified* (caso de no haber cambiado el documento) pequeño o bien una nueva respuesta completa con el documento actualizado. El mensaje *Not Modified*, también conocido como acierto validado, es preferible porque implica que el cliente puede recibir la respuesta directamente de *cache*, con el consiguiente ahorro de ancho de banda y tiempo. Un fallo validado, donde el servidor origen envía una versión actualizada de la respuesta, es equivalente a un fallo en *cache* normal.

Para que la *cache* sepa si un documento es fresco o no, los servidores origen disponen de dos formas para indicar dicha frescura en una respuesta, la cabecera *Expires* y la directiva de control de *cache max-age*. El uso de la cabecera *Expires* implica emplear tiempos absolutos, lo que puede dar lugar a problemas por ejemplo debidos a pequeñas variaciones en el formato o que la noción de tiempo sea distinta en distintos sistemas. Por estos motivos, existen distintas cabeceras y directivas que usan tiempos relativos en lugar de absolutos. Así, *max-age* especifica el número de segundos que la respuesta será considerada fresca tras su generación.

Aún cuando HTTP proporciona estos medios, no se suelen utilizar muy a menudo, por lo que cuando no existe dicha información, las *caches* pueden utilizar libremente reglas locales para estimar el tiempo de vida de un documento y según dicho tiempo, seguir considerándolo como fresco o no. Una política conservadora sería validar siempre las respuestas que no tengan tiempo de expiración. Por otra parte, una política más liberal sería que los documentos se devolvieran a los usuarios como aciertos sin realizar validación alguna.

HTTP proporciona dos mecanismos principales de validación. El primero consiste en el uso de las marcas temporales *Last-modified*. La mayor parte de las respuestas HTTP incluyen la cabecera *Last-modified* que indica cuándo un documento fue modificado por última vez en el servidor origen. La fecha se da con una resolución de un segundo y relativa a la hora en el meridiano de Greenwich, también conocido como GMT (*Greenwich Mean Time*).

Cuando una *cache* quiere validar un documento, dicha marca de tiempo es enviada en la cabecera *If-modified-since* de la petición GET condicional. Si ante dicha petición el servidor origen *responde* con 304 (*Not Modified*), el documento en *cache* sigue siendo válido. En dicho caso, la *cache* ha de actualizar la respuesta almacenada para que refleje las cabeceras HTTP recibidas como *Date* o *Expires*. Si la respuesta no es 304, la *cache* trata la respuesta del servidor origen como nuevo contenido, reemplazando el documento en *cache* y enviando el nuevo documento al usuario.

Este mecanismo tiene como inconvenientes que la marca de tiempo debe ser actualizada aunque no haya cambios en el contenido del documento. El manejo de las marcas de tiempo puede ser complejo cuando se trabaja con sistemas que tienen nociones distintas del tiempo actual y además los valores de *If-modified-since* no pueden ser usados para documentos que deben ser actualizados más de una vez por segundo.

Por otro lado presenta como ventajas que las marcas temporales se pueden almacenar internamente con una cantidad pequeña de memoria y se puede emplear para más cosas aparte de la validación, como por ejemplo, para estimar tiempos de expiración.

El segundo mecanismo de validación son las etiquetas de entidad. Son unas cadenas de texto usadas para identificar una instancia específica de un documento. Las *caches* usan la etiqueta de entidad para validar un documento con la cabecera de petición *If-none-match*. Dicha cabecera obliga al servidor origen a que si ninguna de las etiquetas de entidad es válida, le envíe el nuevo contenido. Pueden existir varias etiquetas de entidad asociadas a un mismo documento, porque una *cache* puede almacenar distintas versiones de un mismo documento y por tanto, asociar múltiples etiquetas de entidad a un mismo documento. Si alguna de las etiquetas es válida, ésta será devuelta en la cabecera *Etag* de la respuesta. Dichas etiquetas son opacas, las *caches* no pueden obtener ninguna información de ellas, sólo comparar unas con otras para comprobar si son iguales o no.

Se puede considerar extraño que una *cache* pueda a veces, devolver documentos sin haberlos validado previamente, pero el proceso de validación supone una penalización en forma de latencia, pues dicho proceso requiere una importante cantidad de tiempo en comparación con la simple devolución del documento sin validar. Si la *cache* es precisa en sus predicciones sobre la frescura en los documentos, puede conseguir reducciones significativas en la latencia.

En ciertos casos, existe también la posibilidad de que las *caches* devuelvan respuestas obsoletas de forma intencionada. Así, si una *cache* trata de validar un documento pero no puede conectar con el servidor origen, devolverá la respuesta obsoleta antes de no poder suministrar ninguna. También puede darse que existan *caches* que trabajen con anchos de banda muy limitados y tengan que ser configuradas para devolver respuestas obsoletas en lugar de intentar llevar a cabo la validación de los documentos.

Los usuarios también pueden indicar si desean recibir documentos obsoletos con la directiva de control de *cache max-stale*. Se puede usar con o sin valor numérico. Si no se indica número, el usuario aceptará documentos en *cache* sin importar cuanto hace que son obsoletos mientras que si se indica el número, sólo se aceptarán documentos que son obsoletos, como mucho, el número de segundos indicado.

Las preferencias de la *cache* o del usuario no son tenidas en cuenta cuando se usan las directivas de control de *cache must-revalidate* y *proxy-revalidate* pues si las respuestas contienen dichas directivas siempre tendrán que ser validados los documentos al convertirse en obsoletos.

2.4. POLÍTICAS DE REEMPLAZO PARA *PROXYS* WEB

Las técnicas de uso de memorias *cache* y de reemplazo de página de memoria virtual no son necesariamente extensibles a las *cache* Web.

Existen tres diferencias fundamentales entre ambos casos. La primera radica en que los documentos que se almacenan en una *cache* Web son de tamaño variable. Un acierto en *cache* se dará sólo si todo el documento se encuentra en la *cache* y los documentos Web presentan una gran variación de tamaño dependiendo del tipo de información que contengan (imágenes, vídeo, texto, audio, etc.).

En segundo lugar, cada página Web requiere un tiempo distinto para poder descargarse aún cuando dos páginas distintas tengan el mismo tamaño. Si un servidor *proxy* quiere reducir la latencia media del acceso a la página, tendrá que ajustar su estrategia de reemplazo en base a la latencia de descarga de cada página en concreto.

La tercera diferencia se produce porque los distintos flujos que acceden a la *cache* del servidor *proxy*, no suelen seguir una pauta específica. Los flujos recibidos por un servidor *proxy* son la unión de flujos de acceso de muchísimos usuarios distintos en lugar de provenir de unas pocas fuentes programadas, como ocurre en el caso de la paginación de memoria virtual

Las *cache proxy* se encuentran en una posición privilegiada para afectar al tráfico relativo a las páginas Web en Internet. Los algoritmos de reemplazo se encargan de decidir qué documentos se almacenan en *cache* y qué otros son sustituidos, lo cual afectará a cuáles de las futuras peticiones serán aciertos en *cache*.

Si por ejemplo, resulta más caro usar ciertos enlaces que otros, la política de reemplazo puede tratar de favorecer aquellos documentos que resulten más caros en relación a otros más baratos. Si se conoce que ciertos enlaces presentan una mayor congestión, se puede tratar de retener aquellos documentos que han de hacer uso de dichos enlaces. Por tanto la *cache proxy* puede contribuir a reducir la carga de la red almacenando documentos que requieran un mayor número de saltos para su obtención.

Las políticas de reemplazo pueden incorporar estas consideraciones asociando un coste de red apropiado a cada documento y tratando de minimizar el coste total para un determinado flujo de acceso. Tener en cuenta aspectos como el coste o el tamaño, hacen que el uso de las *cache proxy* sean un problema mucho más complicado que las memorias *cache* tradicionales.

Además, la política de reemplazo empleada determinará la ordenación que presentarán los documentos almacenados en la *cache*, pero fundamentalmente determinará cual será el siguiente documento a reemplazar de todos los presentes en *cache*, una vez que no se disponga de más espacio para almacenamiento en la misma. De este hecho dependerá el rendimiento, y por tanto, el que se pueda conseguir un mejor o peor resultado del uso de la *cache* del servidor *proxy*.

En los siguientes apartados se presenta una descripción de las cinco políticas de reemplazo empleadas en este Proyecto Fin de Carrera. Los distintos algoritmos, ante una nueva petición, determinan, caso de ser necesario, qué documento o documentos han de eliminarse de la *cache* para poder introducir el nuevo documento pedido, sin exceder el tamaño de la misma y la posición en la que se deberá de situar en ella según la prioridad que le haya sido asignada.

2.4.1. *LEAST RECENTLY USED* (LRU)

Se basa en la suposición de que documentos recientemente referenciados volverán a ser referenciados en un futuro cercano. Considera el coste y tamaño de los documentos como fijo, lo cual es una desventaja, aunque tiene como ventaja su simplicidad.

Esta política, ante la petición de un documento no presente en la *cache*, determina que se eliminarán tantos documentos como sean necesarios hasta que quede sitio suficiente para introducir el nuevo, eliminándose siempre en primer lugar aquel documento que haya sido menos recientemente referenciado. Este algoritmo presenta complejidad $O(1)$ tanto en las inserciones como en las eliminaciones de documentos.

Este algoritmo se usa habitualmente en *proxys* comerciales dada su simplicidad y velocidad, y ha funcionado muy bien para *caches* de CPU (*Central Process Unit*) y sistemas de memoria virtual. Sin embargo, no funciona tan bien para *caches proxy* porque la localidad de los tiempos de acceso para tráfico Web a menudo exhibe muy diferentes tendencias.

2.4.2. *LEAST FREQUENTLY USED* (LFU)

Es una política basada en la frecuencia de acceso a los documentos, pero también tiene en cuenta la información sobre lo reciente que ha sido una referencia, manteniendo la suposición de coste y tamaño fijo de los documentos.

En este caso se eliminará en primer lugar el documento que haya sido accedido menos frecuentemente, y al igual que en el caso anterior y que en el resto de políticas, serán eliminados tantos documentos como sean necesarios para la introducción del nuevo documento pedido. Si dos o más documentos tienen igual número de peticiones, y por tanto igual prioridad, sobre dicho subconjunto se elegirá siguiendo la política LRU.

Manejar tanto un acierto como un fallo requiere un tiempo $O(\log k)$, siendo k el número de documentos en *cache*. Una eliminación, sin embargo, tan sólo requiere un tiempo $O(1)$.

Con esta política se trata de mantener en memoria los documentos más populares y reemplazar los que son raramente usados. No obstante, algunos documentos pueden ser accedidos durante un corto período de tiempo muchas veces y tomar un valor de prioridad muy grande para después no volver a ser accedidos, lo que lleva al mantenimiento en *cache* de documentos innecesarios, siendo esta su principal desventaja. Este fenómeno es conocido como *cache pollution* (contaminación o corrupción de *cache*).

2.4.3. *GREEDY DUAL SIZE* (GDS)

El algoritmo original, *Greedy Dual* fue propuesto por Young [You91b'94] y es actualmente toda una familia de algoritmos. Está relacionado con el caso en el que todos

los documentos en la *cache* tienen el mismo tamaño pero presentan distinto coste a la hora de traerse desde otro lugar de almacenamiento (como por ejemplo un servidor remoto). El algoritmo asocia un valor H a cada documento en *cache*, que representará la prioridad o el peso de dicho documento dentro del conjunto de documentos en *cache*.

Cuando un documento es traído a *cache*, H toma el valor del coste de traer dicho documento a la *cache* (valor siempre positivo) y cuando es necesario reemplazar alguno de los documentos en *cache* para poder introducir uno nuevo, será eliminado aquél cuyo valor de H sea el mínimo de todos los almacenados en *cache* y los restantes documentos en *cache* reducen su valor de H en la cantidad que tenía el elemento eliminado.

Si un documento de la *cache* es accedido, su valor de H se reestablece al del coste de traerlo a la *cache*. Por tanto el valor de H de los documentos recientemente accedidos mantiene buena parte de coste original respecto a los que no han sido accedidos desde hace tiempo. Con esta forma de modificar el valor de H con el paso del tiempo, en el algoritmo se tiene en cuenta el aspecto de la localidad y el coste.

Greedy Dual Size [Cao-Irani'97] es una extensión de Greedy Dual, propuesta por Cao e Irani, en la que se incorpora el tamaño del documento a la hora de determinar el valor de H y con ello, qué documento ha de ser el siguiente en ser reemplazado. De esta forma, para asignar un valor de prioridad a un documento a introducir en *cache*, H se calcula como $H(p) = \text{coste}(p) / \text{tamaño}(p)$, siendo el coste, lo que cuesta traer el documento del servidor remoto y tamaño, el número de *bytes* de que consta el documento. Como se defina el coste dependerá del objetivo perseguido con el algoritmo. Así, si se fija coste a 1, el objetivo es maximizar la tasa de acierto en *cache*, si se establece a la latencia de descarga, se buscará minimizar la latencia media y si se fija al coste de la red, se tratará de minimizar el coste total.

A una primera vista nos parecería que el algoritmo requiere hacer tantas subtracciones como elementos existan en *cache* en el momento de hacerse un reemplazo. Sin embargo, se puede hacer un manejo de H distinto evitando esas restas, haciendo uso de un valor L que actúe como un desplazamiento o inflación sobre los valores futuros de H . Una forma eficiente de implementación del algoritmo se muestra en la figura 2.6.

```
Inicializar  $L \leftarrow 0$ 
Procesar cada petición de forma que:
Petición actual del documento  $p$ :
(1) si  $p$  está en memoria,
(2)    $H(p) \leftarrow L + \text{coste}(p)/\text{tamaño}(p)$ .
(3) si  $p$  no está en memoria,
(4)   mientras no haya suficiente espacio en memoria para  $p$ 
(5)     Sea  $L \leftarrow \min_{q \in M} H(q)$ 
(6)     Eliminar  $q$  tal que  $H(q) = L$ 
(7)   Introducir  $p$  en memoria y establecer
       $H(p) \leftarrow L + \text{coste}(p)/\text{tamaño}(p)$ 
fin
```

Figura 2.6. Algoritmo Greedy Dual Size

Utilizando esta política, se puede implementar el algoritmo manteniendo una cola ordenada según prioridad de los documentos en función de su valor H . Manejar tanto un acierto como un fallo requiere un tiempo $O(\log k)$, siendo k el número de documentos en *cache*. En ambos casos sólo hay que actualizar un elemento en la cola. Una eliminación, por su parte, tan sólo requiere un tiempo $O(1)$, como ocurre para todas las políticas. Este comportamiento es el mismo para todos los algoritmos de la familia Greedy Dual.

2.4.4. GREEDY DUAL SIZE FREQUENCY (GDSF)

Este algoritmo fue propuesto por Cherkasova [Cherkasova'98]. Es una mejora sobre el algoritmo GDS, en la que se incorpora de una forma sencilla las características más importantes del documento y los accesos al mismo, tales como el tamaño del documento, la frecuencia de acceso y cómo de reciente fue el último acceso. Trata de paliar el inconveniente de GDS al no tener en cuenta cuántas veces ha sido accedido un documento en el pasado a la hora de asignarle un peso y por tanto un determinado orden de reemplazo.

Por tanto, la principal diferencia respecto al anterior algoritmo radica a la hora de hacer el cálculo de H , puesto que ahora se tendrá en cuenta la frecuencia con la que ha sido accedido el documento a la hora de hacer su cálculo. La variación a introducir en el algoritmo mostrado en la figura 2.6., en la línea (2), sería:

$$H(p) \leftarrow L + \text{frecuencia}(p) \times \text{coste}(p) / \text{tamaño}(p)$$

De este modo, la inclusión del parámetro “frecuencia” supone que ante un acierto en *cache*, el valor de dicho parámetro se incrementará en uno respecto a su valor anterior a la petición, mientras que si se produce un fallo en *cache*, se le asignará un valor inicial de frecuencia igual a 1 para realizar el cálculo de H. El valor de L es monótono creciente, siendo reemplazado cada vez que se saca un documento de la memoria. Los documentos que no son solicitados durante un largo período de tiempo, no cambian el valor de su prioridad en memoria y por tanto, en algún momento el valor de L llegará a ser suficientemente grande como para que cualquier nuevo documento tenga una mayor prioridad que los documentos no solicitados por un largo espacio de tiempo y que sean estos los reemplazados. Con este mecanismo de envejecimiento (*aging*) se evita la contaminación en *cache*.

2.4.5. *GREEDY DUAL** (GD*)

Este algoritmo, descrito por Lindemann y Waldhorst [Lindemann’02], es también una generalización de GDS. Con él, se trata de tener en cuenta la importancia relativa de la popularidad a largo plazo en contraposición a la correlación temporal de las referencias a corto plazo.

A la hora de desarrollar políticas de reemplazo para *caches proxy*, es importante caracterizar el grado de localidad presente en los flujos de peticiones Web habituales. Estudios realizados han demostrado que la localidad temporal está decreciendo [Barford’99]. Un motivo para este hecho se atribuye al uso efectivo de la *cache* de usuario, puesto que las peticiones generadas por un determinado usuario son precisamente el conjunto de fallos en la *cache* del navegador del usuario. Dicho flujo de peticiones suele exhibir una débil localidad temporal, en particular un documento recientemente accedido es poco probable que sea accedido de nuevo en el futuro.

Usando un modelo de referencia independiente [Coffman’73] se ha demostrado que la probabilidad de referenciar un documento t unidades de tiempo después de que haya sido referenciado, es proporcional a $1/t$, siendo ésta la distribución de probabilidad de los

tiempos de referencia entre peticiones. Dicha distribución se puede emplear como modelo de localidad temporal, aunque no es la única fuente de localidad temporal en los flujos de peticiones Web. También influye sobre ella la popularidad de un documento, siendo por tanto, la correlación en el tiempo y la popularidad las dos dimensiones que presenta la localidad temporal.

Se plantea entonces la pregunta de si la caracterización de localidad temporal usando el modelo de referencia entre llegadas (o peticiones) puede cuantificar los efectos de la popularidad y la correlación temporal independientemente. Se ha demostrado que no es el caso, pues la distribución de referencias entre llegadas se ve predominantemente afectada por el perfil de popularidad de los documentos en el flujo de peticiones Web.

La importante relación entre popularidad y localidad temporal a menudo oculta otro importante aspecto de la temporalidad local, como es la correlación temporal de referencias repetidas a los mismos documentos. Por ello, para caracterizar el nivel de correlación temporal se han de “ecualizar” los efectos de la popularidad. Así, se consideró la distribución de probabilidad de referencias entre llegadas para documentos con igual popularidad y se cuantificó el grado de correlación temporal mediante el parámetro β , la pendiente de la distribución en escala logarítmica de la referencia entre llegadas para documentos con igual popularidad, para los cuales, la probabilidad de que el tiempo de referencia entre llegadas sea igual a t , es aproximadamente proporcional a $t^{-\beta}$ a corto plazo, que por tanto caracteriza la correlación de referencias.

El valor de β suele variar de forma significativa entre distintas trazas y suele tomar valores entre 0 y 1, aunque existe la posibilidad de que β sea mayor que 1, pero no es muy habitual, siendo lo más habitual que oscile entre 0.3 y 0.7.

Si β toma valores próximos a la unidad, GD* se convierte en GDSF, mientras que si β está próximo a cero, degenera en una variación de LFU usando un coste normalizado. En este caso el envejecimiento dinámico no debe ser usado y si se supone un coste constante el algoritmo resultante estará basado en el tamaño y tendrá en cuenta la frecuencia siendo una buena elección cuando no exista correlación o sea muy débil, pero para trabajar con tráfico Web, que exhibe correlación de referencias, en *caches proxy*, no es la mejor elección.

En el algoritmo GD* se redefine el valor de la prioridad o el peso de un documento. Dicho valor reflejará el ahorro normalizado en coste esperado, guardando dicho documento en *cache*. Además utiliza un mecanismo de envejecimiento dinámico similar al usado en GDS y GDSF mediante un valor L de inflación. El único problema adicional será el reflejar tanto la utilidad del documento como el grado de correlación. Para ello, y puesto que las referencias entre llegadas para documentos con igual popularidad sigue una ley $t^{-\beta}$, la máxima duración de tiempo de un documento en *cache* debe ser proporcional a $u(p)^{1/\beta}$, siendo $u(p) = \text{frecuencia}(p) \times \text{coste}(p) / \text{tamaño}(p)$.

En este caso a la hora de hacer el cálculo de H, la expresión a emplear en la línea (2), en el algoritmo descrito en la figura 2.6 sería:

$$(2) \quad H(p) \leftarrow L + (\text{frecuencia}(p) \times \text{coste}(p) / \text{tamaño}(p))^{1/\beta}$$

En esta expresión los distintos elementos tienen igual significado y comportamiento que en GDSF a excepción de β , específica de este algoritmo y que ha sido anteriormente descrita.

Con este algoritmo se tiene en cuenta tanto la popularidad como la correlación temporal. La frecuencia se encarga de proporcionar la popularidad a largo plazo mientras que la correlación temporal se tiene en cuenta mediante la tasa de envejecimiento controlada por el parámetro β . Valores pequeños de β suponen una débil correlación de referencias y por tanto los documentos sufren un envejecimiento más lento, mientras que si los valores son mayores el efecto es el inverso.

CAPÍTULO 3: Emulador *Proxy*

En primer lugar, se presenta una introducción a la programación orientada a objetos y sus principios. En el segundo apartado se muestran las principales características del lenguaje de programación JAVA. Para finalizar se describirá la estructura del emulador, tanto en lo referente a su interfaz gráfico como a su funcionamiento interno y consideraciones tenidas en cuenta. Además, se explicarán algunas de las pruebas llevadas a cabo para comprobar su correcto funcionamiento y se suministrarán algunas recomendaciones para el uso e instalación del emulador.

3.1. INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

La orientación a objetos es un paradigma de programación que facilita la creación de software de calidad por sus factores que potencian el mantenimiento, la extensión y la reutilización del software generado bajo este paradigma.

Se basa en tratar de amoldarse al modo de pensar del hombre y no al de la máquina gracias a la forma racional con la que se manejan las abstracciones que representan las entidades del dominio del problema, y a propiedades como la jerarquía o el encapsulamiento.

El elemento básico de este paradigma no son las funciones, como sucedía en la programación estructurada, sino un ente denominado objeto. Un objeto se puede definir como el "encapsulamiento de un conjunto de operaciones (métodos) que pueden ser invocados externamente, y de un estado que recuerda el efecto de los servicios". [Piattini'96]. Es en definitiva, la representación de un concepto para un programa, y contiene toda la información necesaria para abstraer dicho concepto, es decir, los datos que describen su estado y las operaciones que pueden modificar dicho estado, y que determinan las capacidades del objeto.

Por tanto un objeto además de un estado interno, presenta una interfaz para poder interactuar con el exterior. Es por esto por lo que se dice que en la programación orientada a objetos "se unen datos y procesos", y no como en su predecesora, la programación estructurada, en la que estaban separados en forma de variables y funciones.

Un objeto consta de:

- Tiempo de vida: duración de un objeto en un programa. Siempre está limitada en el tiempo y la mayoría de los objetos sólo existen durante una parte de la ejecución del programa. Los objetos son creados mediante un mecanismo denominado instanciación, y cuando dejan de existir se dice que son destruidos.
- Estado: Todo objeto posee un estado, definido por sus atributos. Con él se definen las propiedades del objeto, y el estado en que se encuentra en un momento determinado de su existencia.
- Comportamiento: Todo objeto ha de presentar una interfaz, definida por sus métodos, para que el resto de objetos que componen los programas puedan interactuar con él.

Otro de los conceptos fundamentales de la programación orientada a objetos es el de clase. Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común. Se podría definir como "un conjunto de cosas (físicas o abstractas) que tienen el mismo comportamiento y características... Es la implementación

de un tipo de objeto (considerando los objetos como instancias de las clases)". [Piattini'96].

Se trata, básicamente, de una plantilla para la creación de objetos ya que al crear un objeto (instanciación) se ha de especificar de qué clase es el objeto instanciado, para que el compilador conozca las características del mismo.

Las clases presentan el estado de los objetos a los que representan mediante variables denominadas atributos. Cuando se instancia un objeto el compilador crea en la memoria dinámica un espacio para tantas variables como atributos tenga la clase a la que pertenece el objeto. Los métodos (también conocidos como servicios) son las funciones mediante las que las clases representan el comportamiento de los objetos. En dichos métodos se modifican los valores de los atributos del objeto, y representan las capacidades del objeto.

3.1.1. PRINCIPIOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

Existen una serie de principios fundamentales para comprender cómo se modela la realidad al crear un programa bajo el paradigma de la orientación a objetos. Estos principios son: la abstracción, el encapsulamiento, la modularidad, la jerarquía, el paso de mensajes y el poliforfismo.

3.1.1.1. Principio de abstracción

Mediante la abstracción la mente humana modela la realidad en forma de objetos, para lo cual busca parecidos entre la realidad y la posible implementación mediante objetos del programa que simulen el funcionamiento de los objetos reales.

Se basa en el hecho de que no necesitamos conocer los detalles de porqué ni cómo funcionan las cosas, simplemente solicitamos determinadas acciones en espera de una respuesta.

Además la abstracción humana se gestiona de una manera jerárquica, dividiendo sucesivamente sistemas complejos en conjuntos de subsistemas, para así entender más fácilmente la realidad. Esta es la forma de pensar que el paradigma de la orientación a objetos intenta cubrir.

3.1.1.2. Principio de encapsulamiento

Este principio permite a los objetos elegir qué información es publicada y qué información es ocultada, para lo que suelen presentar sus métodos como interfaces públicas y sus atributos como datos privados e inaccesibles desde otros objetos.

Para permitir que otros objetos consulten o modifiquen los atributos de los objetos, las clases suelen presentar métodos de acceso. De esta manera el acceso a los datos de los objetos es controlado por el programador, evitando efectos laterales y accesos no deseados.

Con el encapsulado de los datos se consigue que las personas que utilicen un objeto sólo tengan que comprender su interfaz, olvidándose de cómo está implementada, y en definitiva, reduciendo la complejidad de utilización.

3.1.1.3. Principio de modularidad

Mediante la modularidad, se propone al programador dividir su aplicación en varios módulos diferentes (ya sea en forma de clases, paquetes o bibliotecas), cada uno de ellos con un sentido propio.

De esta forma se disminuye el grado de dificultad del problema al que da respuesta el programa, pues este se afronta como un conjunto de problemas más sencillos, de menor dificultad, además de facilitar la comprensión del programa.

3.1.1.4. Principio de jerarquía

Las distintas clases de un programa se organizan de una forma jerárquica. La representación de dicha organización da lugar a los denominados árboles de herencia, como se muestra en la figura 3.1.

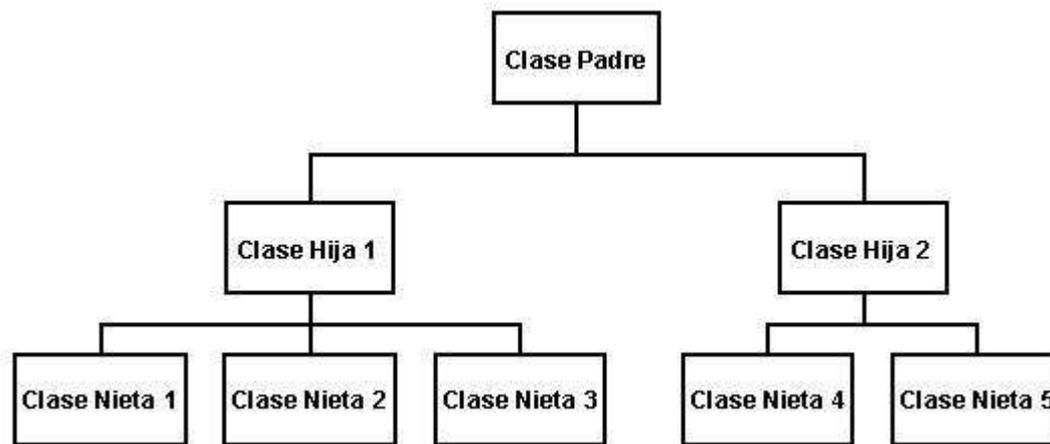


Figura 3.1. Ejemplo de árbol de herencia

Mediante la herencia una clase hija puede tomar determinadas propiedades de una clase padre, simplificándose así los diseños y evitándose la duplicación de código al no tener que volver a codificar métodos ya implementados.

3.1.1.5. Principio de paso de mensajes

Mediante el denominado paso de mensajes, un objeto puede solicitar de otro que realice una acción determinada o que modifique su estado. Se suele implementar como llamadas a los métodos de otros objetos.

3.1.1.6. Principio de polimorfismo

Polimorfismo quiere decir "un objeto y muchas formas". Esta propiedad permite que un objeto presente diferentes comportamientos en función del contexto en que se encuentre. Por ejemplo un método puede presentar diferentes implementaciones en función de los argumentos que recibe, recibir diferentes números de parámetros para realizar una misma operación, y realizar diferentes acciones dependiendo del nivel de abstracción en que sea llamado.

3.2. EL LENGUAJE DE PROGRAMACIÓN JAVA

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems. Sus creadores describen al lenguaje Java de la siguiente manera: Simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutral, portable, de alto rendimiento, multitarea y dinámico.

La principal diferencia entre Java y los demás lenguajes de programación orientados a objetos es que con estos últimos es posible programar mediante el uso de objetos, aunque no es obligatorio, mientras que con Java se tiene necesariamente que programar haciendo uso de objetos, incorporando el uso de la orientación a objetos como uno de los pilares básicos del lenguaje.

3.2.1. CARACTERÍSTICAS DEL LENGUAJE JAVA

3.2.1.1. Simplicidad

Java ofrece toda la funcionalidad de un lenguaje potente, como C y C++, pero sin las características menos usadas y más confusas de éstos. Fue diseñado para ser parecido a los lenguajes antes mencionados y así facilitar un rápido y fácil aprendizaje del mismo. Es más complejo que un lenguaje simple, pero más sencillo que cualquier otro entorno de programación.

Reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos y además ofrece características muy útiles como disponer de estructuras de datos complejas y sus métodos asociados.

3.2.1.2. Orientado a objetos

Java fue diseñado como un lenguaje orientado a objetos desde el principio, soportando las características propias del paradigma de orientación a objetos como son la encapsulación, herencia y polimorfismo, entre otras, con lo que se le puede sacar todo el partido a dichas características. Además la programación orientada a objetos aparece como la tendencia del futuro, especialmente en entornos cada vez más complejos y basados en red.

3.2.1.3. Distribuido

Java proporciona extensas capacidades de interconexión TCP/IP, existiendo librerías de rutinas para acceder e interactuar con protocolos como HTTP y FTP. Esto permite acceder a la información a través de la red con tanta facilidad como a los ficheros locales, facilitando así la creación de aplicaciones distribuidas.

3.2.1.4. Robusto

Java fue diseñado para crear software altamente fiable. Para ello realiza numerosas comprobaciones tanto en tiempo de compilación como en tiempo de ejecución, obligando a la declaración explícita de métodos reduciendo las posibilidades de error. La comprobación de tipos que realiza ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Al ser estricto en cuanto a tipos y declaraciones, la mayor parte de los errores se detectan en tiempo de compilación.

Sus características de memoria liberan a los programadores de una familia entera de errores (la aritmética de punteros), ya que se ha prescindido por completo de los punteros, y la recolección de basura elimina la necesidad de liberación explícita de memoria.

3.2.1.5. Interpretado y compilado a la vez

Java es compilado, en la medida en que su código fuente se transforma en una especie de código máquina, los *bytecodes* (o códigos de *byte*), semejantes a las instrucciones de ensamblador, y que se puede ejecutar sobre cualquier plataforma.

Por otra parte, es interpretado, ya que los *bytecodes* se pueden ejecutar directamente sobre cualquier máquina a la cual se hayan portado el intérprete y el sistema de ejecución en tiempo real (*run-time*), que es totalmente dependiente de la máquina y del sistema operativo.

Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el sistema de ejecución en tiempo real correspondiente al sistema operativo utilizado.

3.2.1.6. Indiferente de la arquitectura

Java está diseñado para que un programa escrito en este lenguaje sea ejecutado correctamente independientemente de la plataforma en la que se esté trabajando (Macintosh, PC, UNIX...). Para ello crea los códigos de *byte*, que pueden interpretarse en cualquier sistema operativo con un intérprete de Java.

La desventaja de un sistema de este tipo es el rendimiento; sin embargo, el hecho de que Java fuese diseñado para funcionar razonablemente bien en microprocesadores de escasa potencia, unido a la sencillez de traducción a código máquina hacen que Java supere esa desventaja. Sin embargo, la ventaja que se tiene es que una vez generados esos códigos de *byte*, toda máquina con un sistema de ejecución apropiado (*run-time*) podrá ejecutar dicho código sin importar la máquina que lo generó.

Por tanto, haciendo uso de Java lo único verdaderamente dependiente del sistema es la Máquina Virtual Java o *Java Virtual Machine* (JVM) y las librerías fundamentales, que nos permitirían acceder directamente al hardware de la máquina.

3.2.1.7. Portable

La indiferencia respecto a la arquitectura representa sólo una parte básica de su portabilidad. Además, Java especifica los tamaños de sus tipos de datos básicos y el comportamiento de sus operadores aritméticos, de manera que los programas son iguales en todas las plataformas. Estas dos últimas características se conocen como la Máquina Virtual Java.

3.2.1.8. Multihebra

Java soporta sincronización de múltiples hilos de ejecución a nivel de lenguaje, especialmente útiles en la creación de aplicaciones de red distribuidas. Al estar los hilos construidos en el propio lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++. El beneficio que se obtiene al usar estos múltiples hilos es un mejor rendimiento interactivo y mejor comportamiento en tiempo real.

3.3. ESTRUCTURA DEL EMULADOR DISEÑADO

A continuación, se llevará a cabo una descripción de la funcionalidad del emulador diseñado. Para una mejor descripción, se acompañará la explicación de imágenes reales, capturadas de pantalla, en las que se muestran cada una de las partes de interfaz, tal y como las vería un usuario de dicho emulador.

3.3.1. DESCRIPCIÓN DEL INTERFAZ CON EL USUARIO

Una vez ejecutada la aplicación sobre el JRE (*Java Runtime Enviroment*, o entorno de ejecución Java) correspondiente a la arquitectura en cuestión, al usuario se le muestra la pantalla principal de la aplicación que se presenta en la figura 3.2. para que el usuario pueda seleccionar los parámetros a emplear en la emulación.

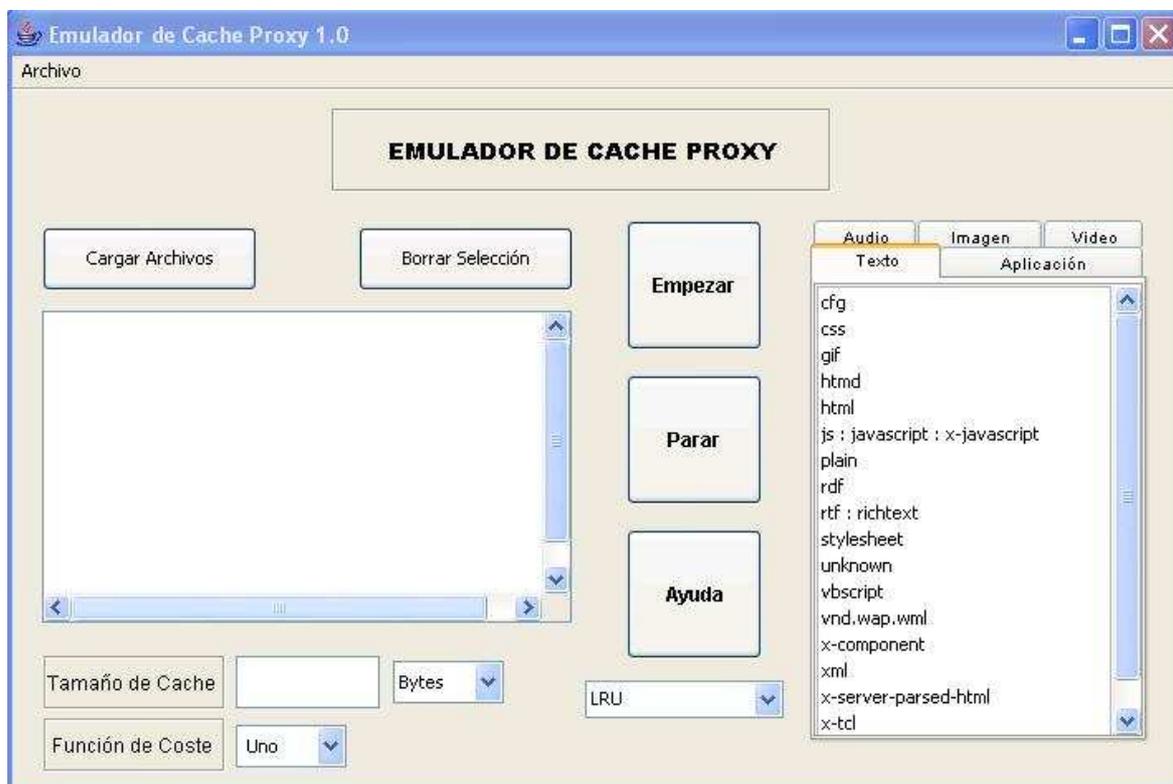


Figura 3.2. Pantalla principal de la aplicación

Cada una de las partes de la pantalla principal se emplea para la selección o introducción de los parámetros a usar en la emulación o para la elección de la siguiente acción a realizar. Su funcionalidad se detalla a continuación.

3.3.1.1. Botones para el manejo de los archivos con muestras de tráfico

Como se muestra en la figura 3.3, existen dos botones para poder seleccionar las muestras de tráfico a emplear o bien para deshacer dicha selección.



Figura 3.3. Botones para manejar los archivos con muestras de tráfico

El botón “Cargar archivos” se emplea para seleccionar los archivos que contienen las muestras de tráfico que se desean usar en la emulación. Al pulsar dicho botón, aparecerá una nueva ventana, que se muestra en la figura 3.4., que permite seleccionar tantas trazas a emplear en las emulaciones como se requiera.

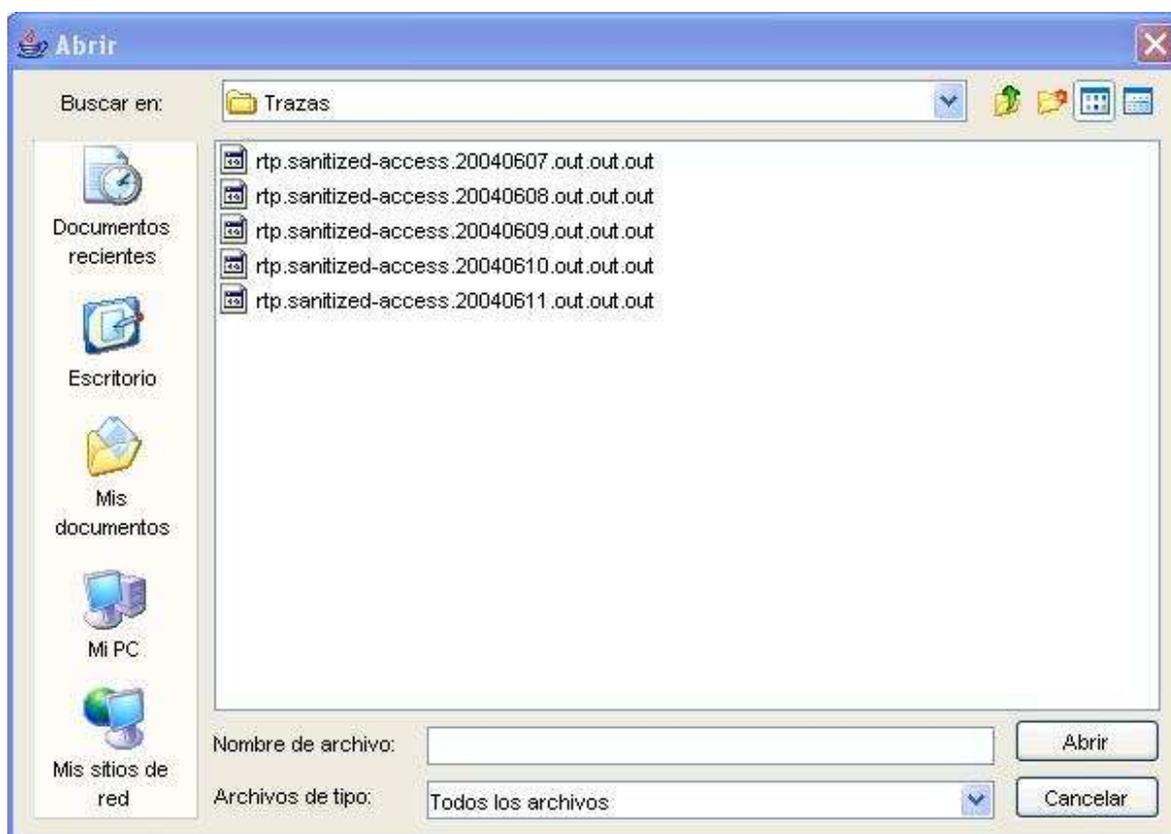


Figura 3.4. Ventana para cargar archivos

Dicha ventana tendrá el formato del sistema operativo instalado en la máquina concreta. Una vez seleccionadas las trazas y pulsado el botón “Abrir”, la ventana se cerrará y se nos volverá a mostrar nuestra pantalla principal donde los nombres de las trazas seleccionadas aparecerán en el panel situado justo debajo del botón “Cargar archivos”. Un ejemplo de esto se observa en la figura 3.5.

Junto al botón antes citado, existe otro denominado “Borrar selección” que permite eliminar todas las trazas hasta ese momento seleccionadas para llevar a cabo la emulación, si nos percatamos de algún error en la selección de las mismas.

3.3.1.2. Panel de trazas seleccionadas

Este panel, situado debajo de los botones de manejo de muestras de tráfico, aparece vacío al iniciar la aplicación. Una vez se hayan seleccionado una o más muestras de tráfico para realizar la emulación, aparecerán reflejados en dicho panel los nombres de los archivos seleccionados. Un ejemplo de esto se puede observar en la figura 3.5.

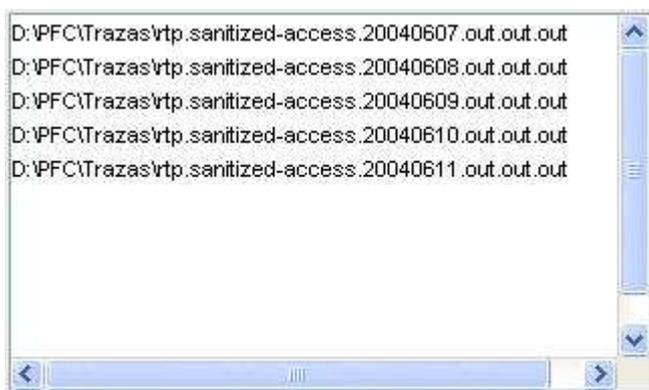


Figura 3.5. Ejemplo de panel de trazas seleccionadas con un ejemplo de selección

3.3.1.3. Área para la selección del tamaño de *cache*

Debajo del panel donde se presentan las trazas seleccionadas, aparece una etiqueta con la indicación “Tamaño de *cache*”. Junto a dicha etiqueta existe un recuadro en blanco donde se podrá introducir el tamaño de *cache* que se desee emplear a la hora de hacer la emulación. Para hacer menos engorrosa la introducción de dicho número, junto al recuadro

antes mencionado, se encuentra un menú desplegable donde se puede seleccionar la unidad del número introducido en el recuadro de entre Bytes, KBytes, MBytes y GBytes. En la figura 3.6. se observa el aspecto del área para la selección del tamaño de *cache*.



Figura 3.6. Área para la selección del tamaño de *cache*

3.3.1.4. Área para la selección de la función de coste

Bajo el área para la selección del tamaño del *cache* se encuentra el área donde se puede seleccionar la función de coste que se empleará en la emulación, y que se muestra en la figura 3.7. Junto a la etiqueta con la indicación “Función de coste” hay un menú desplegable para poder seleccionar la función de coste. Por defecto, la función de coste empleada es “Uno” aunque se puede elegir entre coste “Uno” o “*Packets*” a la hora de realizarse la emulación.



Figura 3.7. Área para la selección de la función de coste

La función de coste se utiliza para determinar la forma en la que se calculará la prioridad asociada a cada documento, y es aplicable sólo para ciertas políticas (esta función será explicada más en profundidad en el siguiente capítulo). En los casos en los que se emplee como política de reemplazo LRU o LFU, la elección de la función de coste no tendrá efecto alguno en los resultados, ya que dichas políticas no la tienen en cuenta a la hora de determinar la prioridad de los documentos. Por tanto, sólo tendrá efecto para las políticas *Greedy Dual Size*, *Greedy Dual Size Frequency* y *Greedy Dual**.

3.3.1.5. Botones para el control del funcionamiento del emulador

En la parte central del emulador se encuentran los botones que se emplean para el control del funcionamiento del emulador. Dichos botones se muestran en la figura 3.8.



Figura 3.8. Botones para el control del funcionamiento del emulador

El botón titulado “Empezar” se emplea para dar comienzo a la ejecución una vez todos los parámetros han sido seleccionados o introducidos correctamente. Caso de no ser así, se presentarán mensajes de error en pantalla. Estos mensajes se producen cuando el tamaño de *cache* introducido no sea correcto, si no se ha seleccionado ningún subtipo de documento a usar en la emulación o si alguno de los archivos elegidos con la muestra de tráfico a usar, no posee el formato adecuado. Por ejemplo, el segundo de dichos mensajes se muestra en la figura 3.9.

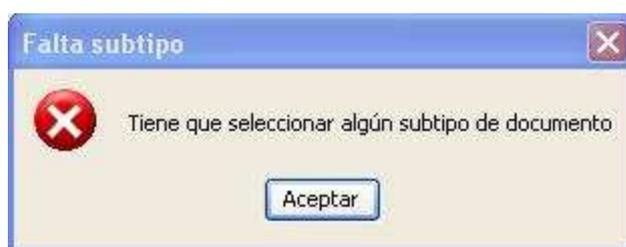


Figura 3.9. Ejemplo de mensaje de error

El segundo botón titulado “Parar” se emplea para detener la emulación antes de que concluya de forma normal (una vez se hayan procesado todas las peticiones de todos los archivos de traza seleccionados). Es el botón a emplear si se desea abortar una emulación, por ejemplo, al detectar que alguno de los parámetros que se está empleando en la emulación no es exactamente el que se quería emplear.

3.3.1.6. Botón de ayuda

Este botón, situado justo debajo de los botones que controlan el funcionamiento del emulador, y cuyo aspecto se observa en la figura 3.10., se encarga de mostrar una nueva ventana con una rápida descripción de la pantalla principal del emulador, una vez es presionado.



Figura 3.10. Botón de ayuda

3.3.1.7. Área para la selección de la política de reemplazo

Bajo los anteriores botones se encuentra un menú desplegable como el que se muestra en la figura 3.11., en el que se puede seleccionar la política de reemplazo a utilizar en la emulación. Las opciones son: LRU, LFU, GreedyDualSize, GreedyDS-Frequency y GreedyDual* y que se corresponden con cada una de las políticas a evaluar en este proyecto.



Figura 3.11. Menú desplegable para la elección de la política de reemplazo

3.3.1.8. Área para la selección del tipo y subtipo/s de documentos a almacenar en *cache*

En la parte derecha de la pantalla principal de la aplicación aparece un panel donde se podrá seleccionar el tipo y subtipo/s de los documentos que se guardarán en *cache*. Dicho panel presenta las pestañas "Audio", "Imagen", "Vídeo", "Texto" y "Aplicación", correspondientes a los cinco tipos de documentos que se pueden seleccionar para introducir en *cache* y a su vez, en el panel correspondiente a cada pestaña, aparece una lista con los distintos subtipos asociados a los tipos de documentos antes mencionados. En la figura

3.12. se muestra el aspecto de cada una de las pestañas, aunque simultáneamente, el usuario sólo verá una.

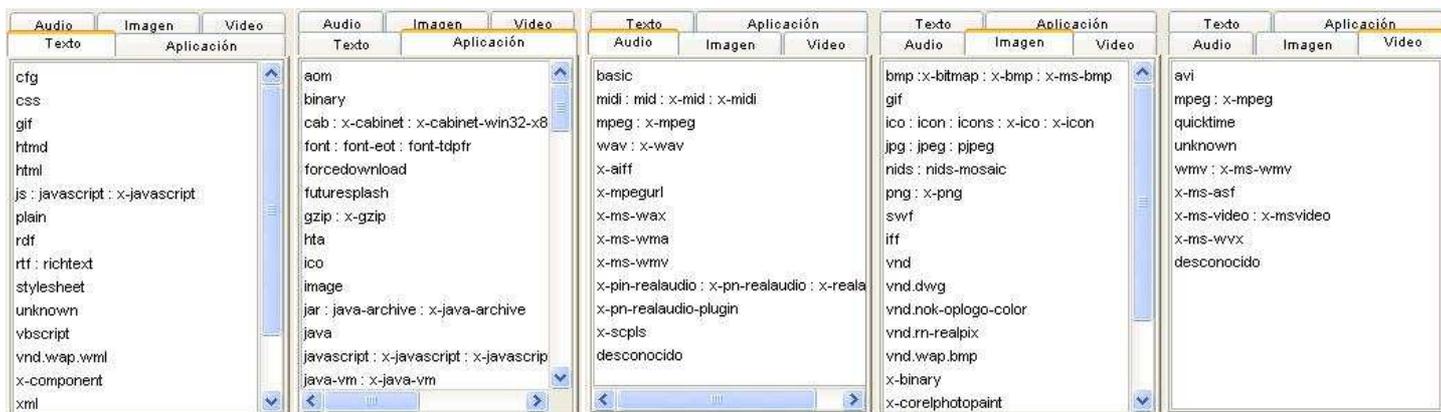


Figura 3.12. Pestañas correspondientes a los distintos tipos de documentos

Se requerirá que al menos uno de estos subtipos sea seleccionado a la hora de hacerse la emulación. Además hay que notar que en cada simulación sólo podrá introducirse en *cache* un único tipo de datos por lo que si se seleccionan subtipos de datos de distintos tipos de documento, tan sólo se considerarán, a la hora de hacerse la emulación, los subtipos que hayan sido seleccionados del tipo de documento cuyo panel se encuentre activo en el momento de pulsar el botón “Empezar”, es decir, siendo mostrado al usuario.

3.3.1.9. Mensaje de final de ejecución

Al realizarse una emulación y concluir correctamente, se presentará al usuario una ventana como la de la figura 3.13., para indicarle que la emulación ha finalizado y, por tanto, se habrá generado el correspondiente archivo de resultados.

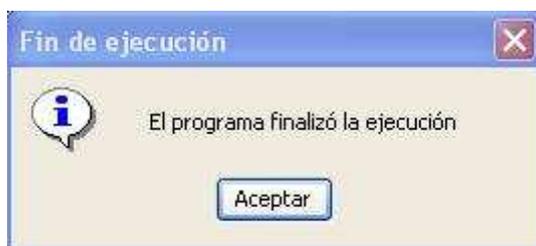


Figura 3.13. Mensaje de fin de ejecución

Una vez pulsado el botón “Aceptar”, el usuario podrá modificar los datos introducidos en el menú de inicio y realizar una nueva emulación, caso de desearlo.

3.3.2. IMPLEMENTACIÓN DEL EMULADOR

Una vez el usuario lleva a cabo la ejecución de la aplicación, se le presenta el interfaz descrito en el apartado anterior. El usuario procederá a establecer los parámetros que desee para llevar a cabo la emulación y una vez verificados todos como válidos serán pasados al programa principal para su uso en la emulación. El interfaz y el resto de la emulación se ejecutan en hilos de computación distintos, de forma que si en un momento de la misma se desea detener la ejecución, por ejemplo, porque se detectó un error en los parámetros elegidos, desde el hilo del interfaz (a través del botón “Parar” de la pantalla principal de la aplicación) se podrá efectuar la detención del proceso.

Se ha elegido como estructura interna empleada por el emulador, para emular una *cache proxy*, una lista enlazada y ordenada. En dicha lista los documentos se encuentran siempre ordenados de menor a mayor prioridad con lo que todo reemplazo que se tenga que llevar a cabo se producirá siempre del primero o primeros elementos de la lista. Para el mantenimiento del orden en dicha lista, la inserción de elementos en la misma se tendrá que llevar a cabo siempre de forma ordenada, viniendo marcada la prioridad asignada a cada elemento por la política de reemplazo elegida. La búsqueda del punto de inserción en la lista se implementó mediante un algoritmo de búsqueda dicotómica o binaria, con el que se consigue un tiempo de búsqueda de orden $O(\log(n))$, siendo n el número de elementos en la lista.

Además, en paralelo a dicha lista, se emplea una tabla Hash, que es básicamente un diccionario que permite un almacenamiento y posterior recuperación, de forma eficiente, de elementos (los denominados valores) a partir de otros elementos, las claves. Las claves son procesadas mediante una función Hash que devolverá un código Hash que determina dónde se encontrará el elemento almacenado o a almacenar en la estructura.

El código Hash tiene la propiedad de que dos claves iguales, procesadas mediante la función Hash, darán lugar siempre a un mismo código Hash. Las tablas Hash pueden proporcionar tiempos constantes de búsqueda de orden $O(1)$ de media. En comparación con otras estructuras de datos, son las más útiles cuando se tengan gran cantidad de documentos almacenados.

En nuestro caso, se empleará como clave el identificador del documento, que es único para cada documento, siendo el valor el documento en sí. Gracias al uso de dicha tabla, se llevará a cabo una búsqueda más efectiva y se puede conocer con una mayor rapidez si un elemento está ya en *cache* o no y agilizar de esa forma el funcionamiento del emulador.

Los datos manejados por el programa, provienen de la lectura de la información referente a cada una de las peticiones, registradas durante el periodo de observación que dio lugar a las trazas empleadas en las emulaciones. Dichos datos son la fecha de la petición, tamaño de la misma, identificador, tipo de documento solicitado y subtipo del correspondiente tipo.

Una vez obtenida esta información de las trazas y usada conjuntamente con los parámetros establecidos por el usuario en el menú de inicio, se puede llevar a cabo la decisión sobre qué documentos se introducirán en *cache* y cuáles no. Disponiendo de estos datos, se realiza una criba de las peticiones obtenidas de las trazas, en primer lugar, teniendo en cuenta la coincidencia o no del tipo de documento seleccionado como parámetro de la simulación en el menú de inicio y el tipo de documento leído de la muestra de tráfico. Si en dicho parámetro existe coincidencia, se comprobará si el tamaño del documento a introducir en *cache* no excede del tamaño total de la *cache* seleccionada, ya que de ser así no se tendrá en consideración el poder guardar dicho documento en *cache*. De ser de tamaño inferior, dicho documento será almacenado en *cache* aún cuando ello requiera reemplazar de la *cache* a todos los demás documentos, que hasta ese momento estuviesen guardados.

Previamente a la inserción del documento en *cache*, también se habrá tenido que comprobar si el subtipo de la petición leída de la traza se encuentra dentro del subtipo o subtipos seleccionados por el usuario para hacer la emulación, y una vez todo esto se cumpla se procederá a la inserción del mismo en la lista y en la tabla, y en caso de ser necesario, al reemplazo de los elementos suficientes como para que se pueda llevar a cabo su inclusión en *cache*.

Para conseguir un funcionamiento más óptimo, se aprovechó la propiedad que presentan las peticiones contenidas en la traza. Dicha propiedad consiste en que cualquier nueva petición tendrá un identificador mayor que cualquiera de las ya existentes. De este modo,

comprobando que el identificador sea mayor que el más grande de los que ya hayan llegado se puede saber con seguridad que el documento no está en *cache* y por tanto, no es necesaria su búsqueda en la tabla Hash, con el consiguiente ahorro de tiempo. En el caso contrario, no se tiene la certeza de que el documento se encuentre en *cache*, ya que pudo haber sido almacenado y posteriormente extraído, por lo que dicha búsqueda sí que se tendrá que llevar a cabo.

En el diagrama de flujo presentado en la figura 3.14. se muestra la estructura seguida por la clase “Emulador” que es la encargada de controlar principalmente, el funcionamiento de la aplicación y se detallan las distintas consideraciones tenidas en cuenta.

Se ha tenido en cuenta, el caso especial en el que un documento se tiene que guardar en *cache* y por su tamaño, da lugar a que todos los documentos almacenados tengan que ser extraídos. Para dicho caso se lleva a cabo el borrado completo de la *cache* (y de la tabla Hash, pues ambas se gestionan en paralelo) en lugar de sacar uno por uno los documentos. Cuando el documento a introducir no requiere que sean sacados todos los demás almacenados en *cache*, es necesario ir sacando uno a uno los documentos, pues hay que comprobar, tras sacar cada documento y ya queda espacio para insertar el nuevo o hay que seguir sacando. Sin embargo, teniendo en cuenta el caso anterior, la operación se realiza de forma más rápida y efectiva.

A la hora de gestionar los aciertos en *cache*, se han tenido en cuenta distintas circunstancias. Si se detecta que el nuevo documento leído de la traza posee un identificador cuyo número coincide con el de un documento ya presente en memoria, se podría considerar que se ha producido un acierto en *cache*. Sin embargo esto sólo será cierto si el tamaño del nuevo documento coincide exactamente con el del que ya está presente en *cache*.

La otra circunstancia en la que se considerará también dicho acierto en *cache*, será cuando el tamaño del nuevo documento, candidato a ser almacenado en *cache*, sea menor que el ya existente, en un 5% o más. Bajo dichas circunstancias, se considera que la transferencia se ha cortado pero será contabilizado como acierto.

En el caso, en el que el nuevo documento tenga un tamaño menor que el previamente almacenado, pero su disminución en tamaño no llegue a ser del 5%, se asumirá que el documento ha sido modificado [Lindemann'02] y por tanto será contabilizado como un fallo en *cache*. De igual forma se considerará fallo en *cache* si el tamaño del nuevo documento fuera mayor que el previamente almacenado, pues ello también indicaría que ha sido modificado.

Una vez finalice el procesado de todas las muestras de tráfico, se generará un archivo de texto en la carpeta `c:\EmulCacheProxy\Resultados\ + tipo de objetos almacenados en cache` (como por ejemplo, `c:\EmulCacheProxy\Resultados\Texto`), en el que aparecerán distintos datos de interés relativos a la emulación realizada, como son:

- Los parámetros elegidos en la pantalla principal de la aplicación:
 - el tamaño de *cache* a emplear en la emulación.
 - la política de reemplazo empleada (numeradas de cero a cuatro, lo que corresponde en orden a, LRU, LFU, GDS, GDSF y GD*).
 - el tipo de documentos almacenados, numerados de cero a cinco siguiendo la codificación explicada en el apéndice C (Codificación de tipos y subtipos de documentos).
 - el subtipo o la lista de subtipos elegida.
 - el número de archivos con muestras de tráfico a emplear así como la ruta completa de la que se obtuvieron.
- Las fechas de inicio y finalización de la emulación
- El tamaño de la lista, de la tabla Hash y espacio libre que quedó en *cache*
- Número de veces que no se ha almacenado un objeto por exceder el tamaño de la *cache*.
- Número de veces que el documento era del mismo tipo que el elegido, de distinto tipo o de distinto subtipo.
- Número de veces que se actualizó el documento por corte de transmisión, o que se produjo fallo en *cache* por tener un tamaño mayor que el almacenado o por tener un tamaño menor que el almacenado en menos del 5% de este.
- Número total de aciertos y bytes acertados así como número total de peticiones y de bytes pedidos del tipo de documento concreto.
- *Hit Rate*(HR) y *Byte Hit Rate*(BHR).

De todos estos datos, los dos de especial importancia son el HR y el BHR, pues serán posteriormente empleados para la obtención de las gráficas y el análisis de los resultados asociados a cada tipo de documentos. Sin embargo, el resto de datos también son relevantes, sobre todo a la hora de comprobar el correcto funcionamiento del emulador o poder dar explicación a comportamientos aparentemente anómalos como se podrá comprobar posteriormente.

Para un conocimiento más profundo de la estructura del emulador, en el Apéndice A se presentan distintos diagramas UML representativos de la estructura del emulador diseñado.

3.3.3. PRUEBAS REALIZADAS PARA VERIFICAR EL CORRECTO FUNCIONAMIENTO DEL EMULADOR

Una vez finalizada la programación del emulador, se llevaron a cabo una serie de pruebas con el fin de verificar su correcto funcionamiento. Con estas pruebas se buscó verificar que el emulador funcionaba correctamente para todas y cada una de las políticas de reemplazo, en base a los parámetros fijados por el usuario en el menú de inicio y teniendo en cuenta las distintas circunstancias a las que se podría enfrentar.

Las pruebas se basaron en la impresión en pantalla de mensajes, a lo largo de la ejecución del emulador, en los momentos determinantes de dicha ejecución y mostrando los valores de las variables adecuadas para determinar la corrección del proceso.

Para realizar las pruebas se emplearon trazas pequeñas (en torno a las 50 líneas) en las que se insertaron líneas con unas características y ubicación específica, buscando poder así comprobar todas las posibles situaciones a las que se puede enfrentar el emulador y comprobar que en efecto, resolvía estas situaciones de forma satisfactoria.

En el apéndice B se presenta una de dichas trazas empleadas, así como la secuencia de mensajes impresos en pantalla durante la emulación de la misma. Para este ejemplo se utilizó la política de reemplazo LFU, se eligió un tamaño de *cache* de 25 Kbytes y que los

documentos almacenados en *cache* fueran de Texto, de los subtipos más habituales (en concreto, css, html, js:javascript:x-javascript, plain y xml).

Como se puede observar de las líneas mostradas por pantalla, que aparecen en el Apéndice B, se fue controlando en todo momento que el emulador gestionase correctamente el tamaño disponible de *cache*. También que se comprobaba si los documentos coincidían con el tipo elegido y una vez comprobado esto que también estuviesen dentro de los subtipos seleccionados, indicándose con los correspondientes mensajes.

Así mismo, se introdujeron líneas específicas para comprobar el correcto funcionamiento del emulador ante la llegada de documentos cuyo tamaño excediera del tamaño de *cache* o para el caso en el que el objeto a guardar en *cache* obligaba al reemplazo de todos los documentos hasta ese momento guardados.

También se hace un exhaustivo control del tamaño de la lista y la tabla Hash conforme se producen modificaciones en las mismas. La introducción y el borrado de documentos en *cache* son procesos críticos y básicos en el funcionamiento del emulador. Si bien el borrado se hace siempre por el principio de la lista, se tiene que verificar que sólo son eliminados los documentos suficientes para que el nuevo sea almacenado. En cuanto a la inserción, cada política de reemplazo determinará la prioridad de cada documento y por tanto, dónde deberá ser insertado, teniéndose que comprobar que dicha inserción se hace en el punto correcto de la lista. De ahí, que se presenten mensajes por pantalla donde se especifica el identificador del objeto a almacenar, su peso y el punto en el que se insertará en la lista (lista, cuya primera posición de inserción es cero).

Al finalizar la emulación, también se muestra en pantalla el contenido final de la lista y la tabla Hash. En el caso de la lista, esto nos permite conocer el estado final de la misma y comprobar que los objetos que han permanecido en ella son los correctos y están correctamente ordenados. En el caso de la tabla Hash, se puede comprobar cómo los documentos existentes coinciden con los de la lista, aunque por la forma en la que dicha tabla lleva a cabo el almacenamiento de los mismo, no tienen porqué coincidir en su ordenación.

Para el ejemplo mostrado en el Apéndice B, se aprecia además el efecto de la *cache pollution* típico de LFU, porque el objeto con identificador 100, que fue popular al principio, se mantiene en *cache* hasta el final de la ejecución, aún cuando no vuelva a ser solicitado de nuevo, al haber ganado suficiente prioridad al principio.

Al mismo tiempo que se obtenían estos resultados de la emulación, se realizó en papel una simulación del comportamiento que debía seguir el emulador para esa traza y con esos mismos datos, con el fin de comprobar ambos resultados (de ahí que no se tomaran un mayor número de líneas de la traza, pues las comprobaciones se hubieran prolongado en exceso).

El proceso descrito fue empleado para comprobar el correcto funcionamiento de todas las políticas de reemplazo y una vez hecho, se supuso que el comportamiento seguiría siendo correcto cuando el número de líneas a procesar fuese mayor.

Una vez comprobado el correcto funcionamiento del emulador para las distintas políticas de reemplazo, se utilizaron programas como OptimizeIt de Borland Jbuilder 2005 y JProfiler 4.0.2 para llevar a cabo un análisis del comportamiento del emulador. El análisis se centró en lo relativo a uso de memoria y posibles pérdidas de la misma, y consumo de tiempo de CPU, sin observar comportamientos anómalos.

Con el fin de que las emulaciones se pudiesen finalizar en un menor tiempo, se intentó pasar el código fuente Java a código nativo máquina, a sabiendas de que ello suponía una pérdida en la versatilidad del emulador en lo relativo a su uso en cualquier plataforma (una de sus características al estar programado en Java). Para ello se emplearon los programas Excelsior JET 3.7 y con Jsmooth 0.9.7, pero no se obtuvo mejora en los tiempos de emulación.

3.3.4. RECOMENDACIONES PARA EL USO DEL EMULADOR

El emulador de *cache proxy* diseñado puede ser instalado en cualquier directorio. Dicho emulador se encuentra en la carpeta “Emulcacheproxy” del disco adjunto a la memoria del Proyecto Fin de Carrera y se recomienda su instalación en una carpeta del mismo nombre en el directorio raíz del disco duro C del sistema en el que se vaya a usar, para mantener todos los archivos asociados a la aplicación en un mismo directorio y facilitar su manejo. Para llevar a cabo dicha instalación tan sólo es necesario copiar el contenido de la carpeta antes mencionada al lugar donde se desea instalar la aplicación.

La aplicación implementada requiere de la existencia del directorio `c:\emulcacheproxy\resultados` y dentro del mismo, de las carpetas con el nombre del tipo de documentos sobre los que se va a trabajar, pues al finalizar la emulación, el archivo de resultados en el que se incluye la información relativa al *hit rate* y *byte hit rate* así como otros datos estadísticos de interés, se guardará en dicho directorio. De no existir tales directorios los archivos de resultados no se guardarán.

Es recomendable también, que se cree dentro del directorio “emulcacheproxy”, la carpeta “Trazas” y en ella se guarden las muestras de tráfico que se vayan a utilizar para hacer las emulaciones. Esto hace más cómodo el uso de la aplicación, si bien no es absolutamente necesario ya que dichas muestras de tráfico pueden ser seleccionadas de cualquier otra carpeta del sistema.

Para llevar a cabo la ejecución de la aplicación es necesario haber instalado previamente el entorno de ejecución Java apropiado para la máquina concreta en la que se desee utilizar el emulador. Dicho entorno se puede obtener gratuitamente de la página web de Sun Microsystems [SUN].

La aplicación ha sido probada usando la versión de la máquina virtual Java 1.5.0_03. Un aspecto importante a tener en cuenta a la hora de hacer uso del emulador, es la necesidad de que se le indique a la máquina virtual Java, que ha de usar una cantidad de memoria suficiente para poder llevar a cabo la emulación, pues de lo contrario, una vez alcanzado el máximo de memoria establecido por defecto, se detendría la ejecución y aparecería un mensaje de error en pantalla.

Por defecto la maquina virtual de Java utiliza 64 MBytes de memoria. Esta cantidad puede modificarse mediante los parámetros de configuración de dicha máquina virtual. Para ello, el formato a seguir sería: `-msXXXm -mxYYYm`. Con la primera indicación se establece la cantidad mínima de memoria que empleará la máquina virtual, que sería XXX y dicho número estaría expresado en Megabytes, como expresa la “m” minúscula tras el número. La segunda indicación constituye la cantidad máxima de memoria que puede usar, también en Megabytes.

Un ejemplo de configuración del tamaño de memoria a emplear por la máquina virtual Java podría ser: `-ms128m -mx512m`. Con esto se le indica a la máquina virtual que comience usando una memoria de 128 MB y una vez supere dicha necesidad de memoria, automáticamente la irá incrementando, teniendo como cantidad límite superior para su uso los 512 MB. La cantidad superior se suele fijar a la memoria RAM (*Random Access Memory*) disponible en la máquina, aunque no tiene porqué ser así y queda a la elección del usuario según las necesidades de éste.

A la hora de ejecutar la aplicación, se recomienda crear un archivo `.bat` (archivo por lotes MS-DOS) en el que se incluyan los comandos necesarios para ejecutar la aplicación y los parámetros que deba utilizar la máquina virtual Java. Un archivo de este tipo, a modo de ejemplo, se incluye en el disco adjunto a la memoria del Proyecto Fin de Carrera.

CAPÍTULO 4: Estudio comparativo del rendimiento según el tipo de documento

En este capítulo se describe la metodología seguida a la hora de realizar el estudio objeto de este Proyecto Fin de Carrera, así como de las métricas utilizadas. También se proporcionará una caracterización de las trazas empleadas y para finalizar se analizarán por separado los resultados obtenidos para cada uno de los tipos de documentos estudiados.

4.1. METODOLOGÍA DE ESTUDIO Y MÉTRICAS EMPLEADAS

A la hora de hacer un estudio de la efectividad en cuanto al uso de una *cache*, existe una gran dependencia en los resultados obtenidos respecto al tamaño de *cache* y a la política de reemplazo empleada. Existen tres métodos principales a la hora de comprobar dicho comportamiento:

- Emulaciones empleando trazas: se registran las peticiones de una serie de usuarios en un archivo de traza, que se usa como entrada a dicha emulación basada en trazas. De este modo, el tiempo de simulación pasa más rápido que el tiempo real.
- Uso de trazas sintéticas: se puede generar trazas de forma sintética aunque se corre el riesgo de que ciertas características de las trazas WWW no sean tenidas en cuenta de la manera correcta.
- Implementación real: se lleva a cabo la evaluación observando a un servidor real que emplea la estrategia de almacenamiento en *cache* bajo estudio. Este será el método que más recursos consuma y no resulta el más conveniente para comparar el comportamiento de distintas configuraciones de *cache* pues los usuarios no repiten nunca las peticiones de la misma forma.

Una vez seleccionado el método de emulaciones basado en trazas como el más apropiado para hacer el estudio, se comenzó por determinar cuál era el tamaño de *cache* infinito para cada uno de los tipos de documentos bajo estudio, relativo a las trazas que se iban a usar.

Para obtenerlo, se realizó una simulación para cada uno de los tipos de documentos con un tamaño de *cache* tal que pudiese contener, hasta el final de la emulación, a cualquier documento que en algún momento fuese accedido, de cualquiera de los subtipos pertenecientes al tipo escogido, fuese cual fuese su tamaño. De esta forma, al final de la emulación se tendría el tamaño mínimo de *cache* que contendría cualquier petición asociada a un determinado tipo de documento así como el límite superior en cuanto a tasa de acierto y tasa de acierto de byte que se podría alcanzar.

En las posteriores emulaciones se utilizaron porcentajes de dicho valor de memoria (concretamente 2, 5, 10, 30 y 50%), con lo que se producirían reemplazos y así se pudo estudiar la influencia de éstos en el rendimiento, medido según las correspondientes métricas, en relación con el tamaño de *cache* y la política de reemplazo seleccionada.

En cuanto a las métricas empleadas para evaluar el comportamiento de los distintos algoritmos que gestionan la *cache* y de esta forma medir la efectividad de su uso, se usaron la tasa de acierto y la tasa de acierto de byte, que son las dos más comunes.

Se define como tasa de acierto o *Hit Rate* (HR), al número de peticiones que producen un acierto en *cache* dividido entre el total de peticiones de ese tipo de documentos. De igual forma, se define la tasa de acierto de byte o *Byte Hit Rate* (BHR) como el número de bytes que han sido transferidos desde la *cache* respecto del total de bytes pedidos, de ese tipo de documento concreto.

Con la última métrica se puede tener una idea del ancho de banda o latencia ahorrada, a diferencia de usar el HR que tan sólo indica el número de aciertos conseguidos. El BHR suele ser menor que el HR porque los objetos pequeños como imágenes o páginas HTML tienden a tener más aciertos en *cache* que los grandes como audio o archivos *PostScript*.

Los aciertos en *cache* para documentos de gran tamaño contribuyen más al BHR que los documentos pequeños. Por el contrario tener más documentos pequeños almacenados en *cache* contribuye a obtener un mayor HR.

Se puede conseguir un mayor HR con una *cache* de mayor tamaño, pero la relación no es lineal. Así, duplicar el tamaño de *cache* no implica duplicar el HR, de hecho existe un límite superior para el HR que no se puede superar, aunque se utilice una *cache* de tamaño infinito [Duska'97]. También se ha observado que cuantos más usuarios hacen uso de una misma *cache proxy*, más se incrementa el HR. El motivo principal de este hecho se encuentra en la relación que existe entre las peticiones Web y la popularidad de los documentos solicitados [Breslau'99].

Además, para la familia de algoritmos *Greedy Dual* evaluada, se ha hecho una distinción adicional. Se han usado dos definiciones de la función de coste distintas. La primera de ellas define el coste constante (el denominado modelo de coste constante), igual a 1, de forma que a todos los documentos se les asigna un mismo coste a la hora de traerse de un servidor remoto. La segunda por el contrario, el llamado modelo de coste de paquetes (*packets*), asume que el coste de traer un documento es proporcional a su tamaño. En este caso se calcula el coste según la expresión $2 + \text{tamaño}/1460$, siendo 1460 el tamaño en bytes de la MTU (*Maximum Transfer Unit*) empleada en TCP. De esta forma, con la

función de coste se tiene en cuenta el número de paquetes enviados y recibidos para satisfacer un fallo en *cache* en el caso que dicho documento no se encuentre en la misma.

A la hora de hacer uso de dichas definiciones con las políticas bajo estudio, se hará referencia a las mismas poniendo entre paréntesis tras el nombre de la política “1” o “*packets*” para indicar que los resultados obtenidos se han conseguido mediante el empleo de la correspondiente definición de función de coste.

Empleando el modelo de coste constante, se trata de conseguir el mayor HR (a costa de tener un bajo BHR) y por tanto minimizar la tasa de fallos en *cache*, mientras que con el modelo de coste de paquetes se busca maximizar el BHR (aunque también mantiene un HR alto, por lo general).

Con el uso del primer modelo, por tanto, se trata de reducir la latencia y con ello, que mejore el rendimiento percibido por los usuarios, pudiéndose dar servicio a un mayor número de peticiones de usuarios desde la *cache* del servidor *proxy*, mientras que con el segundo modelo lo que se busca es reducir el tráfico total soportado por la red al producirse fallos en *cache*.

Ambas métricas son un tanto contradictorias, por lo que es muy difícil para una política de reemplazo conseguir los mejores resultados en ambas simultáneamente.

4.2. CARACTERIZACIÓN DE LAS MUESTRAS DE TRÁFICO

Las muestras de tráfico empleadas para la realización de este estudio sobre la efectividad de las distintas políticas de reemplazo, fueron tomadas de la página de IRCACHE [IRCACHE]. El sistema IRCache consiste en un conjunto de diez servidores *proxy* que emplean almacenamiento en *cache* y que se encuentran distribuidos a lo largo de los Estados Unidos.

Entre los objetivos perseguidos por el proyecto IRCache se encuentran el proporcionar servicios de almacenamiento en *cache*, que operan en base a una estructura jerárquica, tanto a organizaciones como a particulares, servir de banco de pruebas para nuevas versiones del software de gestión de *cache* Squid [SQUID], promover el uso de las *caches* Web en la comunidad de Internet y, el objetivo más interesante, desde el punto de vista de este proyecto, proporcionar archivos con muestras de tráfico relativamente grandes para su uso por parte de investigadores y de otras organizaciones.

Las muestras de tráfico empleadas para la realización de este proyecto, fueron tomadas del NLANR (*National Laboratory for Applied Network Research*) [NLANR], de una *cache proxy* de primer nivel situada en el Research Triangle Park, en Carolina del Norte (Estados Unidos) entre los días 7 y 11 de Junio de 2004.

Originalmente, los archivos que contienen las muestras de tráfico, que en cada una de sus líneas representa una petición de un documento, presentaban los diez campos siguientes:

- Marca de tiempo: hora a la que el *socket* del usuario es cerrado. Tiene formato Unix con resolución de milisegundos.
- Tiempo transcurrido: tiempo transcurrido de la petición. Es el tiempo entre el *accept()* y el *close()* del socket del usuario. Para las conexiones HTTP persistentes, es el tiempo entre la lectura del primer byte de la petición y la escritura del último byte de la respuesta.

- Dirección del usuario: una dirección IP aleatoria que identifica al usuario.
- Etiqueta *Log (Log Tag)* y código HTTP: la etiqueta Log describe cómo fue tratada localmente la petición. El código HTTP es tomado de la primera línea de la cabecera HTTP de respuesta. Las peticiones no-HTTP tendrán un código de respuesta cero.
- Tamaño: número de bytes enviados al usuario.
- Método de la petición: método de petición HTTP (GET, POST, HEAD, ...)
- URL: la URL solicitada.
- Identificador de usuario: será siempre “-“ para el log IRCache
- Información de la jerarquía y nombre del *Host*: descripción de cómo y dónde se obtuvo el documento solicitado.
- Tipo de contenido: El campo *Content-type* procedente de la respuesta HTTP.

Sobre las trazas se llevó a cabo un preprocesado que en un primer momento supuso quedarse sólo con las respuestas a peticiones GET. Las respuestas a peticiones POST no se introdujeron en *cache* al no disponerse de información sobre los tiempos de expiración. Se eliminaron también aquellas entradas que presentaban la cadena “:3128” en la URL, pues corresponden a comunicaciones entre las distintas *caches* de la jerarquía, ya que dichas comunicaciones se hacen a través del puerto 3128. Fueron también excluidos de su almacenamiento en *cache*, los documentos que presentaban las cadenas “.cgi”, “cgi-bin” o “?” en la URL, por corresponder a documentos dinámicos.

De las restantes peticiones, se consideraron sólo las respuestas con códigos de estado HTTP 200 (OK), 203 (*Non Authoritative Information*), 206 (*Partial Content*), 300 (*Multiple Choices*), 301 (*Moved Permanently*), 302 (*Found*) y 304 (*Not Modified*). En el caso de las respuestas 304 se volvió a pedir el documento para que el tamaño del mismo,

incluido en la respuesta fuera el de dicho documento y no el de la petición *If-modified-since*.

En las muestras de tráfico utilizadas, quedaron los campos tiempo, tamaño, identificador, tipo y subtipo de documento. Puesto que en las muestras de tráfico usadas no se disponía de información acerca de los tiempos de expiración, no se pudo realizar el control sobre el funcionamiento de la validación en la *cache*. En el campo identificador, se sustituyó cada URL por un número, que será único para cada una de dichas URL. En cuanto al campo tipo de contenido, éste fue separado en dos campos. El formato del tipo de contenido es “tipo/subtipo” del documento. La primera parte de dicho campo pasó a ser el nuevo campo tipo, en el que a cada tipo de datos se le asignó un número. La codificación asignada a cada tipo de documento se muestra en el Apéndice C. El campo subtipo tendrá también una codificación numérica en base a los distintos subtipos de documentos. La codificación empleada para los distintos subtipos de documentos, también se muestra en el Apéndice C.

En lo relativo a la composición de las muestras de tráfico empleadas, se observó el predominio en cuanto número de peticiones, de las imágenes, con más del 75% del total de las mismas. Si a estas se añaden los documentos de texto y las aplicaciones, los tres tipos conjuntamente, constituyen más del 99% del total de peticiones. En el caso de los bytes totales pedidos, las imágenes con más del 36% del total, las aplicaciones con más del 33% y los documentos de texto con más del 23% son los tipos de documentos predominantes. En la tabla 4.1 se presentan las características de las muestras de tráfico empleadas con más detalle diferenciando entre los distintos tipos de documentos.

Tabla 4.1. Características de las muestras de tráfico según tipo de documento

	Aplicación	Audio	Imágenes	Texto	vídeos
% Total de Peticiones	8.08	0.28	75.54	15.77	0.12
% Total de Bytes Pedidos	33.51	3.49	36.33	23.15	3.19

4.3. COMPARACIÓN DEL RENDIMIENTO PARA CADA UNO DE LOS TIPOS DE DOCUMENTOS

Para llevar a cabo la comparativa del rendimiento de cada una de las políticas de reemplazo estudiadas y de cada uno de los tipos de documentos que se tienen en cuenta, se comenzó realizando la simulación del funcionamiento de la *cache* para los tamaños del 2, 5, 10, 30 y 50% de la *cache* infinita correspondiente a cada tipo de documento, guardando en *cache* todos los subtipos de documentos.

Tras esto se observó el comportamiento si sólo se almacenaban en *cache*, aquellos subtipos que constituían el mayor porcentaje de referencias en la muestras de tráfico empleadas. A dicho subconjunto de subtipos se les hace referencia como los “subtipos más habituales”.

Por último, se estudió, de los subtipos más habituales, cuáles proporcionaban un mayor HR y BHR por separado y se eligió sólo a los que proporcionaban una aportación más significativa constituyendo los llamados “subtipos con mayor HR” y “subtipos con mayor BHR”, respectivamente. Con ellos se trató de observar si se conseguía mejorar la respectiva métrica, guardando tan solo en *cache* esos subtipos de documentos concretos.

Con los resultados obtenidos se han realizado gráficas, para facilitar el análisis de los datos extraídos de las simulaciones. En las gráficas siempre aparecen cinco líneas, cada una correspondiente a una política de reemplazo. Para el caso de GDS, GDSF y GD*, entre paréntesis puede aparecer “1” o “*packets*” pues para estas tres políticas se emplearon dos funciones de coste distintas. En el caso de LRU y LFU no ocurre así pues estas dos políticas no emplean dicha función para determinar la prioridad de los documentos.

En los siguientes apartados se presentan los resultados obtenidos para los tipos de documento aplicación, audio, imágenes, texto y vídeo, que han sido los que se han tenido en cuenta a la hora de realizar el estudio.

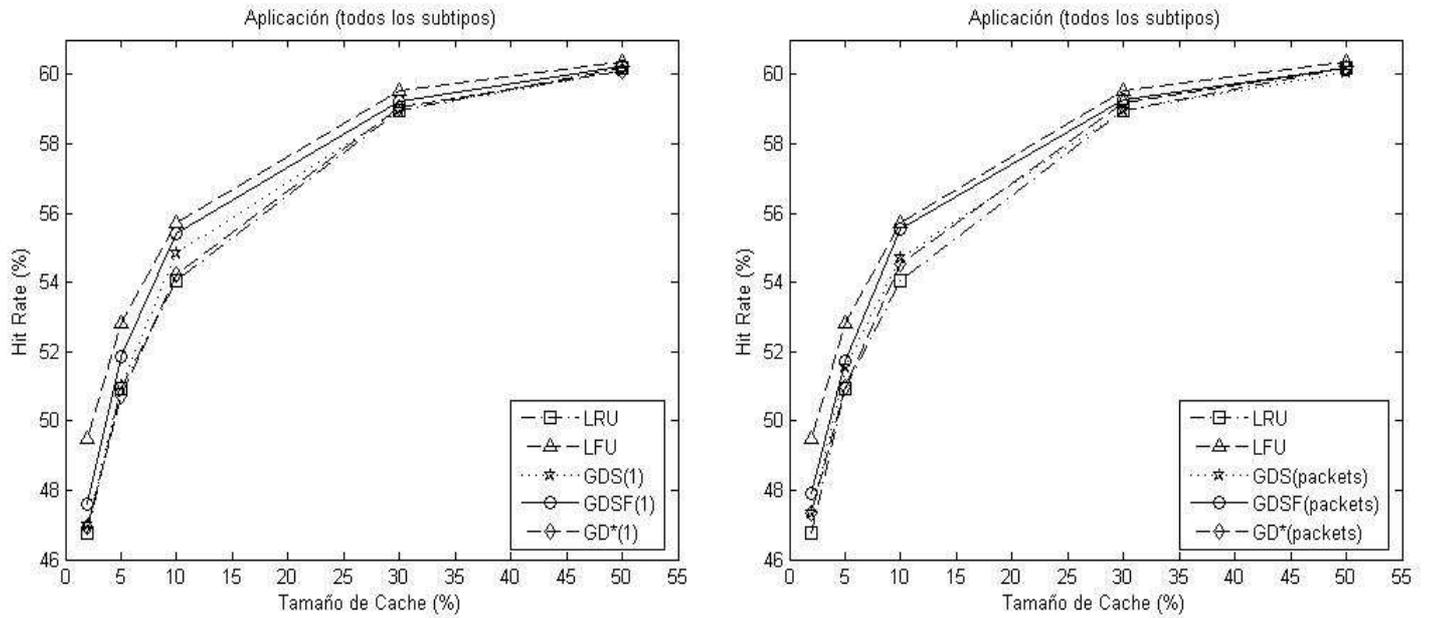
4.3.1. APLICACIONES

4.3.1.1. Resultados para todos los subtipos

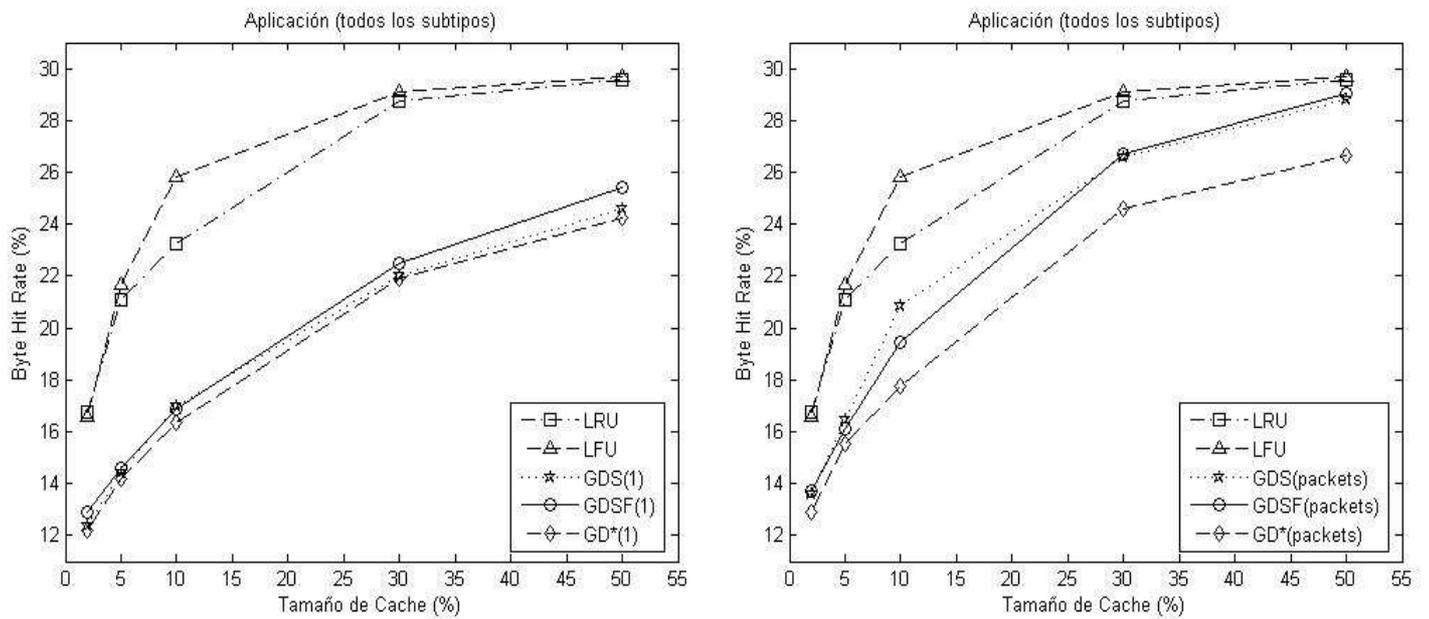
En la figura 4.1. se muestran los resultados obtenidos, en cuanto a HR y BHR, para la inserción en *cache* de todos los subtipos de aplicación, con función de coste 1 y *packets*, respectivamente.

Como se observa en la figura 4.1. (a), aunque todas las políticas presentan un comportamiento similar, LFU es la política que proporciona unos mejores resultados en cuanto a HR seguida de GDSF. A continuación, GD* y GDS proporcionan unos resultados muy parecidos, siendo LRU la que presenta un comportamiento peor. Estos resultados se dan tanto para coste uno como *packets* y son más evidentes cuando el tamaño de la *cache* es menor. Por tanto, las políticas basadas en frecuencia son las que mejores resultados proporcionan en cuanto a HR (para aplicaciones).

En cuanto al BHR, como se aprecia en la figura 4.1. (b) las políticas LFU y LRU son las que proporcionan un mejor rendimiento respecto a las políticas de la familia Greedy. La diferencia de rendimiento es aún mayor si la función de coste empleada por las políticas GDS, GDSF y GD* es uno pues, como cabía esperar, los resultados obtenidos para el BHR con la función de coste *packets* son netamente mejores que los que se obtienen con coste uno a la hora de estudiar el BHR. Incluso se observa que, para tamaños de *cache* pequeños, el HR obtenido con coste *packets* es algo mejor que el obtenido con coste uno, aunque por lo general empleando la función de coste uno se consigue unos resultados mejores en cuanto a HR.



(a) HR para todos los subtipos de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para todos los subtipos de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.1. HR (a) y BHR (b) para todos los subtipos de Aplicación

4.3.1.2. Resultados para los subtipos más habituales

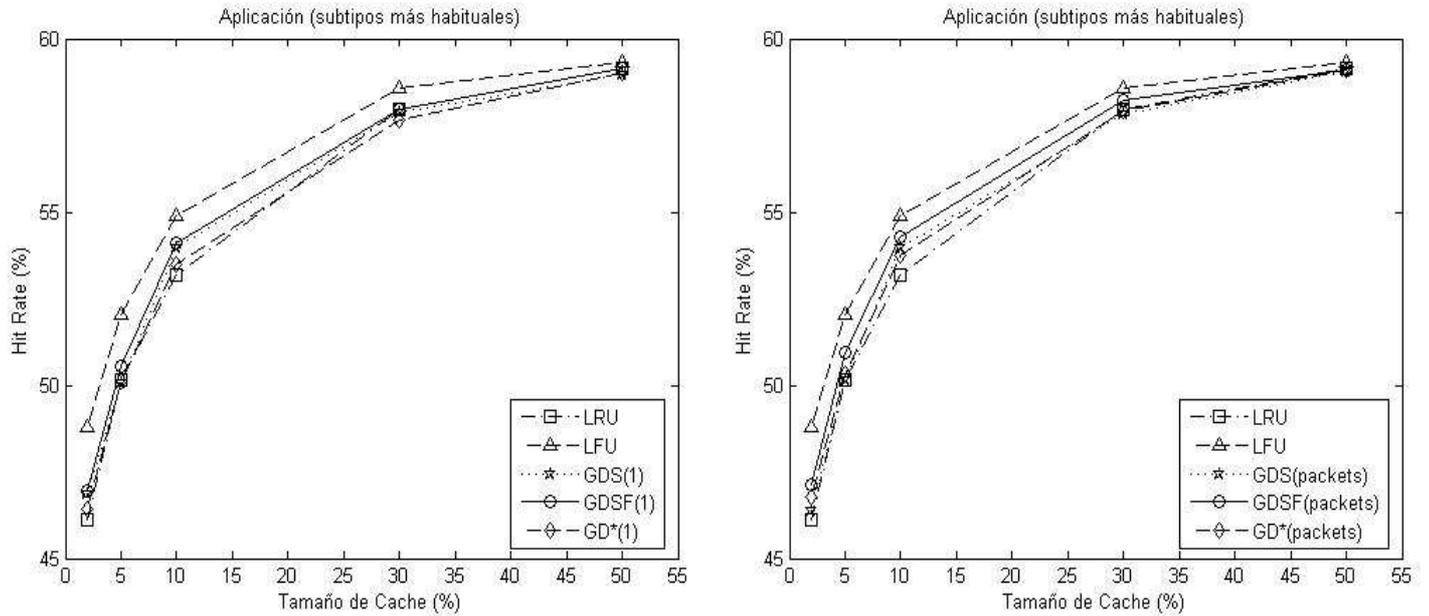
A la hora de hacer las emulaciones para los subtipos más habituales, en primer lugar, se observó qué porcentaje del total de documentos de cada subtipo se daba en las muestras de tráfico empleadas y qué porcentaje de bytes estaba asociado a cada subtipo de documento. En la tabla 4.2. se presentan los subtipos elegidos y los porcentajes por los que fueron seleccionados.

Tabla 4.2. Características de los subtipos “más habituales” de aplicación

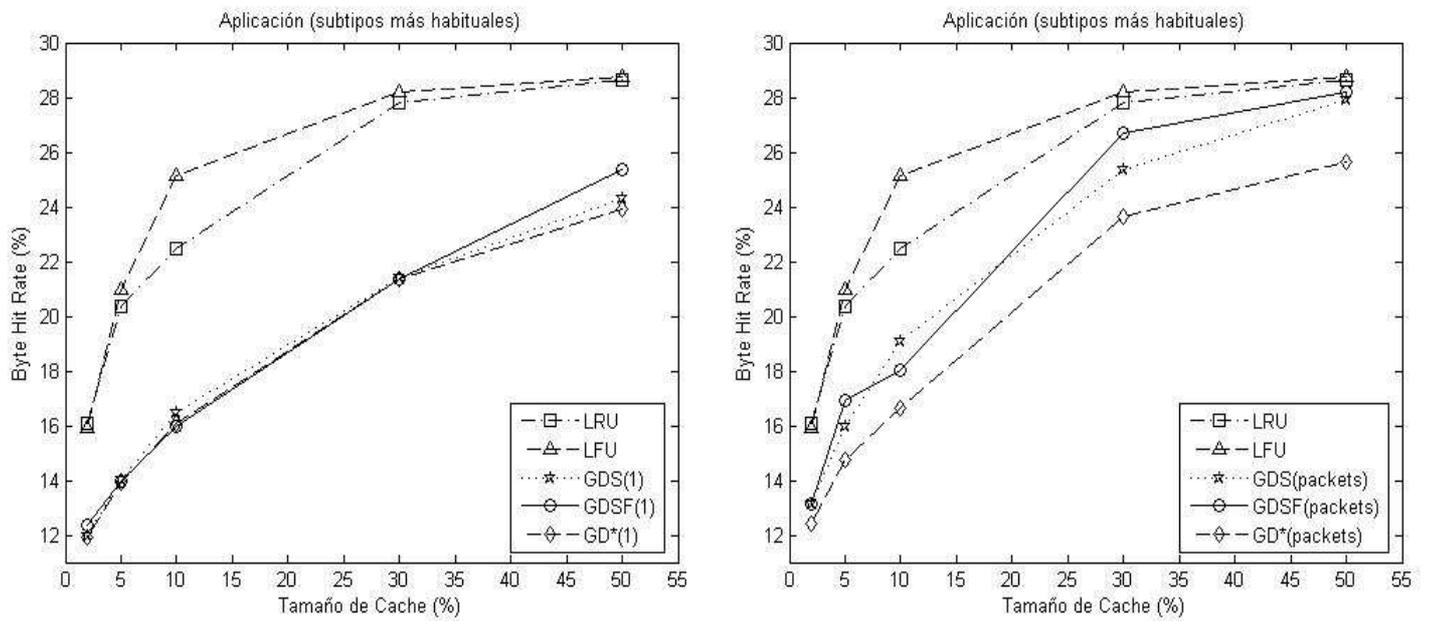
Subtipos de aplicación	% documentos	% bytes
javascript : x-javascript : x-javascríp	73.11	11.79
octet stream : octet-stream : x-20octet-stream	9.94	28.64
pdf	1.93	3.81
x-flash : x-shockwave : x-shockwave-flash : x-shockwave-flash2-preview	7.76	11.25
x-mms-framed	0.62	22.29
x-msdownlad	0.45	5.48
x-tar	0.02	5.95
zip : x-zip : x-zip-compressed	3.17	7.66
Total	97.03	96.9

Una vez seleccionados estos subtipos de aplicación como los “más habituales”, se realizaron las correspondientes emulaciones, seleccionando que tan sólo se guardaran en *cache* los subtipos de la tabla 4.2. Con esta selección, de los resultados obtenidos se realizaron las gráficas que se presentan en la figura 4.2.

Al igual que ocurría cuando se seleccionaban todos los subtipos para ser guardados en *cache*, LFU es la política que proporciona unos mejores resultados en cuanto a HR seguida de GDSF, GD*, GDS y LRU. Estos resultados se pueden observar en la figura 4.2. (a) y se dan tanto para coste uno como *packets* siendo más evidentes cuando el tamaño de la *cache* es menor. Los resultados obtenidos son en torno al 1% peor que para el caso en el que se guardaban todos los subtipos de aplicación.



(a) HR para los subtipos más habituales de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos más habituales de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.2. HR (a) y BHR (b) para los subtipos más habituales de Aplicación

En la figura 4.2. (b) se observan los resultados obtenidos para el BHR de los subtipos más habituales. Las políticas LFU y LRU siguen proporcionando un mejor rendimiento, como ya ocurriera al guardarse todos los subtipos, respecto a las políticas de la familia Greedy. Si se emplea como función de coste *packets*, los resultados que proporcionan GDS, GDSF y GD* son mejores pero siguen siendo notoriamente peores para tamaños de *cache* pequeños, tan sólo aproximándose al rendimiento de LFU y LRU para tamaños grandes de *cache*.

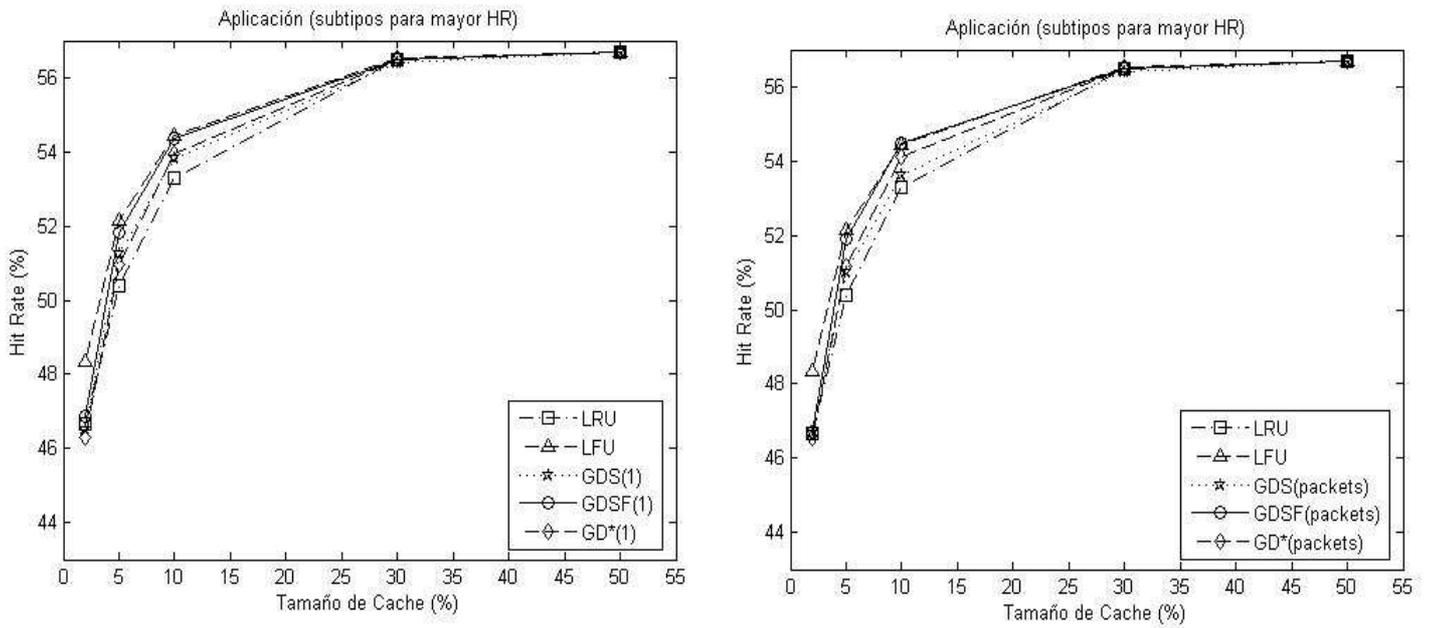
4.3.1.3. Resultados para subtipos con mayor HR y BHR

Con el fin de buscar qué subtipos de los “más habituales” eran los que proporcionaban un mayor HR y BHR, se realizaron una serie de emulaciones en las que, en cada una de dichas simulaciones, tan sólo se guardó en *cache* un subtipo de documento (de los subtipos más habituales de aplicación, aunque este procedimiento se aplicaría también a los otros tipos de documentos). Los resultados obtenidos tras realizar dichas emulaciones se presentan en la tabla 4.3.

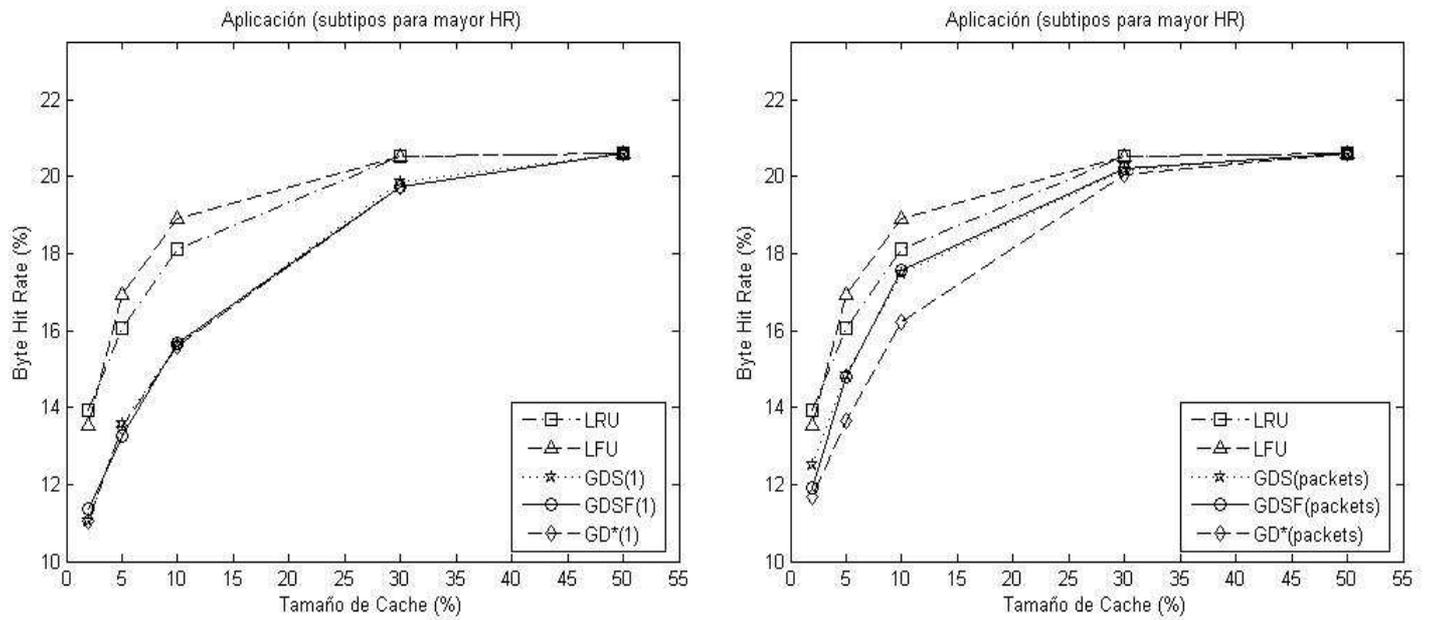
Tabla 4.3. Aportación al HR y BHR de los subtipos “más habituales” de aplicación

Subtipos de aplicación	% HR	% BHR
javascript : x-javascript : x-javascríp	47.75	8.1
octet stream : octet-stream : x-20octet-stream	6.27	9.89
pdf	0.43	0.48
x-flash : x-shockwave : x-shockwave-flash : x-shockwave-flash2-preview	3	2.66
x-mms-framed	0.2	5.16
x-msdownlad	0.39	1.8
x-tar	0.09	1.86
zip : x-zip : x-zip-compressed	2.55	0.49

A tenor de los resultados obtenidos, se consideró que los subtipos *javascript*, *octet stream* y *x-flash* eran los más apropiados para conseguir un mejor HR y fueron los elegidos para tratar de conseguir “mayor HR” mientras que *javascript*, *octet stream* y *x-mms-framed* eran los que debían usarse para buscar “mayor BHR”.



(a) HR para los subtipos para mayor HR de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos para mayor HR de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.3. HR (a) y BHR (b) para los subtipos para mayor HR de Aplicación

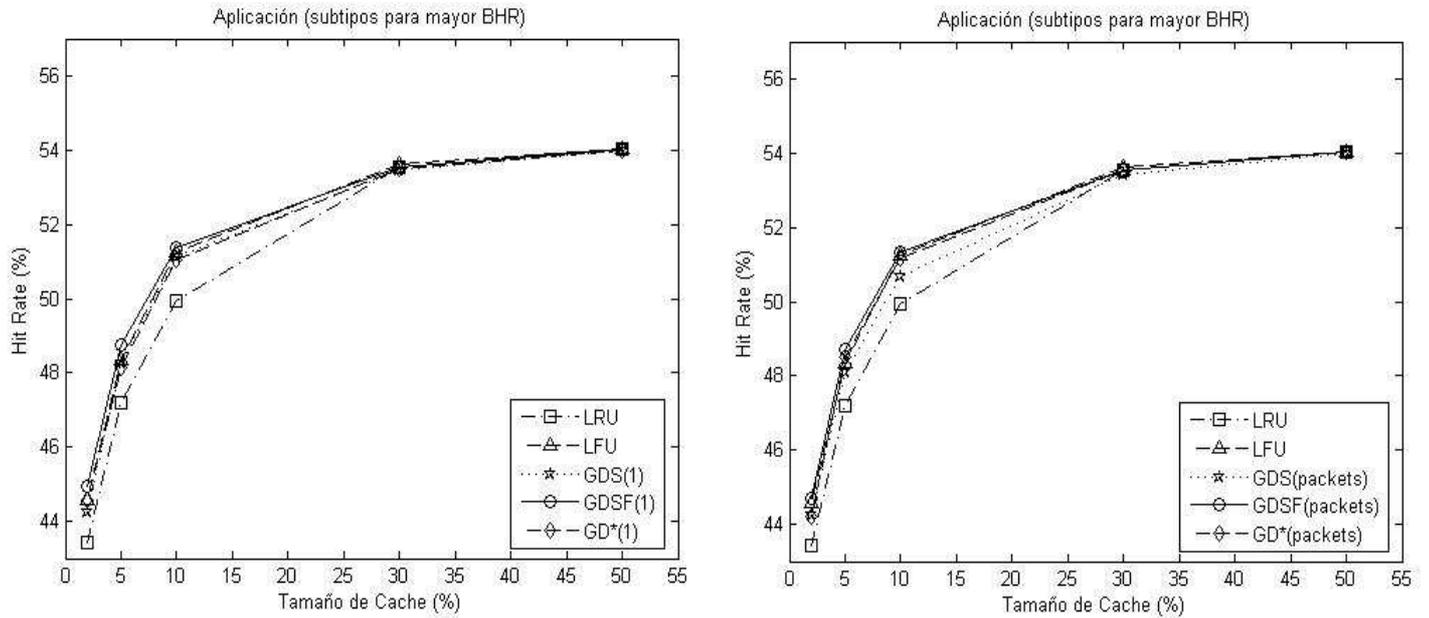
En la figura 4.3 se presentan los resultados de la emulación de los subtipos “para mayor HR”. En la figura 4.3 (a) se presenta el comportamiento en cuanto al HR, tanto para coste uno como *packets*, observándose unos resultados algo peores que para el caso de los “subtipos más habituales” a excepción de la política LRU que para los tamaños de *cache* más pequeños (entre el 2 y el 10%) sí que proporciona mejores resultados que para los “subtipos más habituales”.

La tendencia mostrada en los resultados “para mayor HR” es la misma observada en los casos anteriores aunque para los tamaños de *cache* del 30 y 50% el rendimiento es prácticamente idéntico para todas las políticas, por lo que siempre sería preferible el uso de las políticas más simples.

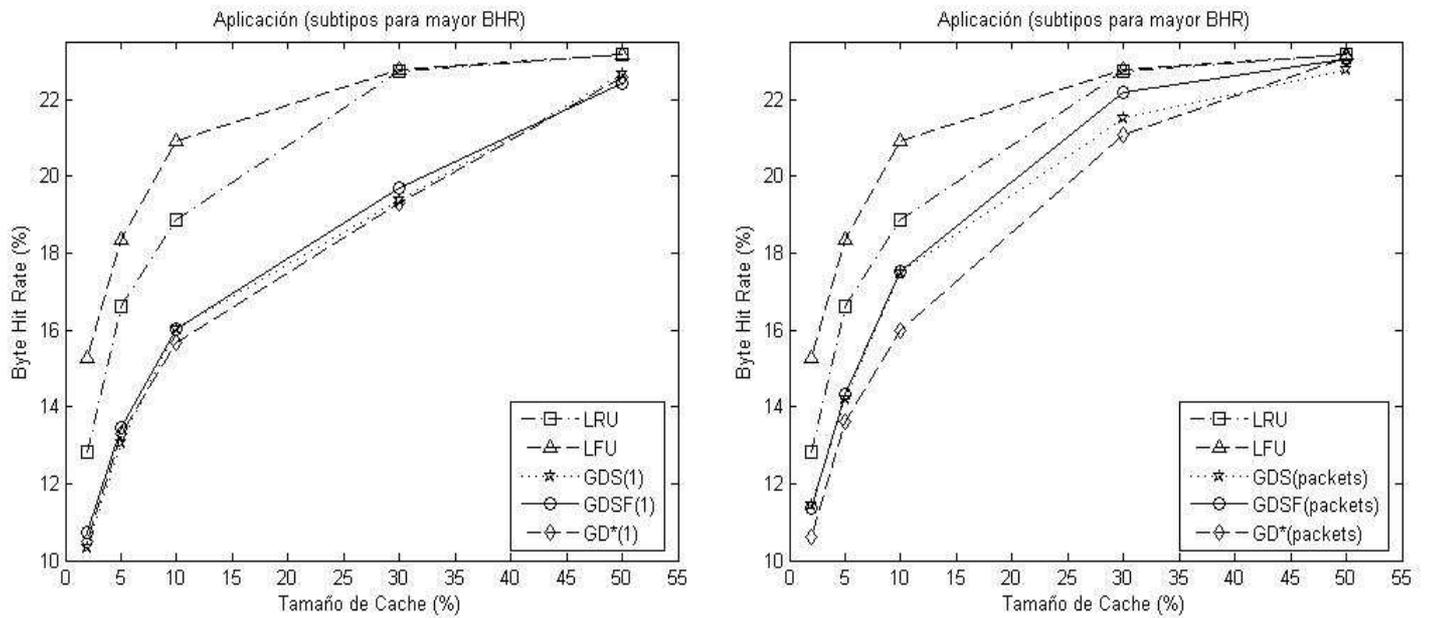
En cuanto al BHR que se obtiene para los subtipos “para mayor HR”, se observa que es el peor resultado hasta ahora, como era previsible ya que con estos subtipos lo que se buscaba era una posible mejora en el HR.

Por su parte, en la figura 4.4, aparecen los resultados obtenidos “para mayor BHR”. En lo referente al HR, mostrado en la figura 4.4. (a), se observa un leve empeoramiento respecto a los resultados obtenidos “para mayor HR”. A partir de tamaño 30% de *cache*, todas las políticas ofrecen el mismo rendimiento.

Por el contrario, los resultados en cuanto a BHR obtenidos son bastante mejores que para los subtipos “para mayor HR”, lo que verifica que el uso de la función de coste *packets*, sin dar los mejores resultados para ambas métricas, sí que suele ser la mejor en cuanto a BHR y suele ser levemente peor respecto al HR que usando la función de coste uno. La política LFU sigue siendo la que proporciona unos mejores resultados seguida de LRU y de las Greedy. Estas últimas ofrecen un mejor rendimiento utilizando como función de coste *packets*.



(a) HR para los subtipos para mayor BHR de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos para mayor BHR de Aplicación con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.4. HR (a) y BHR (b) para los subtipos para mayor BHR de Aplicación

Por tanto, se observa que para los documentos de tipo aplicación, los mejores resultados se dan usando como política de reemplazo LFU y cuando se guardan en *cache* todos los subtipos de documentos, consiguiéndose unos resultados levemente peores en el caso de guardar sólo los subtipos más habituales. Reducir aún más los subtipos a introducir en *cache* no permite mejorar los resultados.

4.3.2. AUDIO

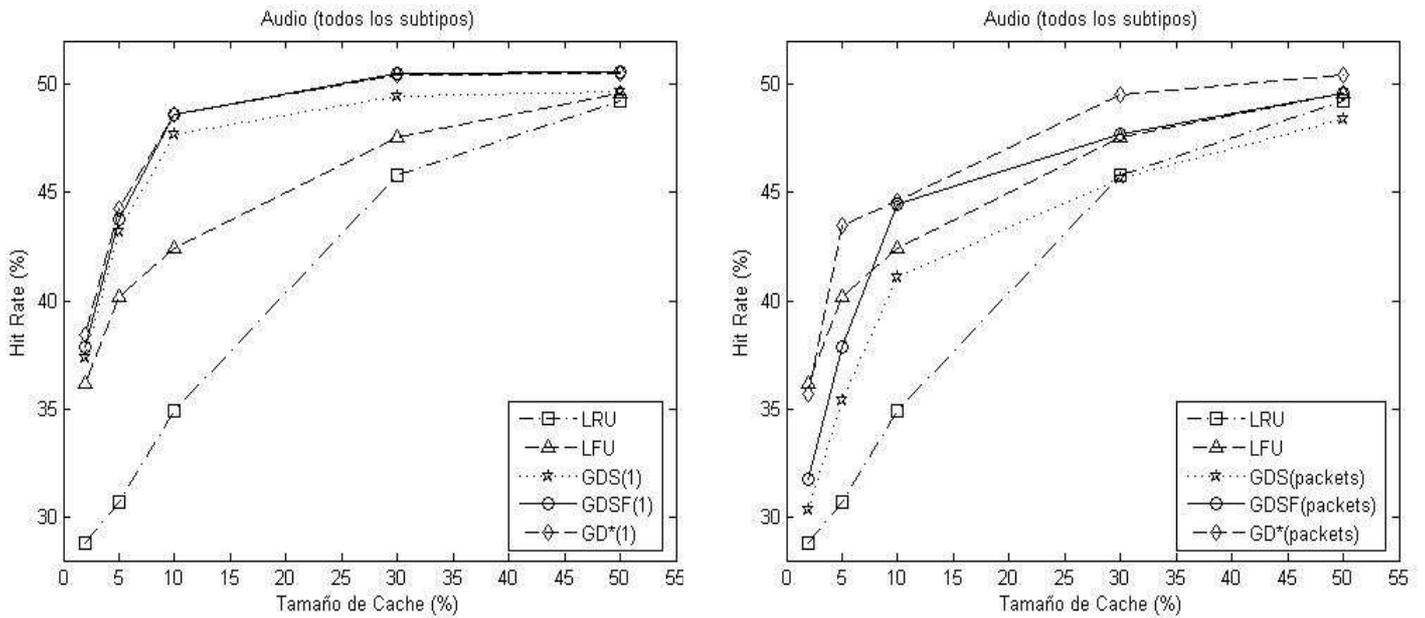
4.3.2.1. Resultados para todos los subtipos

En cuanto a los documentos de audio, se comenzó realizando el estudio del comportamiento de la *cache proxy* cuando se guardaban en la misma todos los subtipos de audio. En la figura 4.5. se muestran los resultados de dichas emulaciones, tanto para el HR como para el BHR.

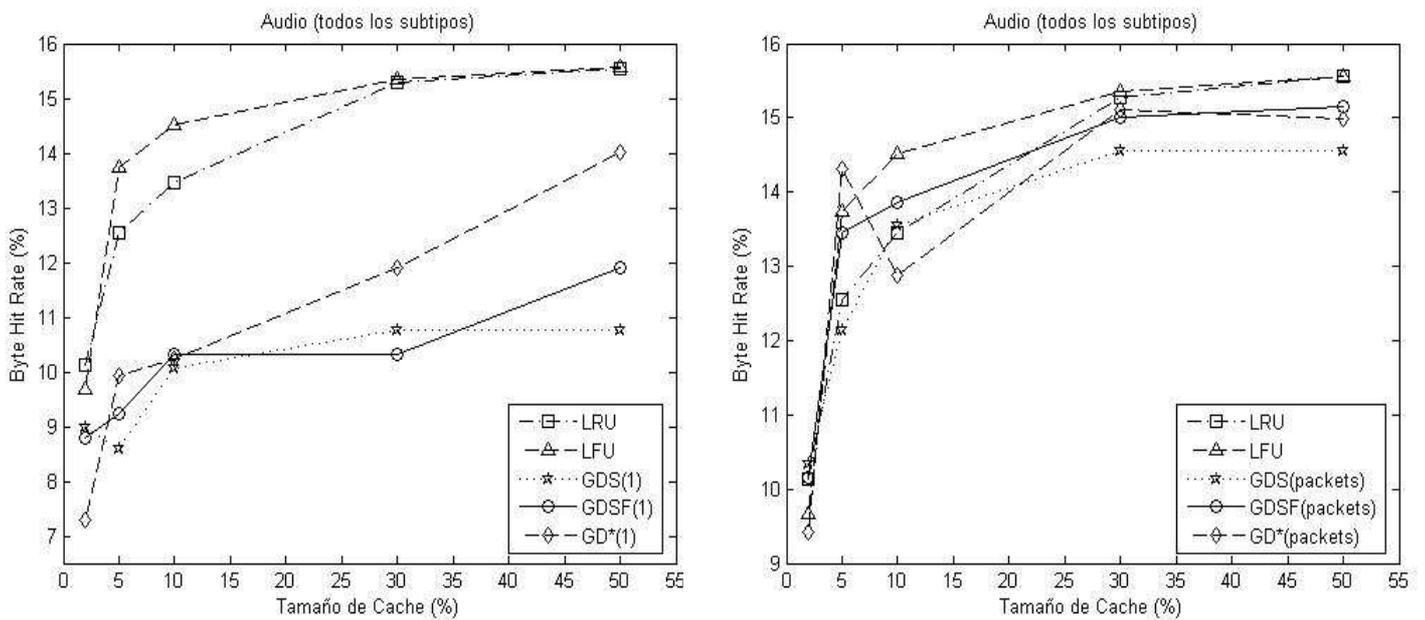
En la figura 4.5. (a), aparecen los resultados obtenidos para el HR cuando se guardan en *cache* todos los subtipos de audio. GD*(1) y GDSF*(1) son las políticas que proporcionan unos mejores resultados en cuanto a HR seguidas de GDS(1), LFU y LRU que es la que tiene un peor comportamiento.

Por el contrario, si se emplea como función de coste *packets*, las políticas de la familia Greedy presentan un peor rendimiento en cuanto al HR y se aproximan a los valores proporcionados por LFU, aunque GD*(*packets*) y GDSF(*packets*) seguirían siendo las mejores opciones. LRU se seguirá manteniendo como la política menos eficaz para el almacenamiento en *cache* de documentos de vídeo, sin distinción de subtipos.

En cuanto al BHR, en la figura 4.5. (b) las políticas LFU y LRU son las que proporcionan un mejor rendimiento respecto a las políticas de la familia Greedy para función de coste uno. Si la función de coste empleada por las políticas Greedy es *packets*, la diferencia de rendimiento es menor, pues dicha función de coste precisamente trata de mejorar el BHR.



(a) HR para todos los subtipos de Audio con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para todos los subtipos de Audio con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.5. HR (a) y BHR (b) para todos los subtipos de Audio

Hay que resaltar que en la figura 4.5 (b), se puede observar para GDS(1) y para GD*(*packets*), entre otros, que existe una reducción del BHR al aumentar el tamaño de *cache*. En el caso de GD*(*packets*) la reducción se produce al pasar del 5 al 10% del tamaño de *cache*. Este hecho, en un principio incongruente, tiene su explicación haciendo uso de la información adicional que se extrae de las emulaciones y que queda recogida en los correspondientes archivos una vez finalizan estas.

Analizando dichos archivos, se comprobó que para GD*(*packets*), con un tamaño del 5%, existía un documento que no se guardaba en *cache* por exceder el tamaño de la misma, mientras que para el tamaño del 10%, sí que se guardaba dicho documento. Esto motiva la eliminación de *cache* de gran parte de los documentos almacenados y es lo que origina la reducción en el BHR. Para tamaños de *cache* superiores, este hecho no resulta tan determinante, de ahí que para tamaños de *cache* mayores, los resultados que se obtienen sí se encuentren dentro de lo esperado.

La explicación de este hecho, es extensible al resto de casos, si bien puede que el número de documentos involucrados sea distinto, como es el caso de GDS(1), para el cual, al pasarse de un tamaño de *cache* del 2 al 5%, pasa de no guardarse en *cache* cuatro objetos por exceder el tamaño de la *cache*, a no guardarse sólo uno. Sin embargo la explicación aplicable a ambos casos, es la misma.

4.3.2.2. Resultados para los subtipos más habituales

Respecto a los subtipos más habituales de audio, se observaron los porcentajes de documentos de cada subtipo y de bytes de los mismos, respecto del total de documentos de audio. Con estos datos, se eligieron aquellos subtipos que pudiesen considerarse como los más habituales, que son los presentados en la tabla 4.4.

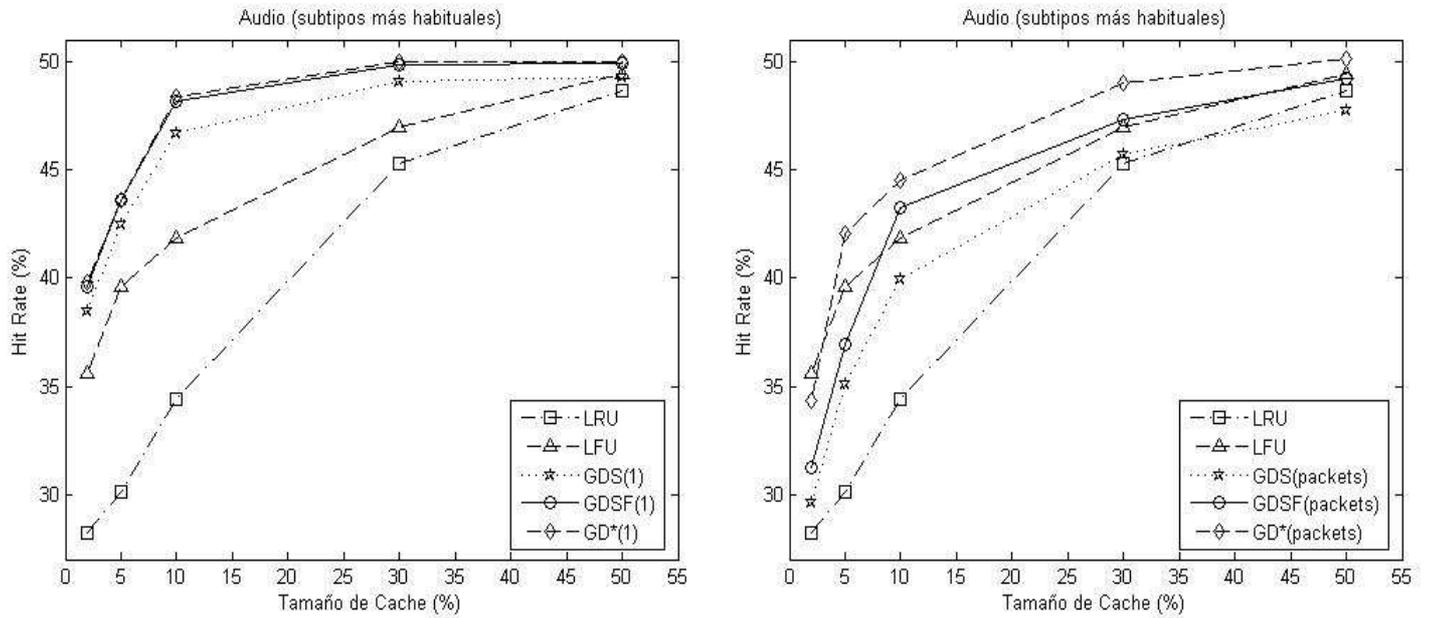
Tabla 4.4. Características de los subtipos “más habituales” de audio

Subtipos de audio	% documentos	% bytes
Basic	60.32	3.41
midi : mid : x-mid : x-midi	8.55	1.56
mpeg : x-mpeg	10.03	76.23
wav : x-wav	15.45	8.92
x-pin-realaudio : x-pn-realaudio : x-realaudio	2.68	8.77
Total	97.04	98.91

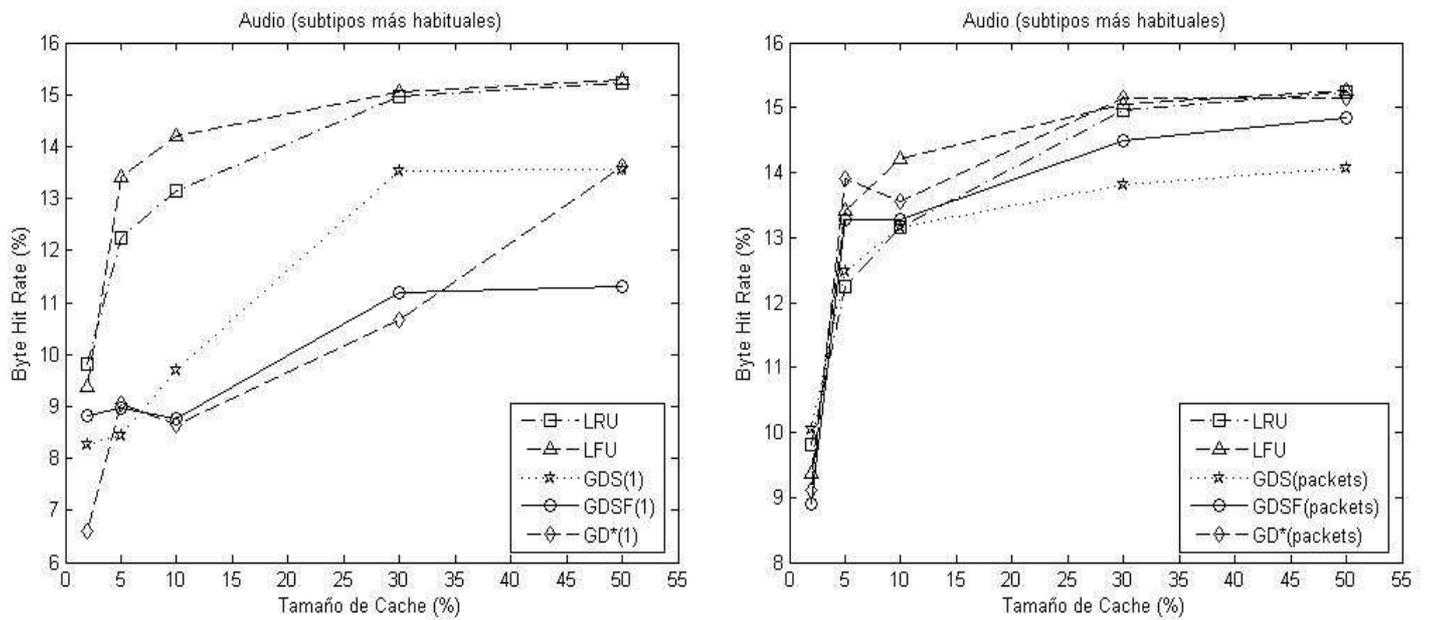
Con estos subtipos de audio elegidos como los “más habituales”, se realizaron las correspondientes emulaciones, seleccionando que tan sólo se guardaran en *cache* los subtipos indicados en la tabla 4.4. Los resultados obtenidos se muestran en la figura 4.6.

En la figura 4.6 (a) se aprecia el comportamiento de las distintas políticas respecto al HR. Como se puede comprobar, el comportamiento sigue el mismo patrón observado (con GD* y GDSF como las políticas más eficientes) para el caso en el que se guardaban en *cache* todos los subtipos de audio, si bien los resultados obtenidos son algo peores, aunque en menos del 1%, tanto para la función de coste uno como *packets*.

El rendimiento demostrado por la *cache proxy* para los subtipos más habituales respecto al BHR se presenta en la figura 4.6. (b). El BHR obtenido con las políticas de la familia Greedy con función de coste uno, es peor que el que se obtenía cuando se guardaban todos los subtipos de audio, salvo para GDSF(1) para los tamaños más grandes de *cache*. Con el uso de la función de coste *packets*, el rendimiento es levemente inferior, excepto para GD*(*packets*) que permite conseguir un mayor BHR sobre todo para los tamaños mayores del 10% de *cache*. LFU seguida de LRU se siguen mostrando como las políticas más apropiadas para conseguir el mejor BHR.



(a) HR para los subtipos más habituales de Audio con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos más habituales de Audio con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.6. HR (a) y BHR (b) para los subtipos más habituales de Audio

4.3.2.3. Resultados para subtipos con mayor HR y BHR

De entre los subtipos “más habituales” se estudió cuáles proporcionaban un mayor HR y BHR siguiendo el procedimiento ya explicado en el apartado anterior, y se obtuvieron los resultados presentados en la tabla 4.5.

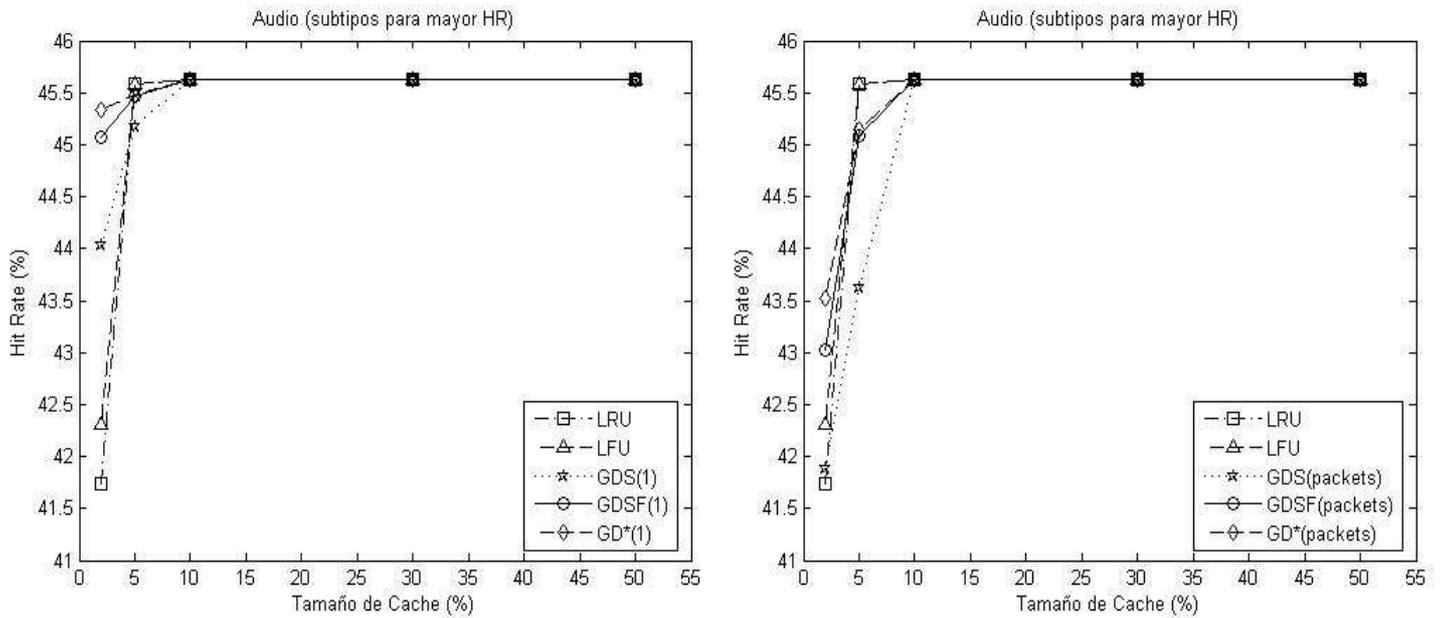
Tabla 4.5. Aportación al HR y BHR de los subtipos “más habituales” de audio

Subtipos de audio	% documentos	% bytes
basic	34.4	1.6
midi : mid : x-mid : x-midi	1.08	0.23
mpeg : x-mpeg	3.12	8.11
wav : x-wav	11.22	4.45
x-pin-realaudio : x-pn-realaudio : x-realaudio	0.4	1.12

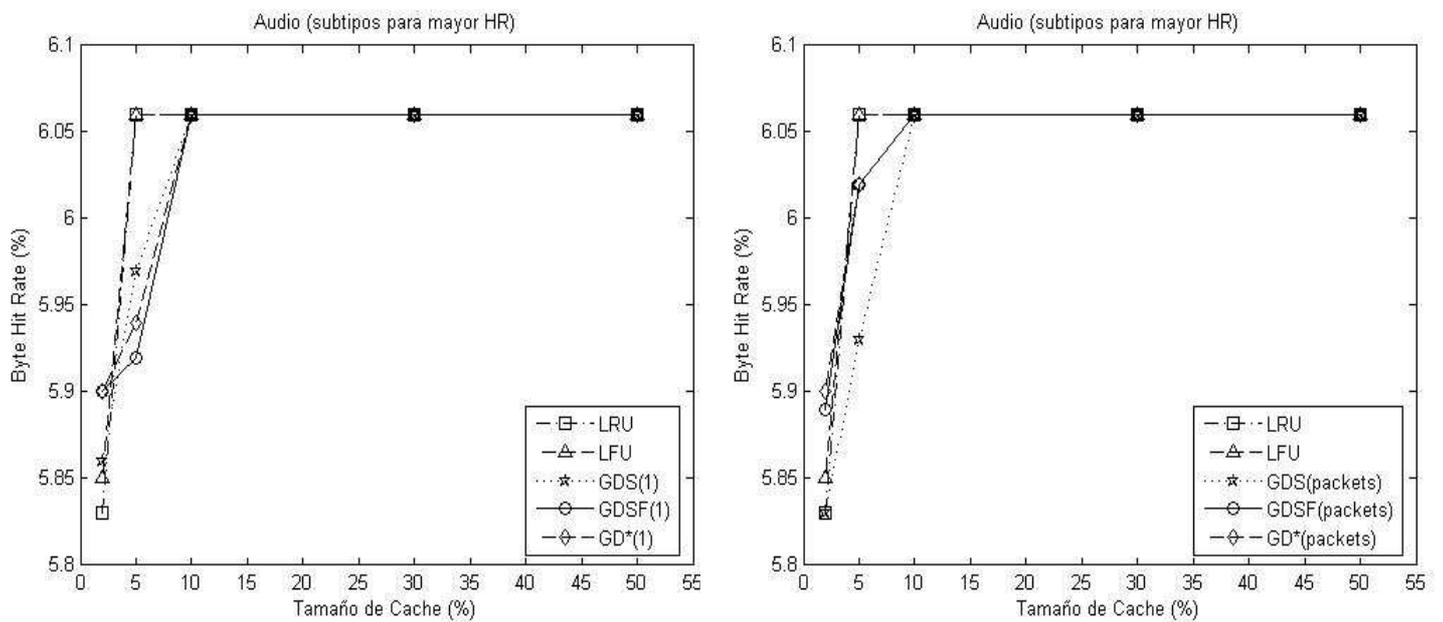
A la vista de estos resultados, se eligieron los subtipos *basic* y *wav* como los más apropiados para conseguir un mejor HR mientras que *mpeg* y *wav* se emplearon para buscar un mejor BHR.

Una vez realizadas las emulaciones, se plasmaron los resultados de las mismas en la figura 4.7., de los subtipos “para mayor HR”. En la figura 4.7(a) se presenta el comportamiento en cuanto al HR, tanto para coste uno como *packets*. En el caso de función de coste uno, el HR presenta una mejora notable respecto al mejor resultado hasta el momento, si bien dicha mejora se da sólo para los tamaños de *cache* más pequeños (del 2 al 10%) obteniéndose peores resultados para los tamaños mayores, pues el rendimiento obtenido para el tamaño del 10% no se consigue superar aunque se aumente el tamaño de la *cache*.

Con la función de coste *packets*, también se obtienen resultados mejores aunque algo por debajo de los conseguidos con la función de coste uno para tamaños de *cache* inferiores al 10% y los mismos resultados para tamaños iguales o superiores al 10% de *cache*. LRU y LFU que no se ven influenciados por la función de coste pero si por los subtipos a almacenar en *cache*, también proporcionan mejores resultados en cuanto al HR para los subtipos elegidos “para mayor HR”.



(a) HR para los subtipos para mayor HR de Audio con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos para mayor HR de Audio con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.7. HR (a) y BHR (b) para los subtipos para mayor HR de Audio

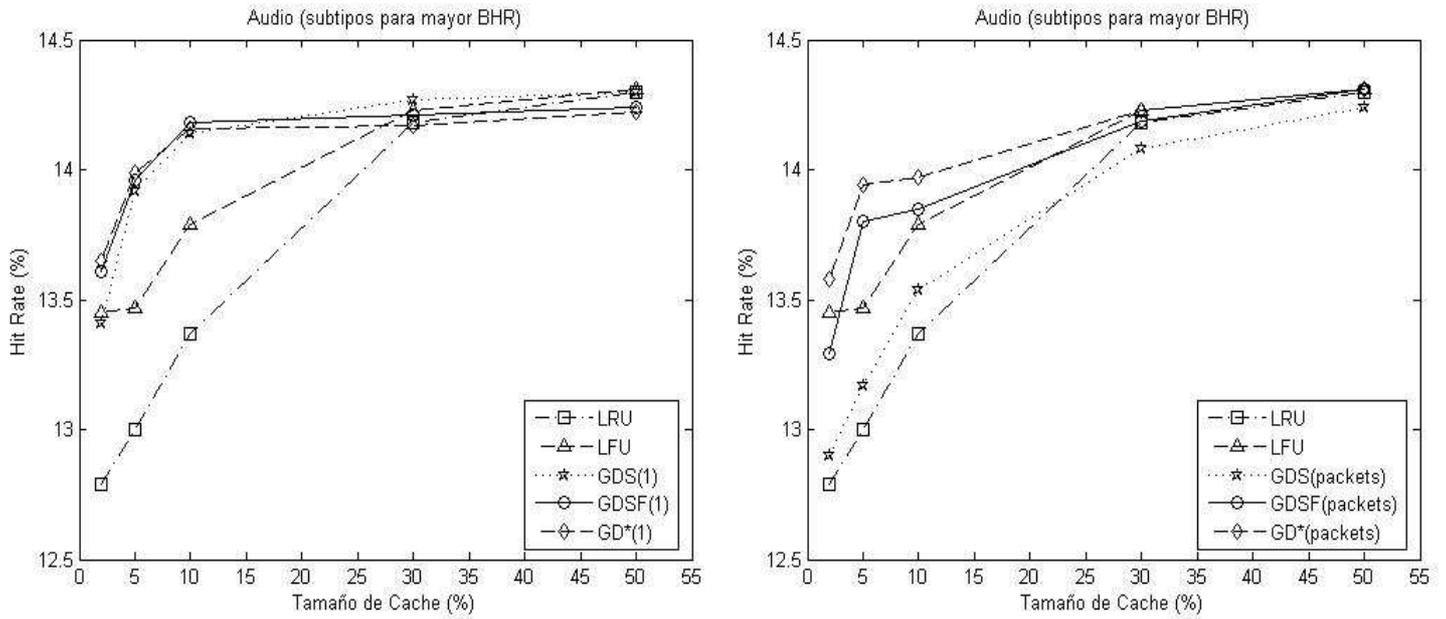
En lo referente al BHR, en la figura 4.7 (b) se observa un comportamiento mucho peor que lo obtenido hasta ahora, tanto si se emplea función de coste uno como *packets*, como cabía prever, ya que los subtipos se eligieron para tratar de proporcionar un elevado HR. Todas las políticas vuelven a ofrecer el mismo rendimiento para tamaños de *cache* por encima del 10%.

Por último, en la figura 4.8, aparecen los resultados obtenidos “para mayor BHR” de los subtipos de audio elegidos. En este caso, como se comprueba en la figura 4.8. (a), el HR que se consigue con las distintas políticas, para los subtipos “para mayor BHR” está muy por debajo de lo que se conseguía hasta ahora guardando todos los subtipos o los más frecuentes y aún más si lo comparamos con los resultados obtenidos para los subtipos seleccionados “para mayor HR”, como se comprobó en la figura 4.7. Dicha disminución del rendimiento se comprueba para todas las políticas siendo aún más acusada por las políticas de la familia Greedy cuando utilizan función de coste *packets*.

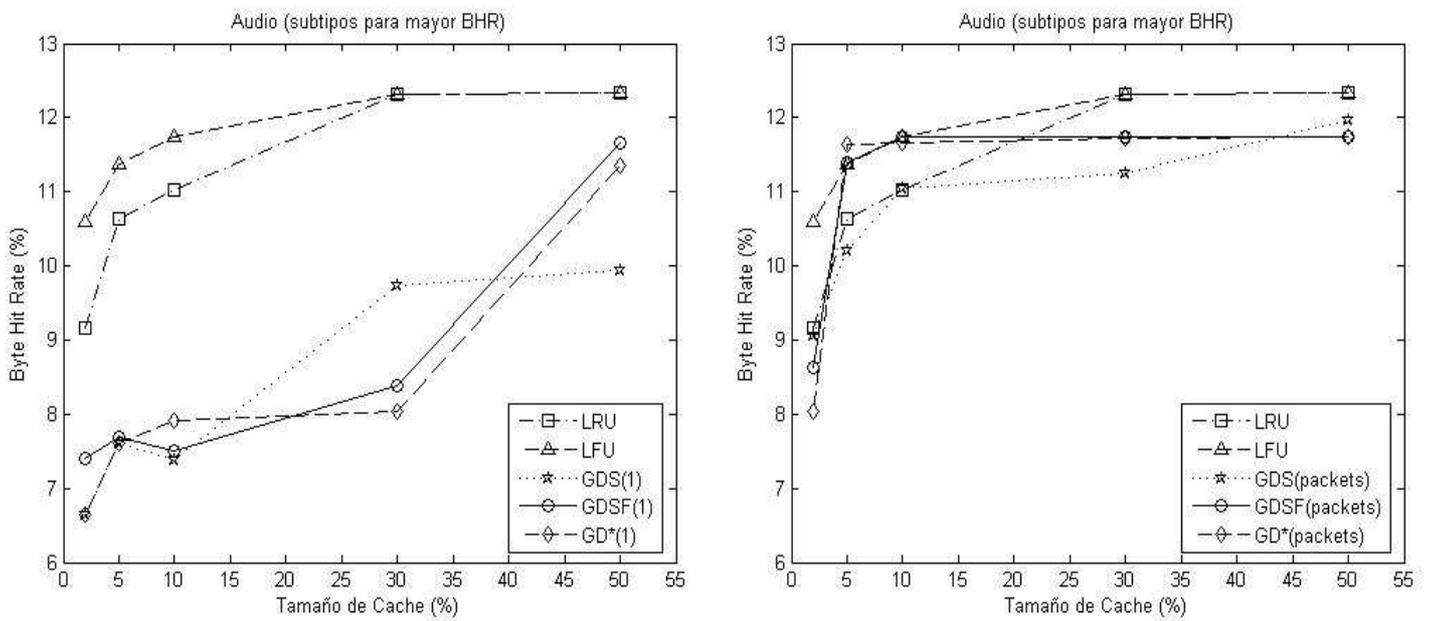
Por otro lado, el BHR no es mejor que el conseguido hasta el momento sin haberse hecho la distinción de subtipos “para mayor BHR”, como se muestra en la figura 4.7. (b). La única mejora se produce para tamaños de *cache* pequeños en las políticas LRU y LFU. Si la función de coste empleada es uno, las políticas de la familia Greedy proporcionan unos resultados bastante peores que en el caso de usarse la función de coste *packets*.

Por todo ello, se puede decir que resultaría acertado elegir los subtipos que proporcionan un mayor HR para ser los únicos guardados en *cache*, si lo que se busca es maximizar el HR haciendo uso de *caches* pequeñas (hasta el 10%) con la política GD*(1) . Sin embargo, esto tendrá la contrapartida de tener un BHR bajo. Además se observa que el HR con los subtipos “para mayor HR” conseguido, es el mismo para todas las políticas de reemplazo una vez el tamaño de la *cache* alcanza el 10% y no varía conforme dicho tamaño aumenta, por lo que no tiene sentido emplear *caches* de mayor tamaño. Si se desea trabajar con *caches* grandes (de tamaño 30 y 50%), las políticas GD*(1) o GDSF(1) incluyendo todos los subtipos, serían las mejores opciones.

Si lo que se desea es conseguir el mejor BHR, la política LFU sin hacerse distinción de subtipos, se muestra como la opción mejor.



(a) HR para los subtipos para mayor BHR de Audio con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos para mayor BHR de Audio con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.8. HR (a) y BHR (b) para los subtipos para mayor BHR de Audio

4.3.3. Imágenes

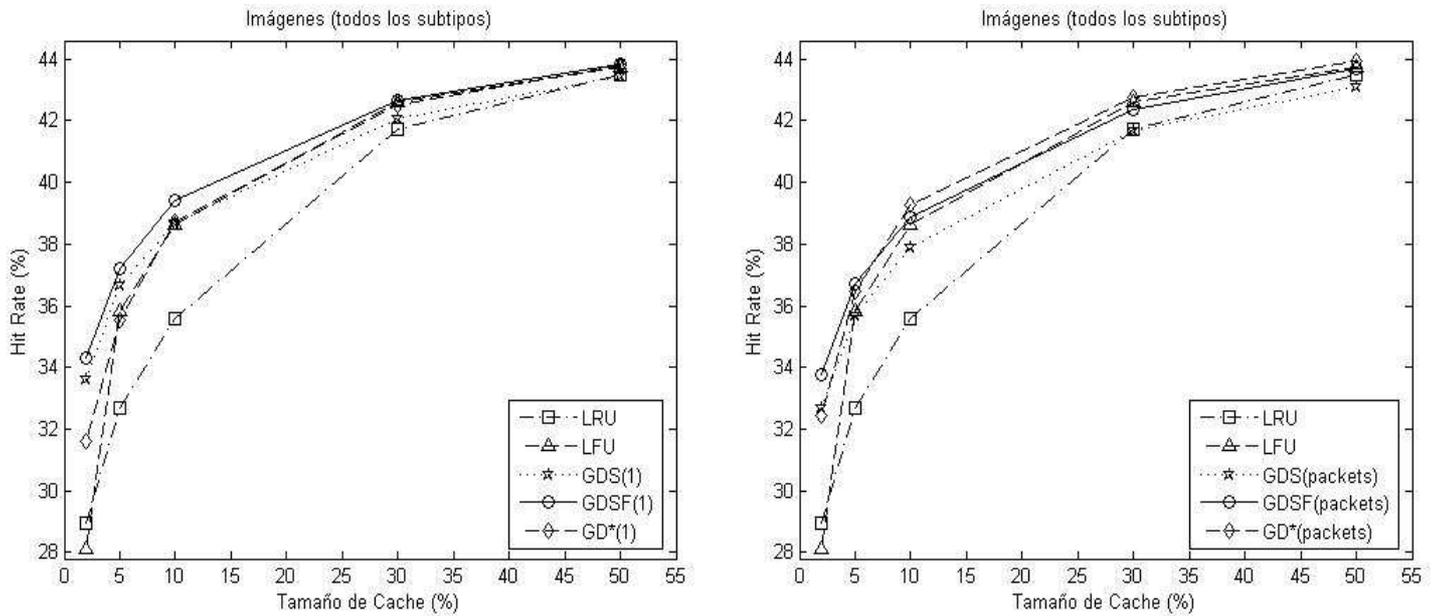
4.3.3.1. Resultados para todos los subtipos

Para las imágenes, del mismo modo que para el resto de tipos de documentos, se comenzó realizando el estudio del comportamiento de la *cache proxy* al guardarse en ella todos los subtipos. En la figura 4.9. aparecen los resultados de las emulaciones, tanto para el HR como para el BHR.

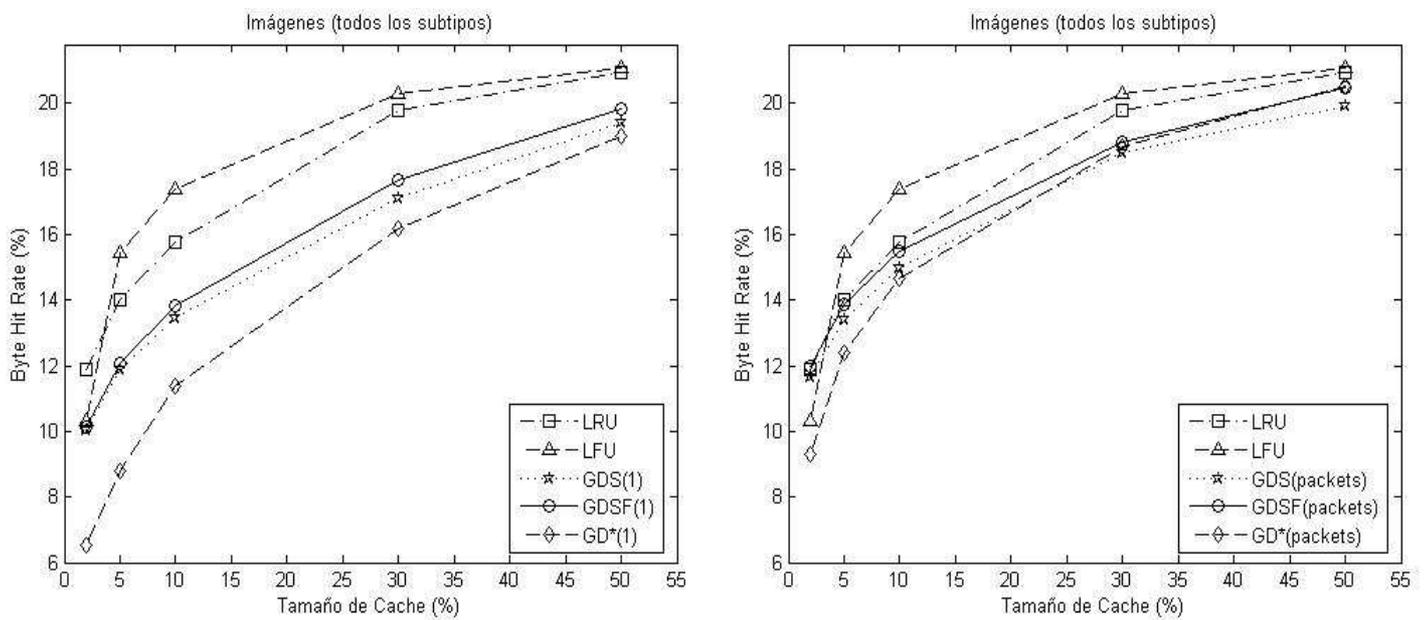
En la figura 4.9. (a) se observan los resultados obtenidos para el HR cuando se guardan en *cache* todos los subtipos de imágenes. Se observa que las políticas de la familia Greedy son las que ofrecen unos mejores resultados, siendo la mejor de ellas GDSF(1), mientras que la política que proporciona un peor rendimiento es LRU.

Cuando se emplea como función de coste *packets*, los resultados que se obtienen son similares aunque algo peores que cuando se emplea la función de coste uno, como se puede apreciar en la parte derecha de la figura 4.9. (a).

Respecto al BHR, en la figura 4.9. (b) se aprecia que las políticas LFU y LRU por ese orden, son las que ofrecen un mejor rendimiento en comparación con las políticas de la familia Greedy. Dicho comportamiento se observa tanto cuando la función de coste es uno como *packets*, si bien, con función de coste *packets* el comportamiento de las políticas de la familia Greedy es mejor y la diferencia de rendimiento es bastante menor que con coste uno. Al igual que ocurriera con el HR, la política GDSF es la mejor de entre las políticas de la familia Greedy.



(a) HR para todos los subtipos de Imágenes con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para todos los subtipos de Imágenes con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.9. HR (a) y BHR (b) para todos los subtipos de Imágenes

4.3.3.2. Resultados para los subtipos más habituales

Para el estudio del rendimiento empleando los “subtipos más habituales”, se comenzó comprobando los porcentajes de documentos de cada subtipo de imágenes y la cantidad de bytes de dichos documentos, respecto del total de documentos de imágenes. En base a estos datos, se escogieron los subtipos de imágenes más habituales. Dichos subtipos aparecen listados a continuación en la tabla 4.6.

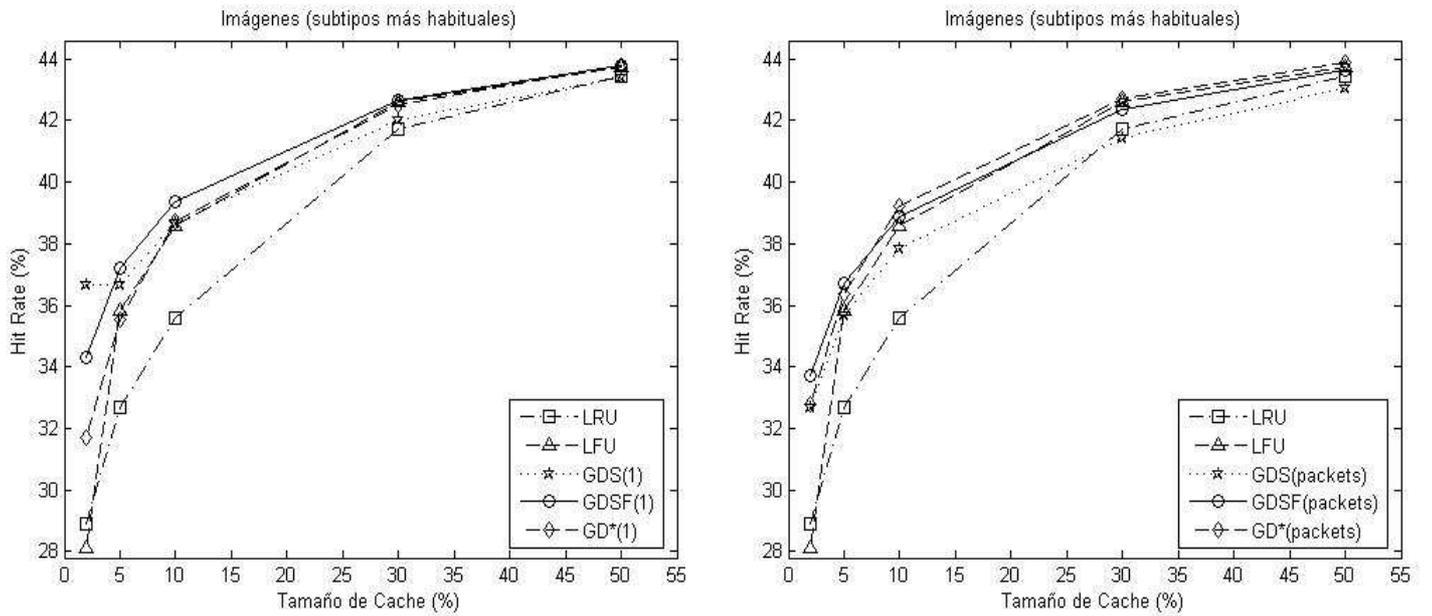
Tabla 4.6. Características de los subtipos “más habituales” de imágenes

Subtipos de imágenes	% documentos	% bytes
bmp : x-bitmap : x-bmp : x-ms-bmp	0.08	0.34
gif	70.38	39.65
jpg : jpeg : pjpeg	29.05	59.36
png : x-png	0.41	0.54
Total	99.93	99.91

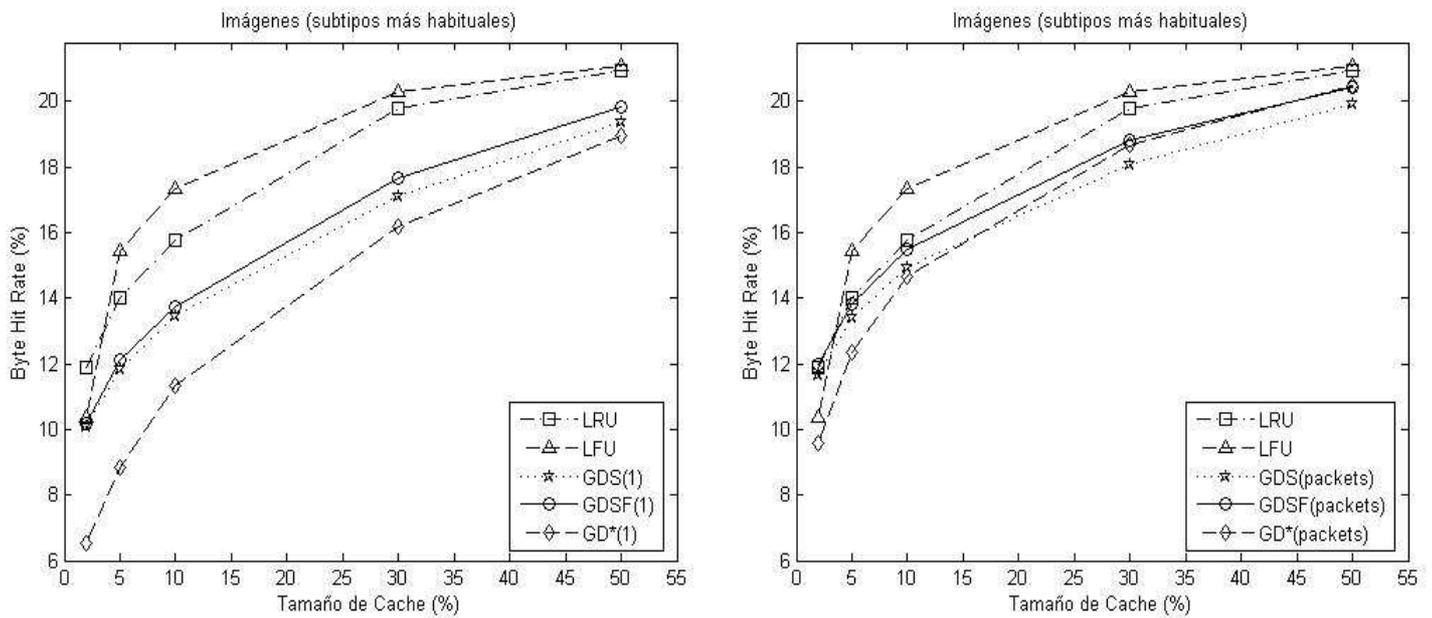
A la vista de estos resultados, se optó por no realizar las emulaciones para los subtipos para “mayor HR” y para “mayor BHR” puesto que al descartar los subtipos *bmp* y *png* apenas se obtendría variación en los resultados respecto a los “subtipos más habituales”. Se realizaron las distintas emulaciones guardando en *cache* los subtipos “más habituales”. Los resultados de las emulaciones se muestran en las gráficas de la figura 4.10.

En la parte (a) de la figura 4.10. se aprecia el comportamiento de las distintas políticas de reemplazo respecto al HR. Se comprueba que se mantiene el comportamiento observado cuando se guardaban en *cache* todos los subtipos. Los resultados obtenidos son muy parecidos a los del apartado 4.3.3.1 aunque levemente peores excepto para GDSF(1) que proporciona los mejores resultados para el HR al guardarse en *cache* los “subtipos más habituales”.

Las políticas Greedy usando una función de coste *packets* no consiguen mejorar los resultados respecto al HR que se obtenían al usar la función de coste uno. Las variaciones en los resultados ofrecidos por todas las políticas para los “subtipos más habituales”, en lo referente al HR, son siempre inferiores al 1%.



(a) HR para los subtipos más habituales de Imágenes con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos más habituales de Imágenes con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.10. HR (a) y BHR (b) para los subtipos más habituales de Imágenes

Para los “subtipos más habituales”, los resultados obtenidos en lo relativo al BHR se muestran en la parte (b) de la figura 4.10. En este caso también se observa una levísima reducción del rendimiento respecto a los resultados para “todos los subtipos”, aunque en la práctica se pueden considerar iguales. Por tanto, LFU seguida de LRU seguirán siendo las políticas de reemplazo que mayor BHR ofrecerán.

Respecto a las políticas de la familia Greedy, GDSF seguirá siendo la mejor y en general, los resultados respecto al BHR de todas las políticas de la familia Greedy serán mejores y más próximos a LRU y LFU cuando la función de coste que se utilice sea *packets*.

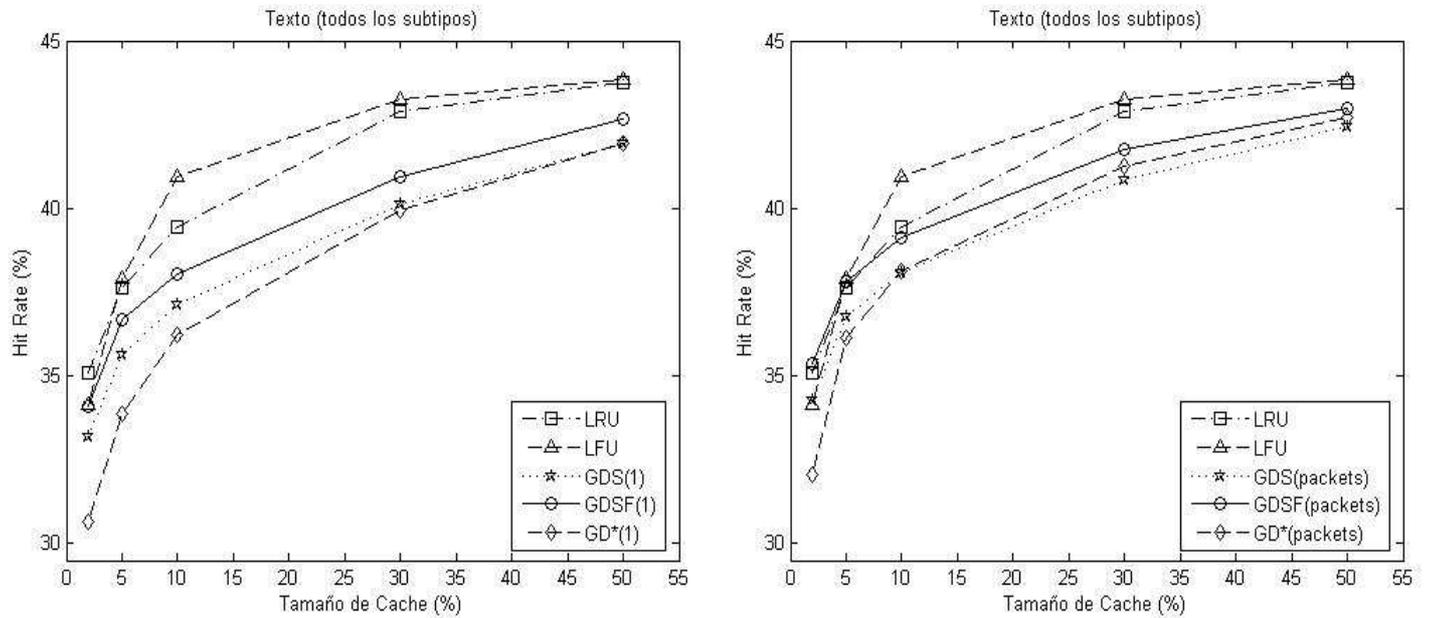
De los resultados obtenidos se puede concluir que la política GDSF(1) es la que ofrece el mejor rendimiento cuando se guardan en *cache* los subtipos más habituales de imágenes. En cuanto al BHR, la política LFU proporciona los mejores resultados y por tanto se presenta como la más adecuada a utilizar.

4.3.4. Texto

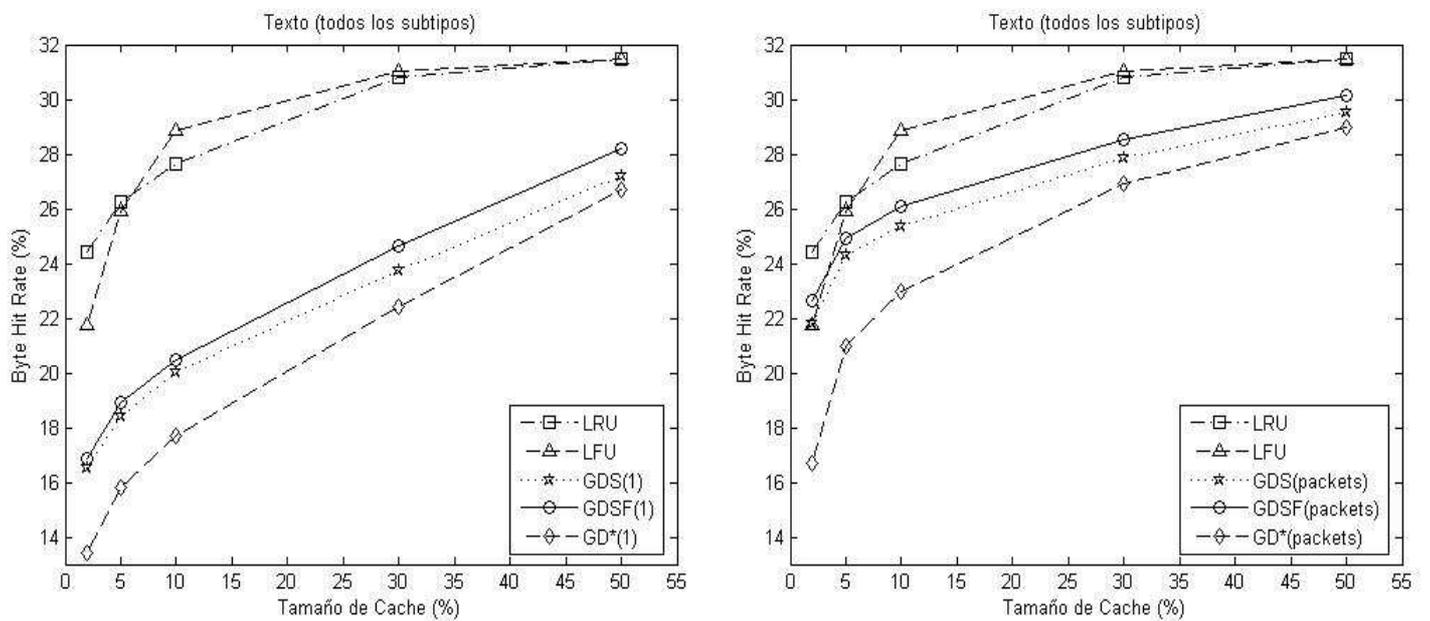
4.3.4.1. Resultados para todos los subtipos

Al igual que para el resto de tipos de documentos, se comenzó realizando el estudio del comportamiento de la *cache proxy* cuando se guardan en ella todos los subtipos, en este caso, de texto. En la figura 4.11. aparecen los resultados de las emulaciones, tanto para el HR como para el BHR.

La figura 4.11. (a), muestra los resultados obtenidos para el HR cuando se guardan en *cache* todos los subtipos de texto. Se observa que la política LFU es la que da lugar a los mejores resultados en cuanto a HR de entre todas las opciones, seguida de LRU. Por su parte, las políticas de la familia Greedy ofrecen un peor rendimiento, siendo siempre la mejor de ellas GDSF. Aún cuando los resultados siempre están por debajo de LRU, empleando función de coste *packets* en las políticas Greedy, en contra de lo que se podría suponer, se consiguen mejorar y se encuentran más próximos a LRU, como se aprecia en la figura 4.11 (a) en la parte derecha.



(a) HR para todos los subtipos de Texto con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para todos los subtipos de Texto con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.11. HR (a) y BHR (b) para todos los subtipos de Texto

En lo relativo al BHR, en la figura 4.11. (b) se observa que las políticas LFU y LRU ofrecen un mejor rendimiento respecto a las políticas de la familia Greedy, de forma análoga a como se acaba de indicar que sucede con el HR. Dicho comportamiento se observa tanto cuando la función de coste es uno como *packets*, si bien, con función de coste *packets*, la diferencia de rendimiento es bastante menor que con coste uno. Al igual que ocurriera anteriormente, la política GDSF es siempre la mejor y GD* la peor, de entre las políticas de la familia Greedy.

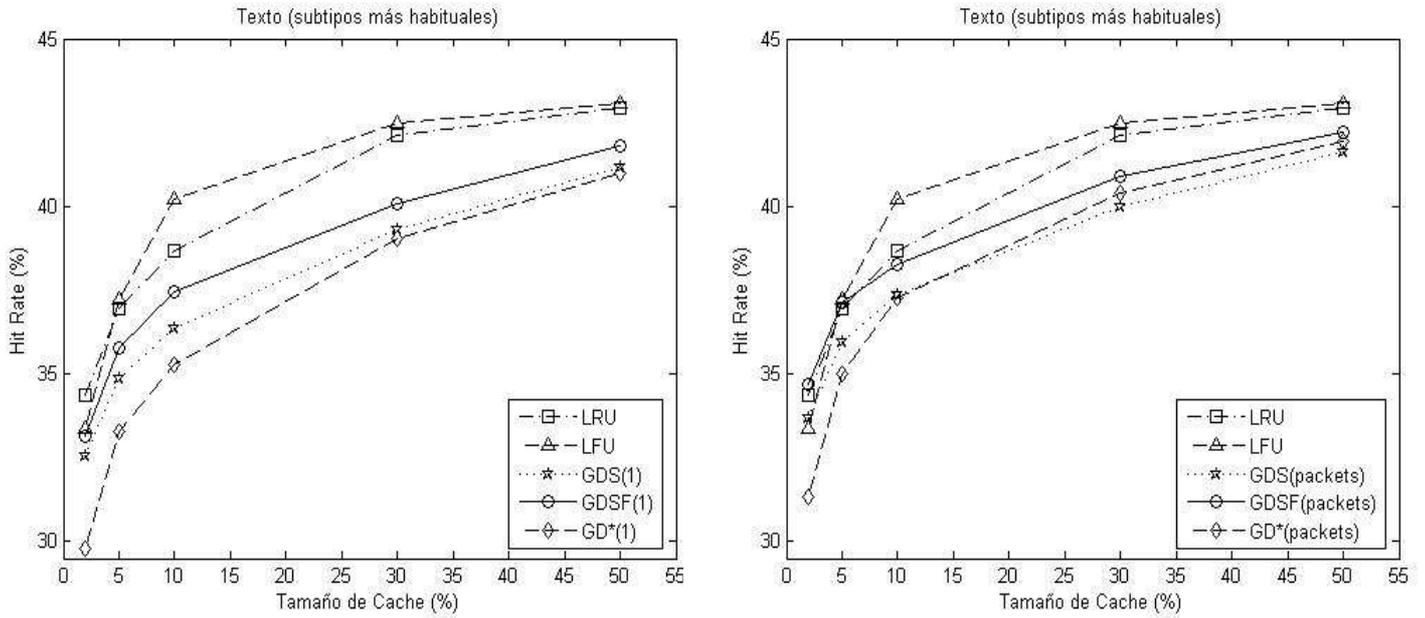
4.3.4.2. Resultados para los subtipos más habituales

Siguiendo el mismo procedimiento ya descrito con anterioridad, se comprobaron los porcentajes de documentos de cada subtipo de texto y la cantidad de bytes de dichos documentos, respecto del total de documentos de texto. En base a estos datos, se escogieron los subtipos de texto más habituales y que aparecen listados a continuación en la tabla 4.7.

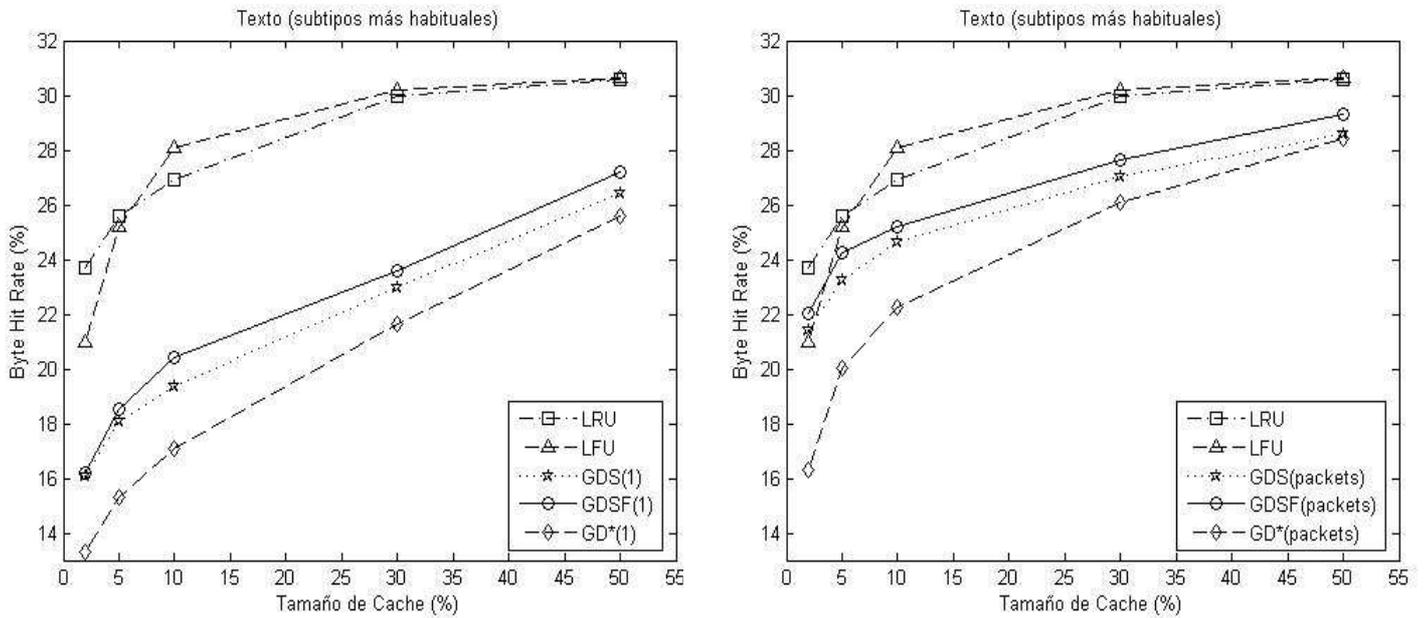
Tabla 4.7. Características de los subtipos “más habituales” de texto

Subtipos de texto	% documentos	% bytes
css	17.43	6.57
html	56.68	64.14
js : javascript : x-javascript	0.49	0.23
plain	22.05	26.92
xml	2.7	1.67
Total	99.36	99.55

Guardando sólo los subtipos “más habituales” en *cache*, se realizaron las correspondientes emulaciones, cuyos resultados se muestran en las gráficas de la figura 4.12. En la parte (a) de dicha figura se aprecia el comportamiento de las distintas políticas de reemplazo respecto al HR. Se puede comprobar como se mantiene la tendencia apuntada por las emulaciones en las que se empleaban todos los subtipos de texto, pues LFU sigue siendo la política que proporciona un mejor rendimiento, seguida de LRU y GDSF. Esta última, de nuevo, se presenta como la mejor opción dentro de la familia de políticas Greedy.



(a) HR para los subtipos más habituales de Texto con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos más habituales de Texto con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.12. HR (a) y BHR (b) para los subtipos más habituales de Texto

Al igual que sucediera anteriormente, las políticas Greedy usando una función de coste *packets* proporcionan un HR mayor y más próximo a LRU. Los resultados ofrecidos por todas las políticas con todas sus posibles variantes para los “subtipos más habituales” son ligeramente peores que los que se obtuvieron para “todos los subtipos” anteriormente, aunque con variaciones en torno al 1%.

Los resultados obtenidos respecto de los “subtipos más habituales” en lo relativo al BHR se muestran en la parte (b) de la figura 4.12. Al igual que ocurre con el HR, el BHR proporciona un rendimiento levemente inferior al que se obtuvo con “todos los subtipos” y se mantiene el mismo comportamiento ya descrito, es decir, LFU, LRU y GDSF como políticas más eficientes. Como era de esperar, los resultados de las políticas de la familia Greedy respecto al BHR son bastante mejores cuando la función de coste utilizada es *packets*.

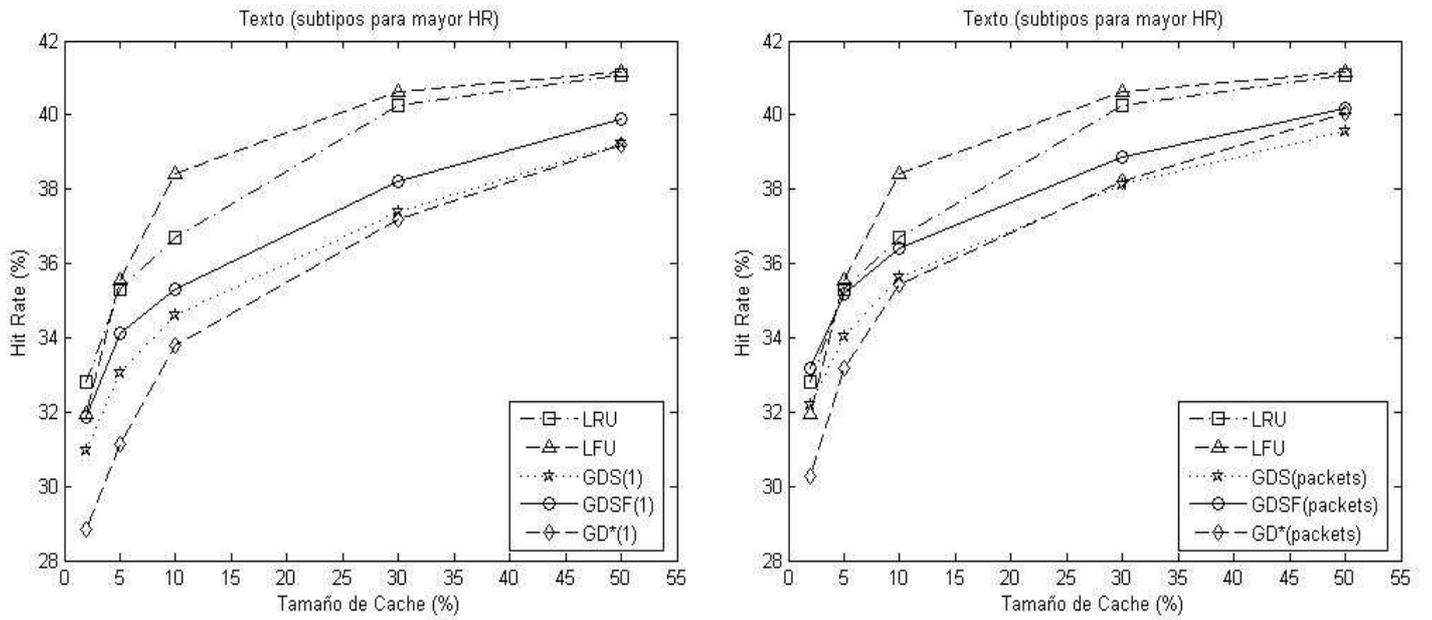
4.3.4.3. Resultados para subtipos con mayor HR y BHR

A partir de los subtipos “más habituales” se realizaron emulaciones con el fin de conocer cuáles proporcionaban un mayor HR y BHR siguiendo el procedimiento ya explicado. Los resultados obtenidos se recogen en la tabla 4.8.

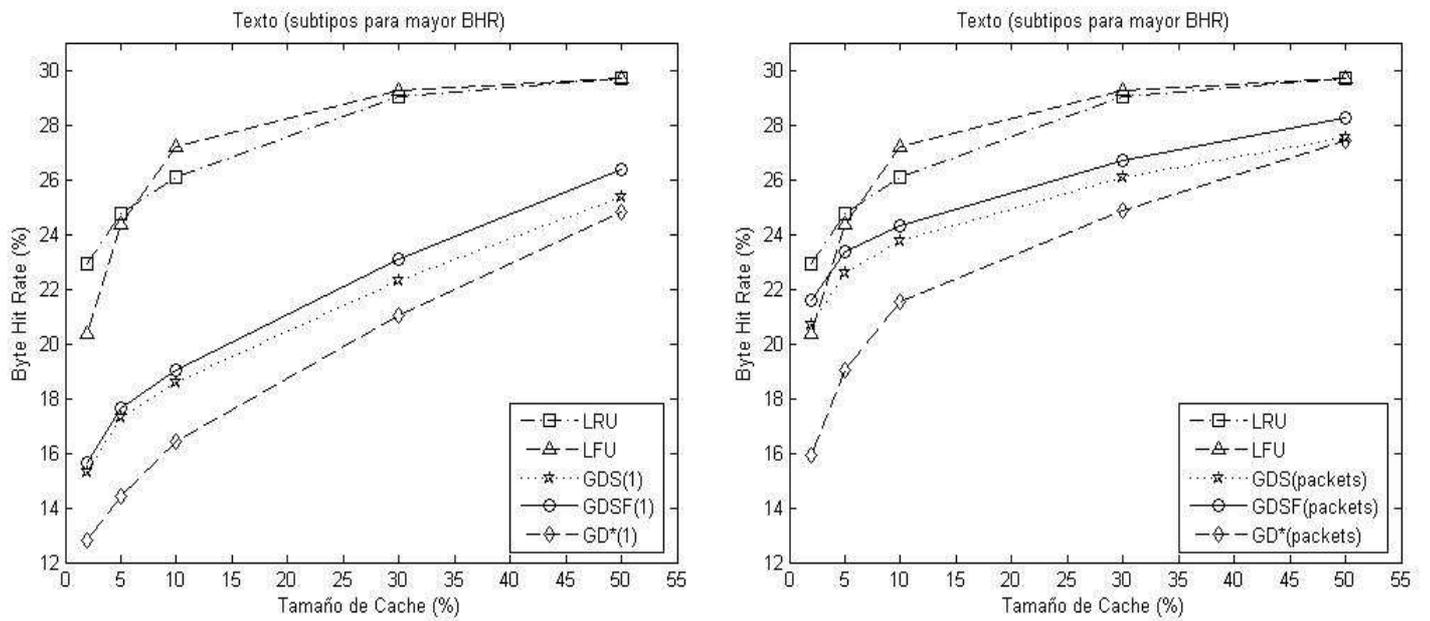
Tabla 4.8. Aportación al HR y BHR de los subtipos “más habituales” de audio

Subtipos de texto	% documentos	% bytes
css	12.32	4.45
html	22.23	18.32
js : javascript : x-javascript	0.69	0.2
plain	7.89	7.29
xml	1.99	0.85

Según muestra la tabla 4.8., se eligieron los subtipos *css*, *html* y *plain* para tratar de conseguir tanto un mejor HR como BHR, pues por coincidencia en este caso, los mismos subtipos aparecían como los apropiados a la hora de maximizar ambas métricas.



(a) HR para los subtipos para mayor HR de Texto con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos para mayor BHR de Texto con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.13. HR (a) y BHR (b) para los subtipos para mayor HR y BHR de Texto

Los resultados de las emulaciones para conseguir un mayor HR con la inclusión en *cache*, únicamente de los tres subtipos antes mencionados se presenta en la figura 4.13. (a). De nuevo LFU seguida de LRU y GDSF se muestra como la mejor elección a la hora de conseguir el mayor HR, si bien los resultados obtenidos son algo peores que en los casos anteriores y como también ocurriera para los casos ya vistos, las políticas de la familia Greedy presentan un mejor comportamiento al utilizar como función de coste *packets*.

Respecto al BHR, en la figura 4.13 (b) se observa que no ha habido variación en la pauta seguida por cada una de las políticas y que ha sido repetidamente descrita. En cuanto a los porcentajes de BHR conseguidos, son algo inferiores a los que se consiguieron con “los subtipos más frecuentes”.

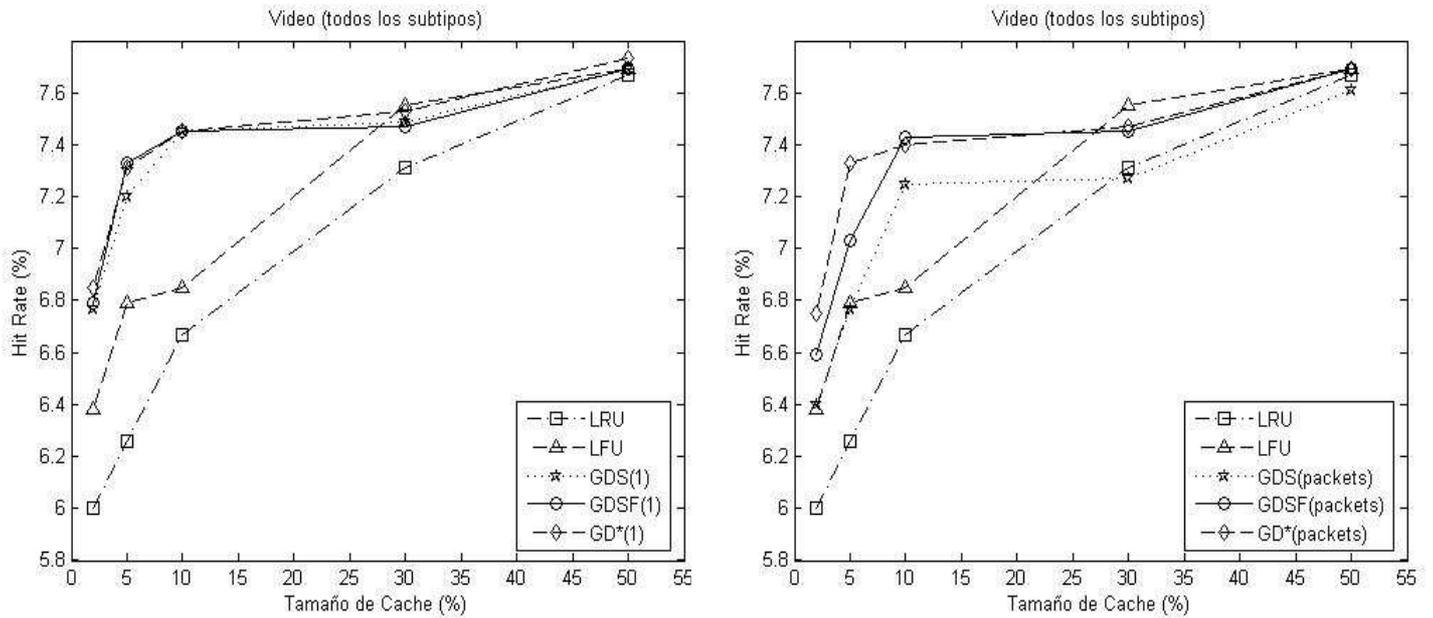
De los resultados se puede deducir que para los documentos de texto la política LFU es la que proporciona unos mejores resultados siempre, seguida de cerca por LRU. Por tanto, el uso de alguna de estas dos políticas sería lo más adecuado ya que con las demás se consiguen resultados peores y resultan más complejas.

4.3.5. Vídeo

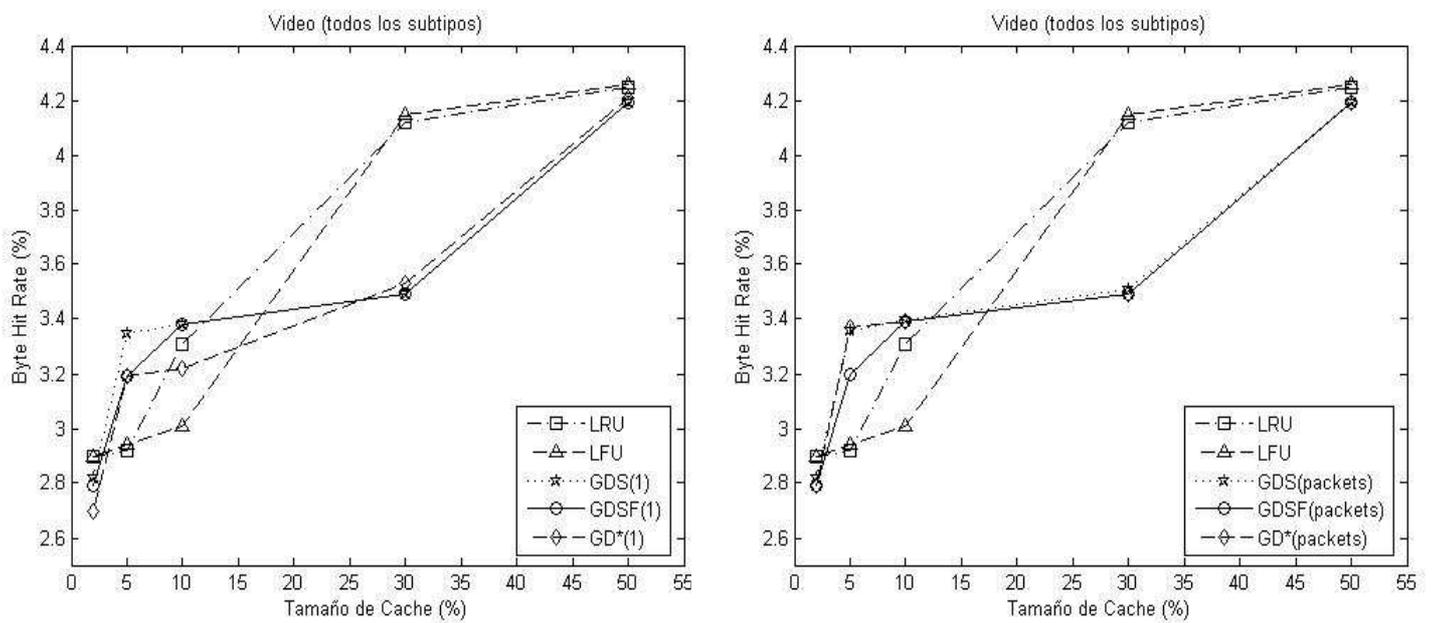
4.3.5.1. Resultados para todos los subtipos

Para los documentos de vídeo, al igual que ocurrió para el resto de tipos de documentos, se comenzó realizando el estudio del comportamiento de la *cache proxy* para las distintas políticas de reemplazo, cuando se guardaban en *cache* todos los subtipos de vídeo. En la figura 4.14. se presentan los resultados de las emulaciones, tanto para el HR como para el BHR.

En la figura 4.14. (a), se han representado los resultados obtenidos para el HR cuando se guardan en *cache* todos los subtipos de vídeo. Se comprueba que las políticas de la familia Greedy son las que proporcionan un mejor HR, siendo la mejor GD* y la peor de ellas GDS, sobre todo para tamaños de *cache* pequeños (del 10% e inferiores).



(a) HR para todos los subtipos de Vídeo con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para todos los subtipos de Vídeo con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.14. HR (a) y BHR (b) para todos los subtipos de Vídeo

Utilizando función de coste uno, las tres políticas Greedy presentan un comportamiento muy parecido, mientras que con función de coste *packets*, los resultados son algo peores en todas ellas. Dicho empeoramiento hace que sean superadas en rendimiento por LFU para los tamaños de *cache* del 30 y 50% e incluso que GDS(*packets*) se vea superada en rendimiento por LRU para dichos tamaños de *cache*. Respecto de las dos políticas que son independientes de la función de coste, LRU proporciona siempre los peores resultados en cuanto a HR.

En lo relativo al BHR, en la figura 4.14. (b) se observa que LFU y LRU en ese orden, proporcionan siempre los mejores resultados para tamaños de *cache* del 30 y 50%. Respecto a las políticas de la familia Greedy se observa que los resultados en cuanto al BHR son mejores si se emplea función de coste *packets*. Para tamaños de *cache* del 10% e inferiores, los comportamientos de las tres políticas Greedy son muy parecidos. Para tamaños inferiores al 10%, GDS(*Packets*), GDSF(*Packets*) o GD*(*Packets*) son las políticas más apropiadas a emplear a la hora de conseguir el mejor BHR.

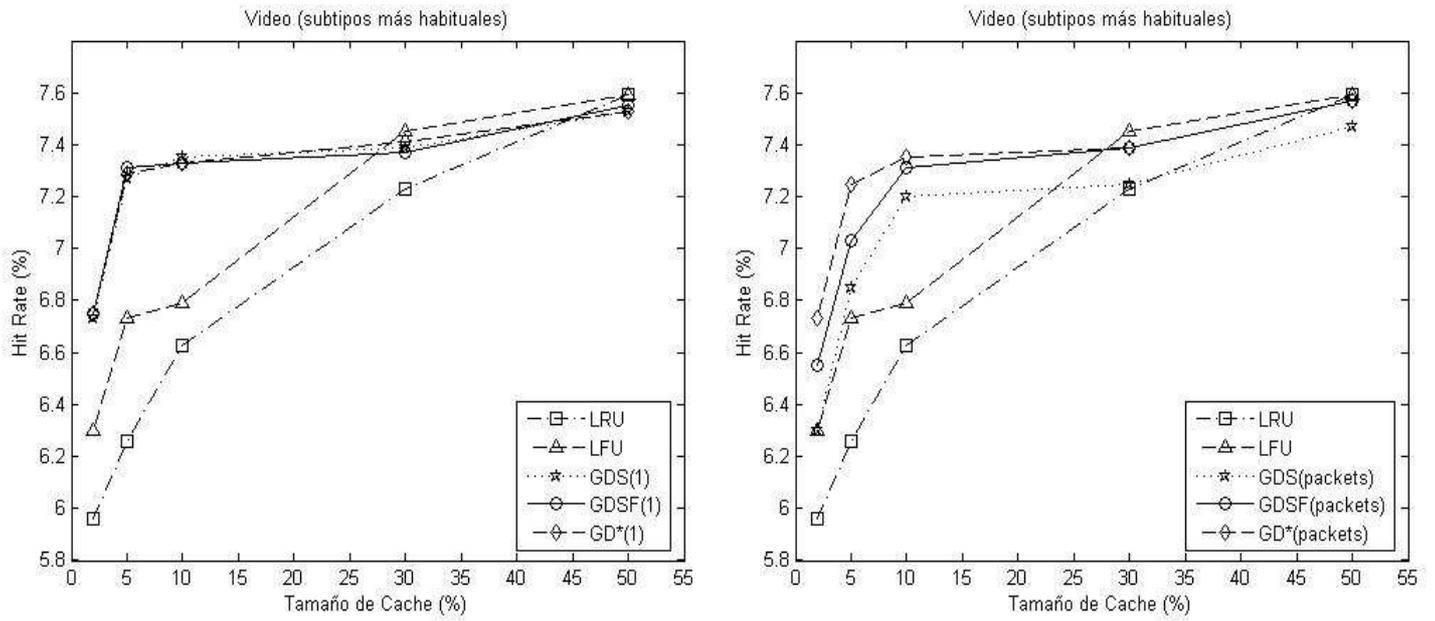
4.3.5.2. Resultados para los subtipos más habituales

Aplicando el mismo procedimiento que se empleó para el resto de tipos de documentos, se comprobaron los porcentajes de documentos de cada subtipo de vídeo y la cantidad de bytes de dichos documentos, respecto del total de documentos de vídeo. En base a estos datos, se escogieron los subtipos de vídeo más habituales y son los que aparecen listados en la tabla 4.9.

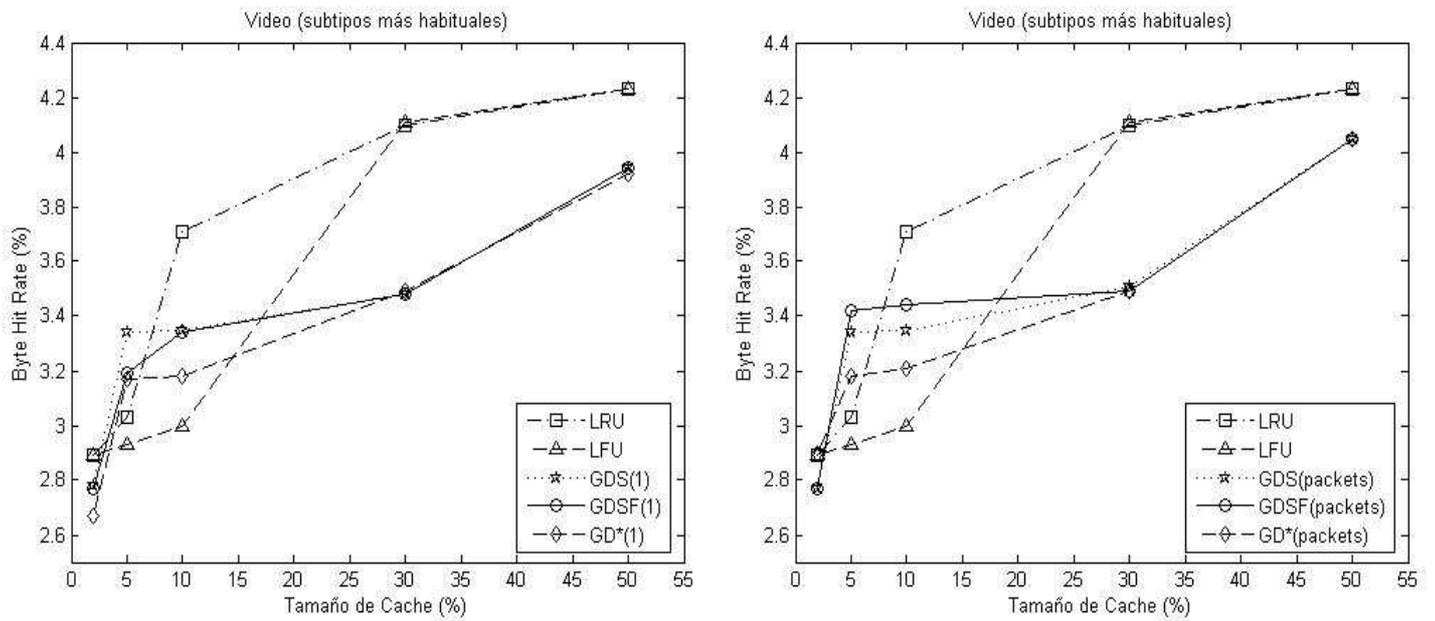
Tabla 4.9. Características de los subtipos “más habituales” de vídeo

Subtipos de vídeo	% documentos	% bytes
mpeg : x-mpeg	30.05	72.35
quicktime	3.06	15.82
wmv : x-ms-wmv	3.38	9.23
x-ms-asf	59.59	1.33
Total	99.09	98.75

Almacenando sólo estos subtipos en *cache*, se realizaron las correspondientes emulaciones. Los resultados obtenidos se recogen en las gráficas de la figura 4.15. En la parte (a) de la figura antes indicada, aparece el rendimiento ofrecido por las distintas políticas de reemplazo respecto al HR.



(a) HR para los subtipos más habituales de Vídeo con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos más habituales de Vídeo con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.15. HR (a) y BHR (b) para los subtipos más habituales de Vídeo

Se aprecia una disminución en el HR para todas las políticas y para ambas funciones de coste, de menos del 0.2%, por lo que se puede considerar que los resultados son prácticamente idénticos a los obtenidos para todos los subtipos, pues la tendencia apuntada por éstos, también se mantiene ahora para los “subtipos más habituales”. El motivo de dicho resultado probablemente sea que los “subtipos más habituales” constituyen más del 99% del total de documentos de vídeo, de ahí la poca variación en los resultados.

Las políticas de la familia Greedy seguirán teniendo un comportamiento mejor empleando función de coste uno que empleando función de coste *packets*. Con la función de coste uno, todas ofrecerán aproximadamente el mismo rendimiento. Sin embargo, con el empleo de la función de coste *packets*, GD* dará los mejores resultados de las tres.

Respecto al BHR, en la figura 4.15. (b) se observa un leve empeoramiento, más apreciable para las políticas que emplean función de coste uno que cuando se usa *packets*. Para tamaños del 5 y 10% de *cache*, LRU es la única política de reemplazo que presenta una mejora en el rendimiento respecto al BHR, teniendo en cuenta los subtipos “más frecuentes”. Dicha mejora hace que para el tamaño de *cache* del 10%, LRU sea la política de reemplazo que proporcione un mayor BHR.

4.3.5.3. Resultados para subtipos con mayor HR y BHR

Como para el resto de tipos de documentos, a partir de los subtipos “más habituales” se realizaron emulaciones para conocer cuáles proporcionaban un mayor HR y BHR. Los resultados obtenidos se muestran en la tabla 4.10.

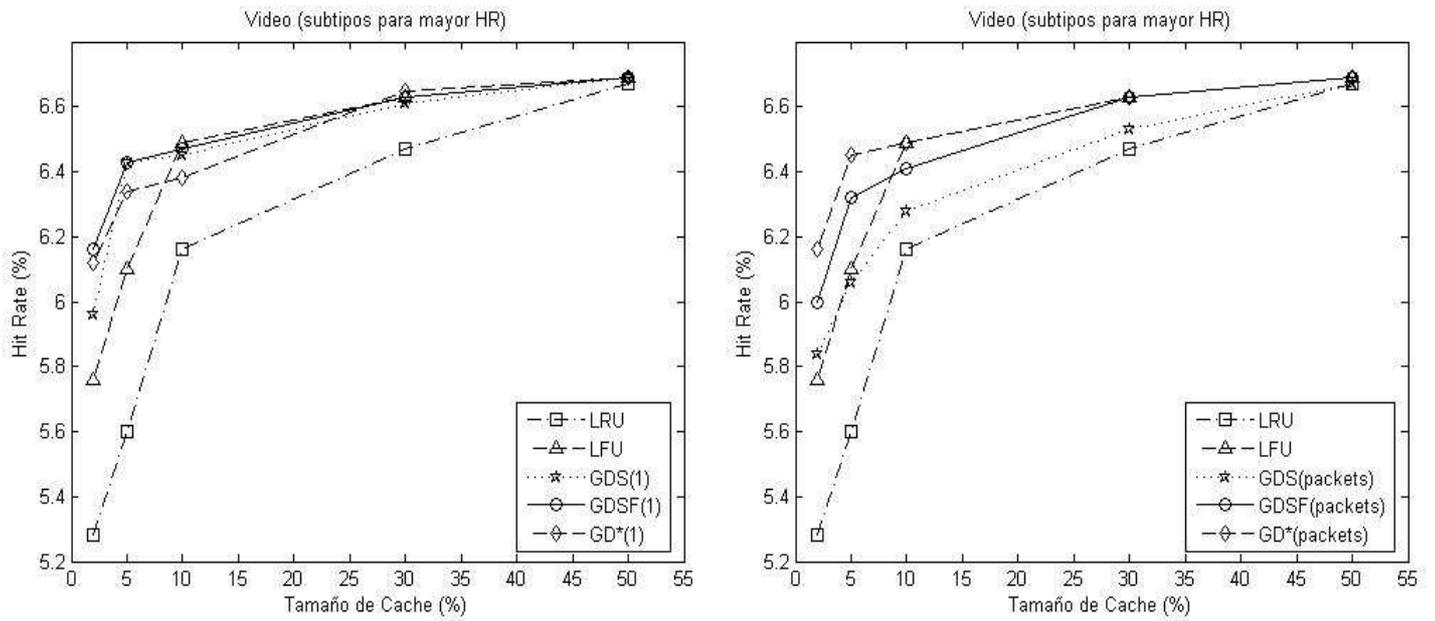
Tabla 4.10. Aportación al HR y BHR de los subtipos “más habituales” de vídeo

Subtipos de vídeo	% documentos	% bytes
mpeg : x-mpeg	2.44	2.85
quicktime	0.6	0.39
wmv : x-ms-wmv	0.4	0.94
x-ms-asf	4.25	0.08

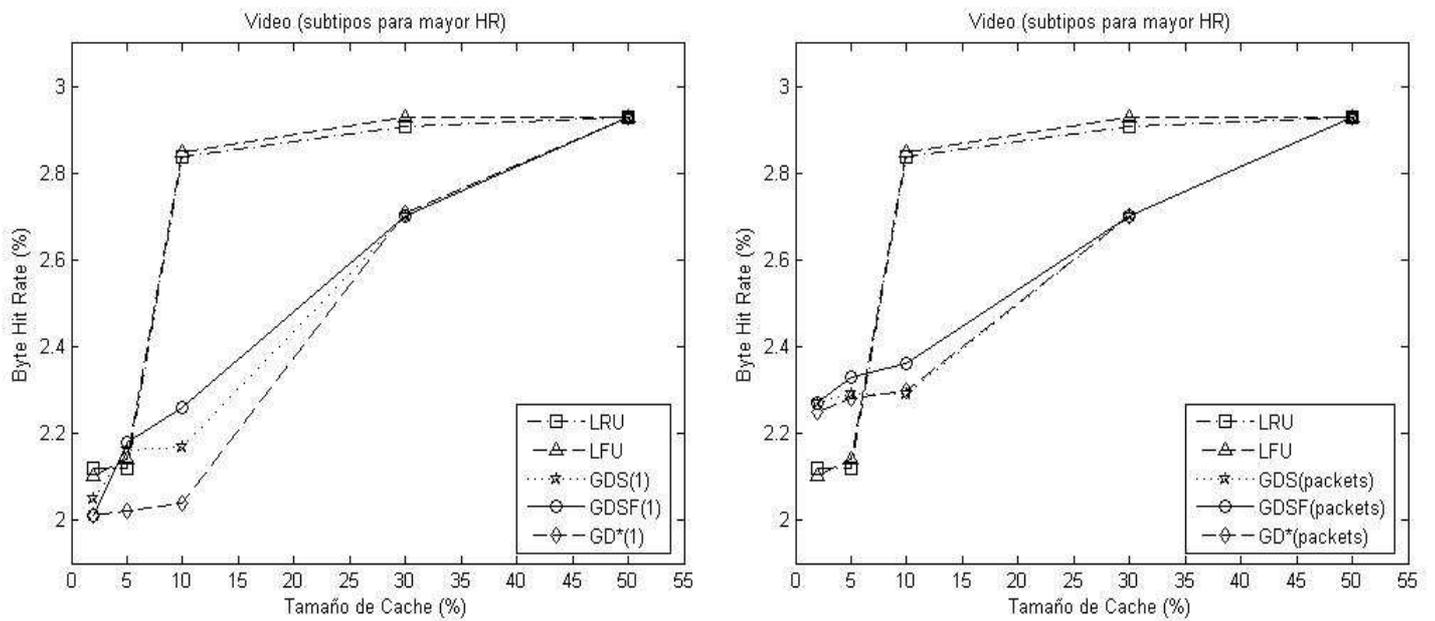
Según estos resultados, se eligieron los subtipos *mpeg* y *x-ms-asf* como los más apropiados para conseguir un mejor HR mientras que *mpeg* y *wmv* se emplearon para buscar un mejor BHR.

Los resultados de las emulaciones se muestran en las figuras 4.16. y 4.17. En cuanto a los subtipos “para mayor HR”, en la figura 4.16(a) se aprecia el comportamiento tanto para coste uno como *packets*, del HR. Se aprecia un mal comportamiento del HR para todas las políticas siendo muy malo cuando se emplea la función de coste *packets* en las políticas Greedy. El mayor empeoramiento lo sufren las políticas de la familia Greedy, por lo que su rendimiento se ve equiparado al de LFU. Para un tamaño de *cache* del 50% todas las políticas ofrecen los mismos resultados y conforme disminuye el tamaño de *cache* se hacen más patentes las diferencias.

Por su parte, en la figura 4.16 (b) se observa que el rendimiento ofrecido para el BHR es aún peor ya que los subtipos se eligieron tratando de mejorar el HR y por tanto sin tener en cuenta el rendimiento que ofrecería el BHR. En este caso se hace más patente el bajo rendimiento ofrecido por las políticas Greedy respecto de LFU y LRU que son las dos mejores. El uso de función de coste *packets*, por parte de las políticas Greedy, tan sólo mitiga en parte su bajo rendimiento. El único punto en el que todas las políticas ofrecen igual comportamiento vuelve a ser para un tamaño de *cache* del 50%.



(a) HR para los subtipos para mayor HR de Video con coste 1 (izquierda) y coste *packets* (derecha)



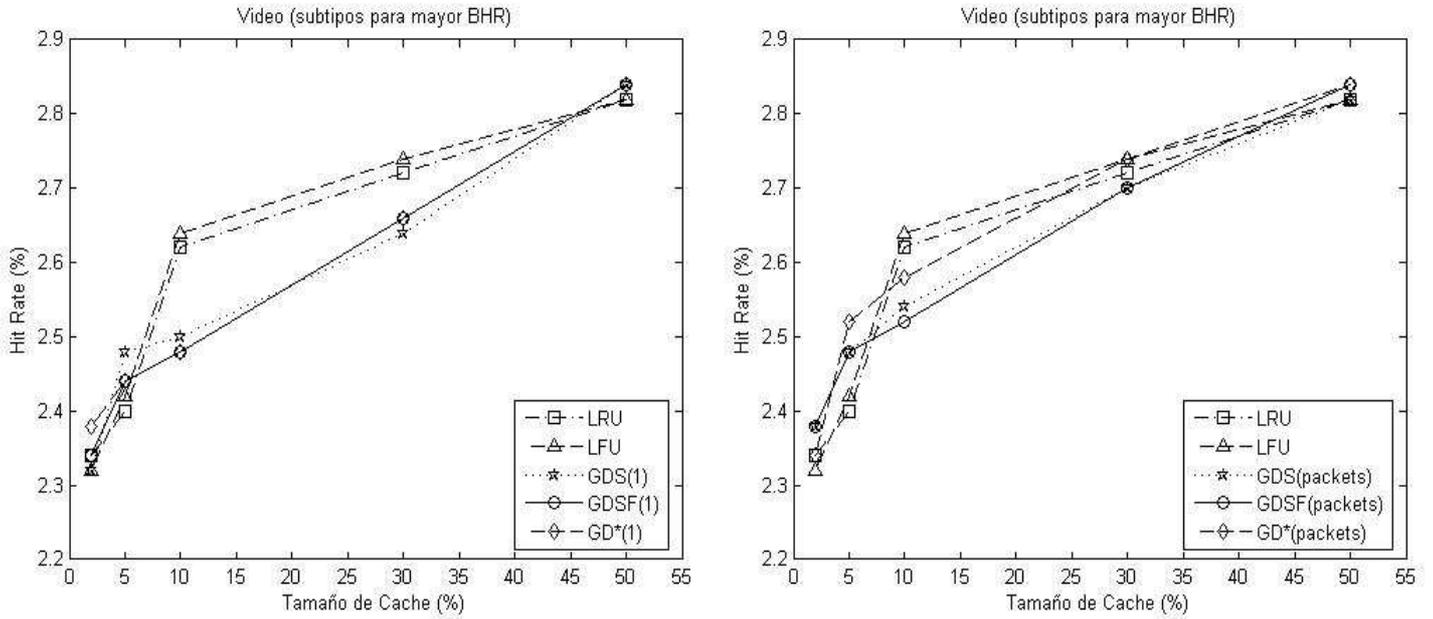
(b) BHR para los subtipos para mayor HR de Video con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.16. HR (a) y BHR (b) para los subtipos para mayor HR de Video

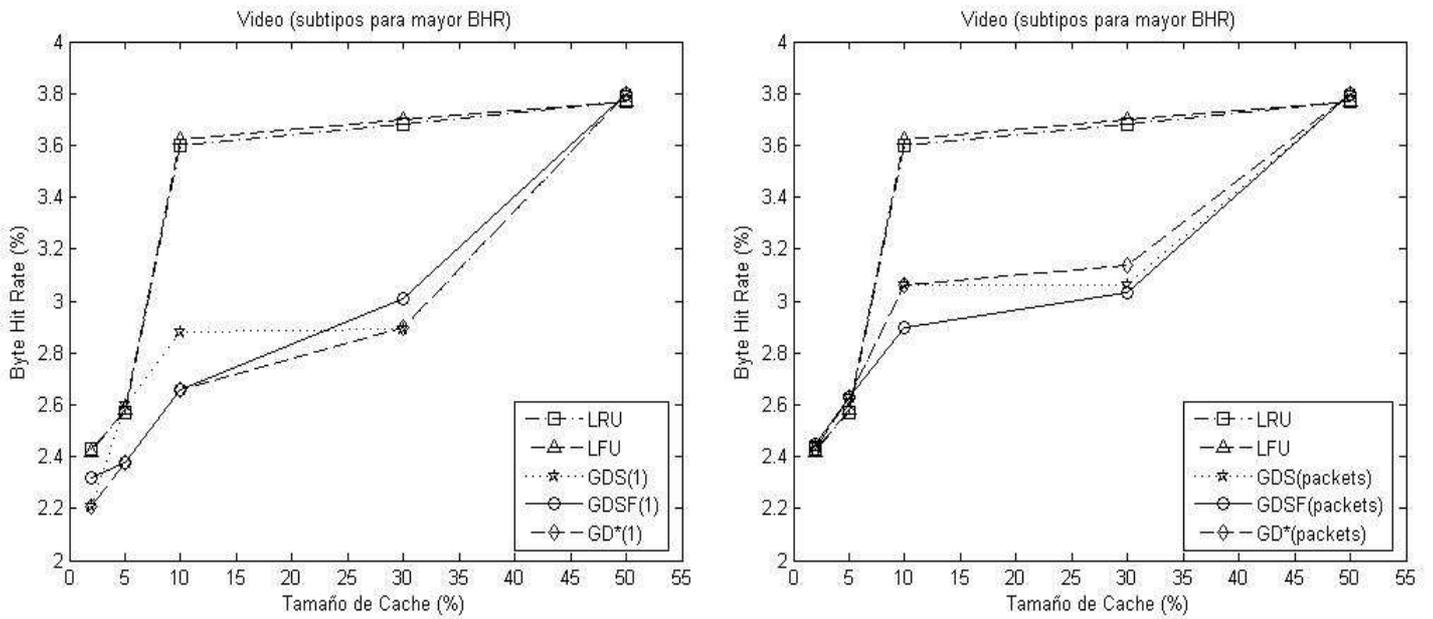
Por último, en la figura 4.17, aparecen los resultados obtenidos “para mayor BHR” de los subtipos de vídeo elegidos. En este caso, como se comprueba en la figura 4.17. (a), el HR obtenido para los subtipos “para mayor BHR” es inferior a la mitad del obtenido con los subtipos “para mayor HR” y aún menor si se compara con los resultados que se obtenían si se introducían en *cache* todos los subtipos. A diferencia de como ocurría hasta ahora LFU y LRU pasan a ser, en este caso, las políticas que proporcionan mejor HR para todos los tamaños, excepto para 50% de *cache*, donde se ven superadas levemente por las políticas Greedy. Curiosamente, las políticas Greedy proporcionan un mayor HR empleando función de coste *packets*, de ahí que se observe una mayor similitud en el rendimiento proporcionado por estas a LFU y LRU cuando se usa dicha función de coste.

Por otro lado, el BHR no es mejor que el conseguido hasta el momento si bien no sufre una caída tan importante como el HR. Esto se puede apreciar en la figura 4.17. (b). Como era la tendencia hasta el momento, LFU y LRU por este orden y con escasa diferencia, son las políticas que dan mejores resultados en cuanto al BHR. Las políticas de la familia Greedy ofrecen un rendimiento peor salvo para un tamaño de *cache* del 50%, caso en el que incluso superan mínimamente en rendimiento a LFU y LRU.

Observando todos los resultados obtenidos, podemos concluir que la política GD*(1) es la que mejores resultados ofrece para el HR guardando en *cache* todos los subtipos de vídeo (o los más habituales, ya que la diferencia de rendimiento que se daba entre ambos casos era mínima). Respecto al BHR, para tamaños de *cache* grandes (30 y 50%), la política LFU ofrece los mejores resultados. Para tamaños de *cache* pequeños, las políticas de la familia Greedy con función de coste “*packets*” son las que presentan un mejor rendimiento, a excepción del tamaño 10% de *cache* para el cuál, LRU será la que ofrezca mejores prestaciones cuando se guarden en *cache* los subtipos más habituales.



(a) HR para los subtipos para mayor BHR de Video con coste 1 (izquierda) y coste *packets* (derecha)



(b) BHR para los subtipos para mayor BHR de Video con coste 1 (izquierda) y coste *packets* (derecha)

Figura 4.17. HR (a) y BHR (b) para los subtipos para mayor BHR de Video

CAPÍTULO 5: Conclusiones y líneas futuras

En este capítulo se presentan las conclusiones obtenidas tras la realización de este Proyecto Fin de Carrera y se explica lo que ha sido necesario implementar y usar para poder llegar a dichas conclusiones. Además, se incluyen posibles líneas futuras de trabajo que se pueden seguir.

5.1. CONCLUSIONES FINALES

El objetivo de este Proyecto Fin de Carrera es la evaluación de distintas políticas de reemplazo asociadas a su comportamiento en las *caches* Web de los servidores *proxy*.

Para poder llevar a cabo dicho estudio, se ha implementado un emulador de *cache* de un servidor *proxy* con diferenciación de tipos de documento. Dicho emulador permite al usuario seleccionar o introducir todos los parámetros necesarios a la hora de hacer las

emulaciones y proporciona los resultados de dichas emulaciones en las métricas correspondientes con lo que, se pueden realizar las distintas comparativas entre las distintas políticas de reemplazo.

Antes de llevar a cabo las emulaciones se comenzó por determinar cuál era el tamaño de *cache* infinito para cada uno de los tipos de documentos bajo estudio, relativo a las trazas que se iban a utilizar.

Las políticas utilizadas a la hora de realizar el estudio fueron LFU, LRU, GDS, GDSF y GD*. Para las tres últimas se usaron dos funciones de coste distintas, “1” y “*packets*” ya que las dos primeras políticas (LRU y LFU) no emplean función de coste a la hora de determinar la prioridad de los documentos.

Así, para cada tipo de documento, se realizó la emulación del funcionamiento de la *cache* para los tamaños del 2, 5, 10, 30 y 50% de la *cache* infinita correspondiente a cada tipo de documento, guardando en *cache* todos los subtipos de documentos.

A continuación se analizó la composición de las muestras de tráfico para determinar qué grupo de subtipos aparecían con más frecuencia para cada uno de los tipos de documento estudiados. A dichos grupos se les denominó los “subtipos más habituales” y en base a ellos se realizaron las correspondientes emulaciones.

Por último, se estudió, de los subtipos más habituales, cuáles proporcionaban un mayor HR y BHR por separado y se eligió sólo a los que proporcionaban una aportación más significativa constituyendo los llamados “subtipos con mayor HR” y “subtipos con mayor BHR”, respectivamente. Con ellos se trató de observar si se conseguía mejorar la respectiva métrica, guardando tan solo en *cache* esos subtipos de documentos concretos.

Con los resultados obtenidos se realizaron una serie de gráficas, para facilitar el análisis de los datos extraídos de las emulaciones. En dichas gráficas se representó el rendimiento de las dos métricas estudiadas en este Proyecto Fin de Carrera, en función del tamaño de *cache*, para cada uno de los tipos de documentos y para cada una de las políticas de reemplazo utilizadas.

Por lo tanto, haciendo uso del emulador y comparando los resultados obtenidos, se ha llegado a las siguientes conclusiones:

1. Para los tipos de documento, texto y aplicaciones, la política de reemplazo LFU proporciona los mejores resultados, tanto en lo relativo al HR como al BHR y sin necesidad de hacer distinciones de subtipos de documentos.
2. Respecto a los documentos de audio, GD*(1) es la política que proporciona unos mejores resultados en lo referente al HR si bien, para tamaños del 30% o superiores se han de usar todos los subtipos de audio, mientras que para tamaños del 10% e inferiores se consiguen mejores resultados eligiendo los “subtipos para mayor HR”.
En cuanto al BHR, la política LFU incluyendo todos los subtipos es la que ofrece un mejor rendimiento.
3. Para documentos de vídeo, GD*(1) incluyendo todos los subtipos se presenta como la política idónea para maximizar el HR.
En cuanto al BHR, para tamaños grandes (30% de *cache* y superiores), LFU proporciona el mayor rendimiento. Para los tamaños más pequeños, GDSF(*Packets*) guardando los subtipos más habituales es la mejor opción, a excepción del tamaño 10% de *cache* para el cuál, LRU es la que ofrece un mayor rendimiento.
4. En el caso de las imágenes, la política de reemplazo GDSF(1) es la que permite conseguir un mayor HR cuando se guardan en *cache* los subtipos más habituales de imágenes.
En lo relativo al BHR, los mejores resultados se consiguen para la política LFU al guardarse en *cache* todos los subtipos, si bien al guardarse los “subtipos más habituales” apenas existen variaciones.

5.2. LÍNEAS FUTURAS

Entre las posibles líneas sobre las que se puede seguir trabajando en el futuro se puede destacar:

1. Gracias al diseño empleado para la implementación del emulador, es posible incrementar de forma sencilla, el número de políticas de reemplazo con las que pueda trabajar el emulador. Así, se podrían incluir otras políticas de reemplazo como LFU-DA (LFU Dynamic Aging) [Lindemann'02] o Hybrid [Wooster'97], con la consiguiente posibilidad de realizar estudios comparativos más completos empleando mayor número de políticas.
2. Otro aspecto a considerar sería permitir la selección o introducción de un mayor número de parámetros desde la ventana principal de la aplicación. De esta forma, se podría especificar de forma más detallada las características buscadas en la emulación a ejecutar. Así, por ejemplo, se podría permitir fijar el tamaño máximo o mínimo de los documentos almacenables en *cache*.
3. Haciendo uso de muestras de tráfico con más información, se podría conseguir que al finalizar las emulaciones se obtuviera más información en los archivos de resultados que permitiese realizar un estudio estadístico más profundo. De este modo, se podría por ejemplo estudiar la influencia de las políticas de reemplazo en otros aspectos como la latencia o el número de saltos necesarios a la hora de acceder a un determinado documento.
4. También se podría modificar la aplicación para permitir seleccionar al usuario más de un tipo de documento simultáneamente, al hacerse la emulación, y poder aplicar una política de reemplazo distinta a cada uno de los tipos de documentos elegidos.

Bibliografía

- [Abrams'95] M. Abrams , C. R. Standridge, G. Abdulla, S. Williams y E. A. Fox. "Caching Proxies: Limitations and Potentials". Julio, 1995.
- [Barford'99] P. Barford, A. Bestavros, A. Bradley y M. Crovella. "Changes in Web Client Access Patterns: Characteristics and Caching Implications". World Wide Web, páginas 15-28. 1999.
- [Bobadilla'03] J. Bobadilla y A.Sancho. *Java: a través de ejemplos*. Madrid, Ra-Ma, 2003.
- [Breslau'99] Breslau, Lee, Pei Cao, Li Fan, G. Phillips y S. Shenker. "Web Caching and Zipf-like Distributions: Evidence and Implications". En Proceedings of Infocomm'99. Abril 1999.

- [Cao-Irani'97] P. Cao y S. Irani. "Cost Aware WWW *Proxy* Caching Algorithms". En Proceedings del Simposio USENIX sobre Tecnología de Internet y Sistemas. Diciembre 1997.
- [Cherkasova'98] L. Cherkasova. "Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy". Hewlett Packard. Noviembre 1998.
- [Coffman'73] E. G. Coffman y P.J. Denning, *Operating Systems Theory*. Prentice-Hall, 1973.
- [Deitel'04] H. M. Deitel. *Cómo programar en JAVA*. Traducción, Alfonso Vidal; revisión técnica, Gabriela Azucena Campos, México, Pearson Education, 2004.
- [Duska'97] Duska, Bradley, D. Marwood y M. Feely. "The Measured Access Characteristics of World-Wide-Web Client *Proxy Caches*". En el Simposio USENIX sobre las tecnologías y sistemas de Internet. Diciembre 1997.
- [Eckel'04] B. Eckel. *Piensa en Java*. Traducción, Jorge González; revisión técnica, Javier Parra, Prentice Hall, 2004.
- [IRCACHE] <http://www.ircache.net/>.
- [Jin'00] S. Jin y A. Bestavros. "Greedy Dual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams". Computer Communications Special Issue on 5th Web Caching and Content Delivery Workshop, 22, 174-183, 2000.
- [Khayari'03] R. A. Khayari, M. Best y A. Lehmann. "Impact of Document Types on the Performance of Caching Algorithms in WWW Proxies: A Trace Driven Simulation Study". Department of Computer Science, University of the Armed Forces, Munich, 2003.

- [Lindemann'02] C. Lindemann y O.P. Waldhorst, "Evaluating the Impact of Different Document Types on the Performance of Web *Cache* Replacement Schemes". En *IEEE International Conference on Dependable Systems and Networks (DSN'02)*, páginas 717-726, Washington, D.C. (EEUU), Junio 2002.
- [Luotonen'94] A. Luotonen y K. Altis. "World-Wide Web Proxies". Abril, 1994.
- [NLANR] National Laboratory for Applied Network Research, <http://www.nlanr.net/>.
- [Nottingham'04] M. Nottingham. "Caching Tutorial for Web Authors and Webmasters". http://www.mnot.net/cache_docs/, Febrero, 2004.
- [Piattini'96] M. G. Piattini, J. A. Calvo-Manzano, J. Cervera y L. Fernández. *Análisis y diseño detallado de Aplicaciones informáticas de gestión*. Ra-Ma, 1996.
- [RFC 2518'99] Y. Goland, E. Whitehead, A. Faizi, S. Carter y D. Jensen "HTTP Extensions for Distributed Authoring -- WEBDAV". RFC 2518, Febrero 1999.
- [RFC 2616'99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach y T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1". RFC 2616, Junio 1999.
- [SQUID] <http://www.squid-cache.org/>.
- [Wessels'01] D. Wessels, *Web Caching, reducing network traffic*. Sebastopol (EEUU), O'Reilly & Associates, 2001.

- [Wooster'97] R. P. Wooster y M. D. Abrams. “*Proxy Caching that Estimates Page Load Delays*”. *Computer Networks*, 29(8-13): 977-986, Septiembre 1997.
- [SUN] Sun Microsystems, <http://www.sun.com/>.
- [You91b'94] N. Young. “The k-server dual and loose competitiveness for paging”. En *Algorithmic*, Junio 1994, vol. 11, páginas 525-41. Versión rescrita de “Online caching as *cache size varies*”, en el segundo Simposio anual ACM-SIAM de algoritmos discretos, 241-250, 1991.
- [Yu'03] F. Yu, Q. Zhang, W. Zhu y Y. Zhang. “*Network-Adaptative Cache Management Schemes for Mixed Media*”. Microsoft Research, China, 2003.

Apéndice A. Diagramas UML.

A continuación se muestran los diagramas UML más importantes del emulador de *cache proxy* diseñado.

Se han seleccionado los diagramas UML de las clases “*emulcacheproxy*”, “*emulador*”, “*MiLista*” y “*ListaGDS*” como los más representativos del emulador diseñado.

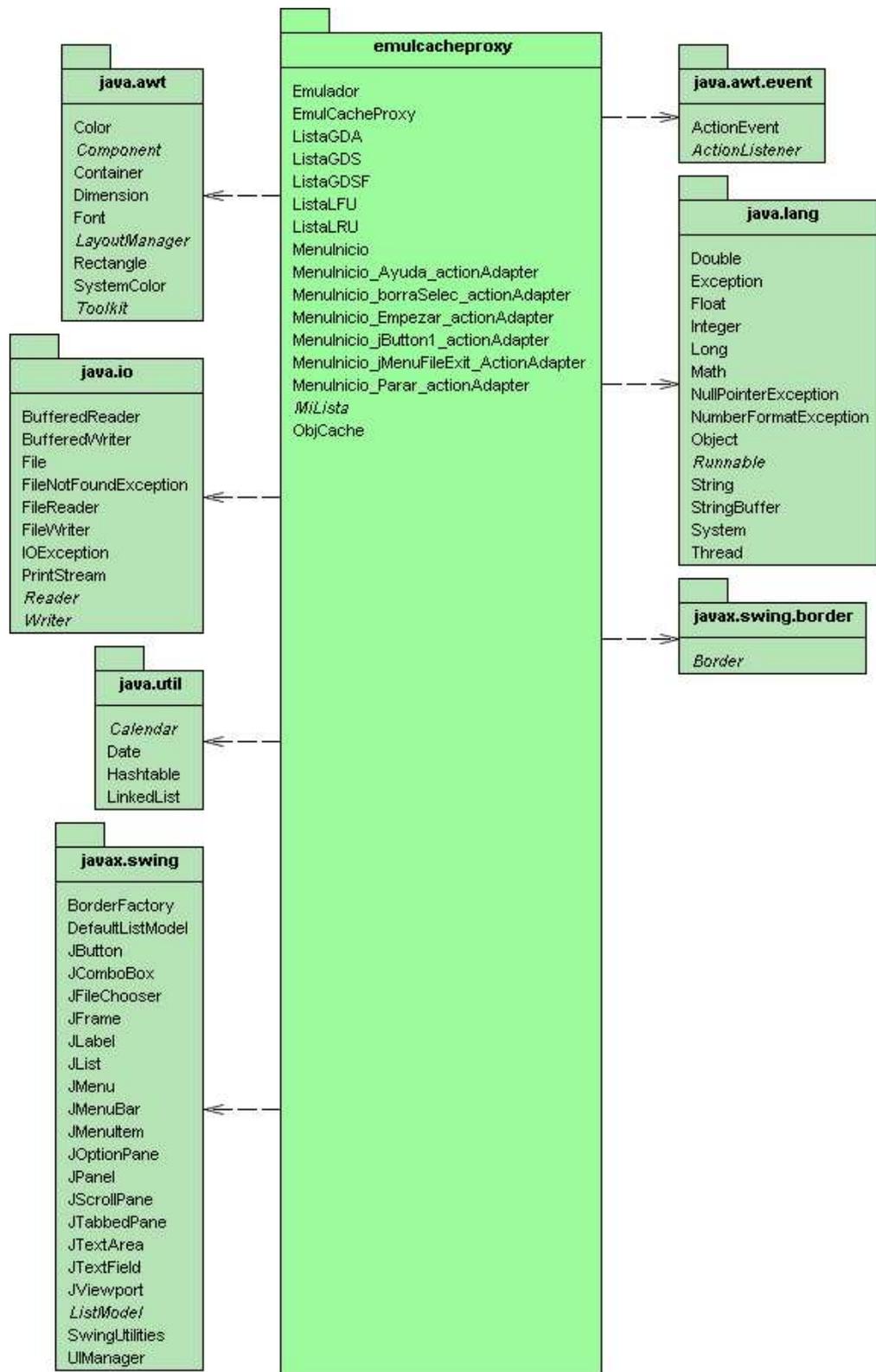


Figura A.1. Diagrama UML de “emulcacheproxy”

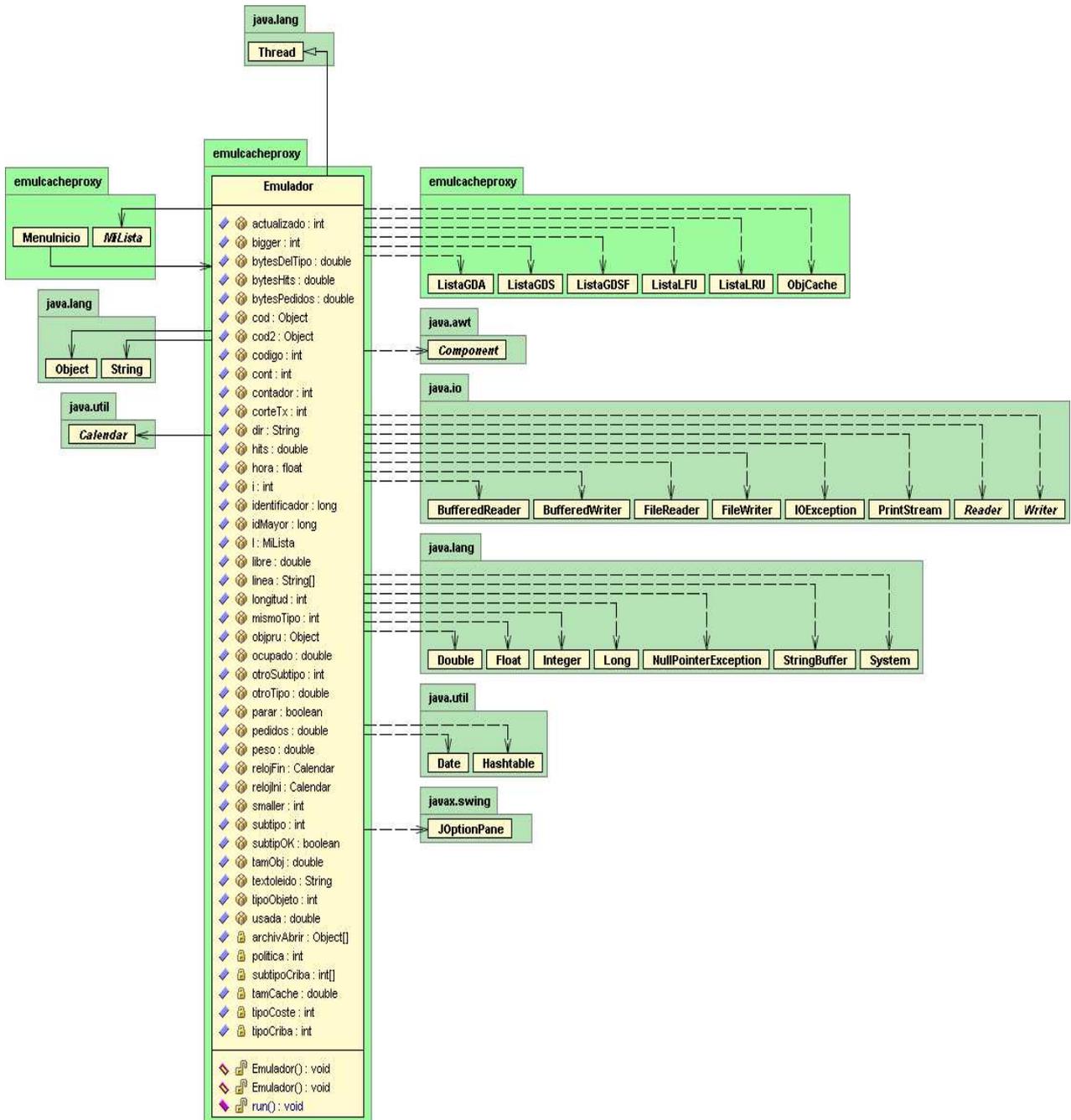


Figura A.2. Diagrama UML de “Emulador”

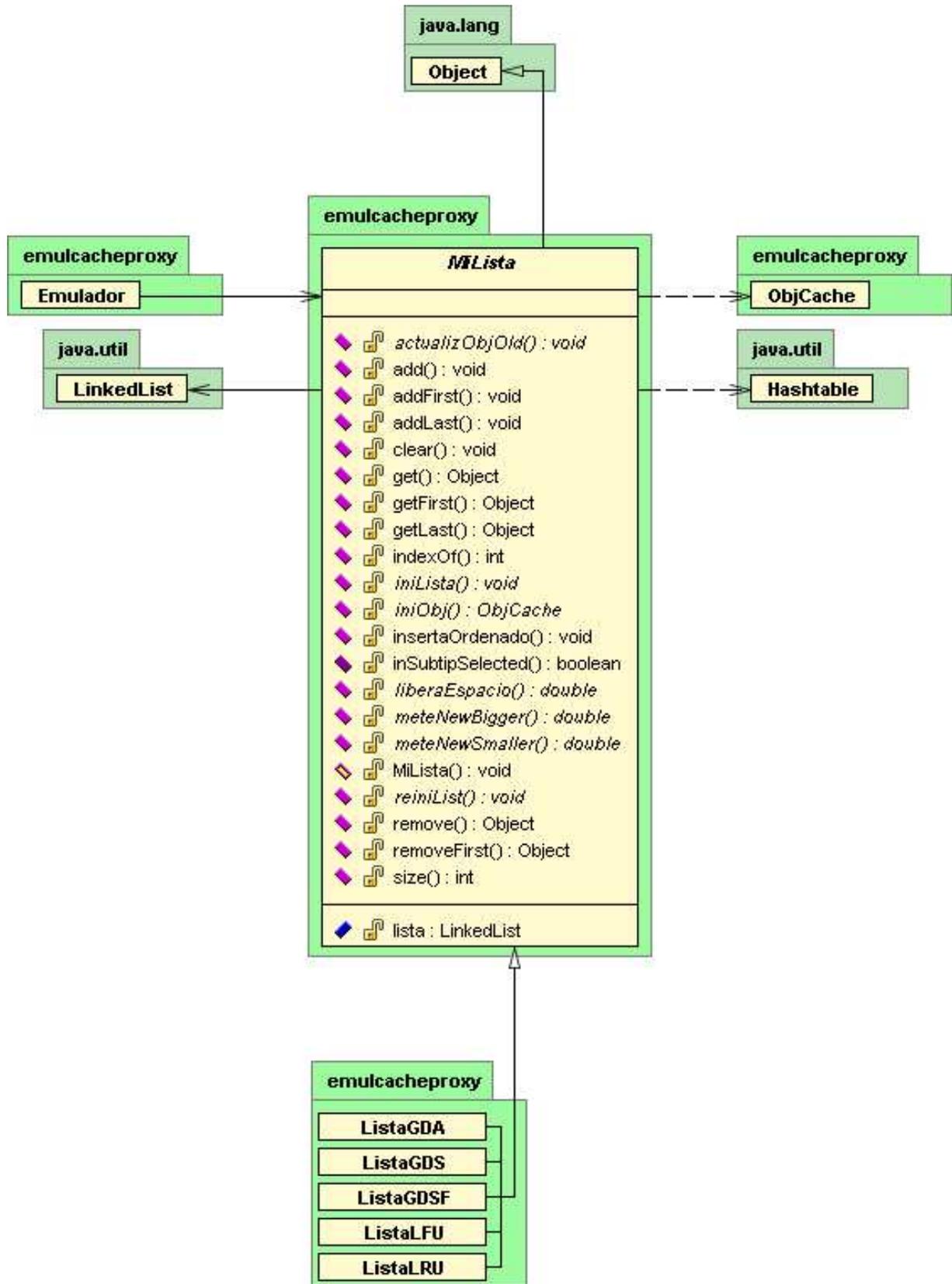


Figura A.3. Diagrama UML de “MiLista”

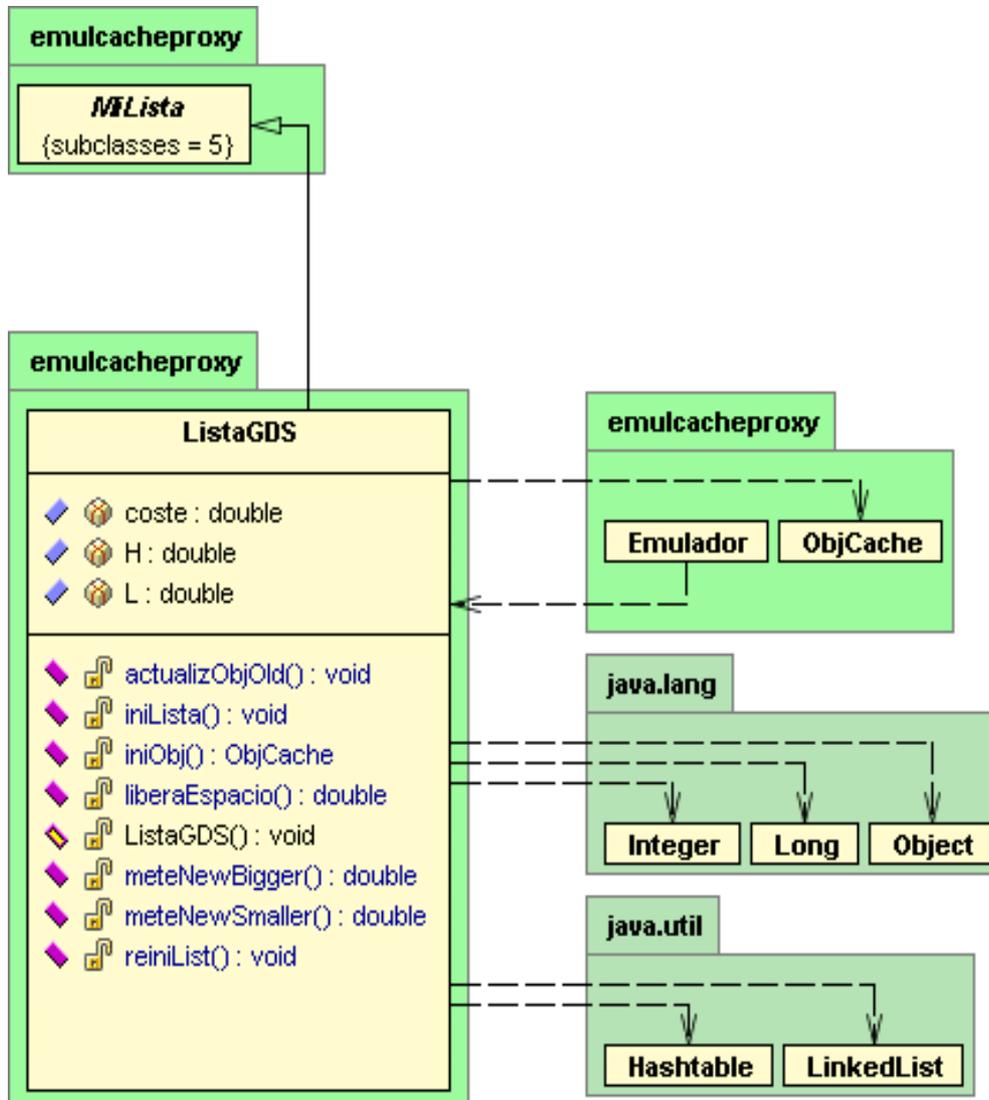


Figura A.4. Diagrama UML de “ListaGDS”

Apéndice B. Ejemplo de verificación del funcionamiento del emulador.

En este apéndice se muestra un ejemplo de cómo se llevó a cabo la verificación del correcto funcionamiento del emulador para cada una de las políticas de reemplazo evaluadas.

Muestra de tráfico (con 45 líneas), utilizada para comprobar el correcto funcionamiento del emulador.

```
1086566414.234 360 0 0 79
1086566414.341 3485 1 4 14
1086566414.365 1029 2 4 6
1086566414.341 24900 88 4 14
1086566414.366 102900 50 4 19
1086566414.367 1029 51 4 19
1086566414.368 6679 3 4 6
1086566414.234 360 0 0 79
1086566414.365 1029 2 4 6
1086566414.341 3485 100 4 14
1086566414.370 3485 100 4 14
1086566414.370 3485 100 4 14
1086566414.234 360 0 0 79
1086566414.381 1125 4 2 1
1086566414.383 6679 3 4 6
1086566414.234 360 0 0 79
1086566414.385 1029 2 4 6
```

1086566414.388 6679 3 4 6
1086566414.390 420 5 4 6
1086566414.234 360 0 0 79
1086566414.385 1029 2 4 6
1086566414.395 420 5 4 6
1086566414.421 7358 6 2 1
1086566414.422 6974 7 2 1
1086566414.421 7358 6 2 1
1086566414.459 669 8 4 6
1086566414.475 524 9 4 6
1086566414.459 669 8 4 6
1086566414.561 6495 10 4 6
1086566414.712 459 11 4 6
1086566414.712 459 11 4 6
1086566415.076 476 12 2 1
1086566415.231 1719 13 4 1
1086566416.358 6686 3 4 6
1086566416.767 2273 14 2 1
1086566416.767 22073 99 2 1
1086566417.725 12537 15 4 4
1086566418.190 6686 3 4 6
1086566418.212 6016 16 2 3
1086566418.259 408 17 2 1
1086566418.306 141 18 2 1
1086566418.306 141 18 2 1
1086566418.619 373 19 2 1
1086566418.307 141 18 2 1
1086566418.654 44 20 2 1

A continuación se muestran los mensajes obtenidos por pantalla como resultado de la emulación. A modo de recordatorio, se indica que se empleó una política de reemplazo LFU, se eligió un tamaño de *cache* de 25 Kbytes y que los documentos almacenados en *cache* fueran de Texto, de los subtipos más habituales (en concreto, css, html, js:javascript:x-javascript, plain y xml).

El documento con identificador 0 es de otro tipo
La petición leída, con identificador 1, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter el primer objeto en la lista de tamaño: 3485.0
Al meter el primer elemento de la lista, lo que queda libre es: 21515.0
La petición leída, con identificador 2, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 2 en la lista de tamaño: 1029.0
Queda espacio para meterlo
El documento nuevo con peso 1.0, entra en la posición 1 de la lista tras otros de igual peso
Tras meter el nuevo objeto la lista tiene 2 objetos
Tamaño de la tabla: 2
Al meterlo el espacio libre que queda es: 20486.0
La petición leída, con identificador 88, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 88 en la lista de tamaño: 24900.0

Se borra todo porque el objeto es más grande que todos los almacenados
La petición leída, con identificador 50, coincide en el tipo de documento que es 4
El objeto es más grande que la *Cache* y no se guarda en ella
La petición leída, con identificador 51, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? false
El subtipo del documento con identificador 51 no coincide con los seleccionados
La petición leída, con identificador 3, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 3 en la lista de tamaño: 6679.0
Hay que sacar documentos para poder meter el nuevo
La lista tiene 1 objetos ANTES de liberar espacio para el nuevo
No hay espacio, al sacar el identificador 88 queda : 25000.0 y 0 elementos aún en la lista
Tamaño de la lista: 0
Tamaño de la tabla: 0
Una vez sacados los elementos necesarios ha quedado libre: 25000.0
Tras meter el nuevo objeto la lista tiene 1 documentos, y la tabla 1
El documento con identificador 0 es de otro tipo
La petición leída, con identificador 2, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 2 en la lista de tamaño: 1029.0
Queda espacio para meterlo
El documento nuevo con peso 1.0, entra en la posición 1 de la lista tras otros de igual peso
Tras meter el nuevo objeto la lista tiene 2 objetos
Tamaño de la tabla: 2
Al meterlo el espacio libre que queda es: 17292.0
La petición leída, con identificador 100, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 100 en la lista de tamaño: 3485.0
Queda espacio para meterlo
El documento con peso 1.0, entra en la posición 2 de la lista tras otros de igual peso
Tras meter el nuevo objeto la lista tiene 3 objetos
Tamaño de la tabla: 3
Al meterlo el espacio libre que queda es: 13807.0
La petición leída, con identificador 100, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 2.0
El documento a insertar con peso 2.0, entra en la posición 2 sin haber otro de igual peso
La petición leída, con identificador 100, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 3.0
El documento a insertar con peso 3.0, entra en la posición 2 sin haber otro de igual peso
El documento con identificador 0 es de otro tipo
El documento con identificador 4 es de otro tipo
La petición leída, con identificador 3, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 2.0
El documento a insertar con peso 2.0, entra en la posición 1 sin haber otro de igual peso
El documento con identificador 0 es de otro tipo
La petición leída, con identificador 2, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 2.0
El documento con peso 2.0, entra en la posición 1 de la lista tras otros de igual peso
La petición leída, con identificador 3, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 3.0
El documento nuevo con peso 3.0, entra en la posición 2 de la lista tras otros de igual peso

La petición leída, con identificador 5, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 5 en la lista de tamaño: 420.0
Queda espacio para meterlo
El documento a insertar con peso 1.0, entra en la posición 0 sin haber otro de igual peso
Tras meter el nuevo objeto la lista tiene 4 objetos
Tamaño de la tabla: 4
Al meterlo el espacio libre que queda es: 13387.0
El documento con identificador 0 es de otro tipo
La petición leída, con identificador 2, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 3.0
El documento con peso 3.0, entra en la posición 3 de la lista tras otros de igual peso
La petición leída, con identificador 5, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 2.0
El documento a insertar con peso 2.0, entra en la posición 0 sin haber otro de igual peso
El documento con identificador 6 es de otro tipo
El documento con identificador 7 es de otro tipo
El documento con identificador 6 es de otro tipo
La petición leída, con identificador 8, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 8 en la lista de tamaño: 669.0
Queda espacio para meterlo
El documento a insertar con peso 1.0, entra en la posición 0 sin haber otro de igual peso
Tras meter el nuevo objeto la lista tiene 5 objetos
Tamaño de la tabla: 5
Al meterlo el espacio libre que queda es: 12718.0
La petición leída, con identificador 9, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 9 en la lista de tamaño: 524.0
Queda espacio para meterlo
El documento con peso 1.0, entra en la posición 1 de la lista tras otros de igual peso
Tras meter el nuevo objeto la lista tiene 6 objetos
Tamaño de la tabla: 6
Al meterlo el espacio libre que queda es: 12194.0
La petición leída, con identificador 8, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 2.0
El documento con peso 2.0, entra en la posición 2 de la lista tras otros de igual peso
La petición leída, con identificador 10, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 10 en la lista de tamaño: 6495.0
Queda espacio para meterlo
El documento con peso 1.0, entra en la posición 1 de la lista tras otros de igual peso
Tras meter el nuevo objeto la lista tiene 7 objetos
Tamaño de la tabla: 7
Al meterlo el espacio libre que queda es: 5699.0
La petición leída, con identificador 11, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 11 en la lista de tamaño: 459.0
Queda espacio para meterlo
El documento con peso 1.0, entra en la posición 2 de la lista tras otros de igual peso
Tras meter el nuevo objeto la lista tiene 8 objetos
Tamaño de la tabla: 8
Al meterlo el espacio libre que queda es: 5240.0

La petición leída, con identificador 11, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 2.0
El documento con peso 2.0, entra en la posición 4 de la lista tras otros de igual peso
El documento con identificador 12 es de otro tipo
La petición leída, con identificador 13, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 13 en la lista de tamaño: 1719.0
Queda espacio para meterlo
El documento con peso 1.0, entra en la posición 2 de la lista tras otros de igual peso
Tras meter el nuevo objeto la lista tiene 9 objetos
Tamaño de la tabla: 9
Al meterlo el espacio libre que queda es: 3521.0
La petición leída, con identificador 3, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 4.0
El documento a insertar con peso 4.0, entra en la posición 8 sin haber otro de igual peso
El documento con identificador 14 es de otro tipo
El documento con identificador 99 es de otro tipo
La petición leída, con identificador 15, coincide en el tipo de documento que es 4
Coincide en tipo, ¿también en subtipo? true
Quiero meter objeto nuevo con identificador 15 en la lista de tamaño: 12537.0
Hay que sacar documentos para poder meter el nuevo pero NO TODOS
La lista tiene 9 objetos ANTES de liberar espacio para el nuevo
No hay espacio, al sacar el identificador 9 queda : 4038.0 y 8 elementos aún en la lista
La lista tiene 8 objetos ANTES de liberar espacio para el nuevo
No hay espacio, al sacar el identificador 10 queda : 10533.0 y 7 elementos aún en la lista
La lista tiene 7 objetos ANTES de liberar espacio para el nuevo
No hay espacio, al sacar el identificador 13 queda : 12252.0 y 6 elementos aún en la lista
La lista tiene 6 objetos ANTES de liberar espacio para el nuevo
No hay espacio, al sacar el identificador 5 queda : 12672.0 y 5 elementos aún en la lista
Tamaño de la lista: 5
Tamaño de la tabla: 5
Una vez sacados los elementos necesarios ha quedado libre: 12672.0
El documento a insertar con peso 1.0, entra en la posición 0 sin haber otro de igual peso
Tras meter el nuevo objeto la lista tiene 6 documentos, y la tabla 6
La petición leída, con identificador 3, coincide en el tipo de documento que es 4
Se ha producido un acierto en *cache*
El documento ahora tiene peso: 5.0
El documento a insertar con peso 5.0, entra en la posición 5 sin haber otro de igual peso
El documento con identificador 16 es de otro tipo
El documento con identificador 17 es de otro tipo
El documento con identificador 18 es de otro tipo
El documento con identificador 18 es de otro tipo
El documento con identificador 19 es de otro tipo
El documento con identificador 18 es de otro tipo
El documento con identificador 20 es de otro tipo

////////////////////////////////////
Fin de la emulación.

Al finalizar el procesamiento de las muestras de tráfico, se obtienen una serie de datos estadísticos a cerca del proceso:

El tamaño de *cache* es 25000.0 Bytes
Política de reemplazo: LFU
El tipo de objeto a guardar es 4
Los subtipos seleccionados son 1, 4, 5, 6 y 14
Tamaño de la lista: 6
Tamaño de la tabla: 6
Numero de veces que no se ha almacenado un objeto: 1
Numero de veces que no se procesado un archivo por ser de tipo distinto: 20.0
Veces que no actualicé objeto por corte de transmisión: 0
Número de aciertos: 11.0
Número total de peticiones: 45.0
Número de bytes acertados: 37306.0
Número total de bytes pedidos: 261366.0
Hit Ratio (%): 24.444444444444443
Byte Hit Ratio(%): 14.273470918176045
////////////////////////////////////

Tras acabar la emulación, también se presenta el contenido final de la lista y la tabla Hash para que se pueda comprobar que en ambos existen los mismos documentos.

Contenido de la Lista al finalizar la emulación

Fecha: 1.0865664E9
Tamaño (Bytes): 12537.0
Identificador: 15
Tipo de objeto: 4
Subtipo del objeto: 4
Peso según la política: 1.0

Fecha: 1.0865664E9
Tamaño (Bytes): 669.0
Identificador: 8
Tipo de objeto: 4
Subtipo del objeto: 6
Peso según la política: 2.0

Fecha: 1.0865664E9
Tamaño (Bytes): 459.0
Identificador: 11
Tipo de objeto: 4
Subtipo del objeto: 6
Peso según la política: 2.0

Fecha: 1.0865664E9
Tamaño (Bytes): 3485.0
Identificador: 100
Tipo de objeto: 4
Subtipo del objeto: 14
Peso según la política: 3.0

Fecha: 1.0865664E9
Tamaño (Bytes): 1029.0
Identificador: 2
Tipo de objeto: 4
Subtipo del objeto: 6

Peso según la política: 3.0

Fecha: 1.0865664E9
Tamaño (Bytes): 6686.0
Identificador: 3
Tipo de objeto: 4
Subtipo del objeto: 6
Peso según la política: 5.0

Contenido de la Tabla Hash al finalizar la emulación

Fecha: 1.0865664E9
Tamaño (Bytes): 12537.0
Identificador: 15
Tipo de objeto: 4
Subtipo del objeto: 4
Peso según la política: 1.0

Fecha: 1.0865664E9
Tamaño (Bytes): 669.0
Identificador: 8
Tipo de objeto: 4
Subtipo del objeto: 6
Peso según la política: 2.0

Fecha: 1.0865664E9
Tamaño (Bytes): 459.0
Identificador: 11
Tipo de objeto: 4
Subtipo del objeto: 6
Peso según la política: 2.0

Fecha: 1.0865664E9
Tamaño (Bytes): 3485.0
Identificador: 100
Tipo de objeto: 4
Subtipo del objeto: 14
Peso según la política: 3.0

Fecha: 1.0865664E9
Tamaño (Bytes): 1029.0
Identificador: 2
Tipo de objeto: 4
Subtipo del objeto: 6
Peso según la política: 3.0

Fecha: 1.0865664E9
Tamaño (Bytes): 6686.0
Identificador: 3
Tipo de objeto: 4
Subtipo del objeto: 6
Peso según la política: 5.0

Apéndice C. Codificación de tipos y subtipos de documentos.

En este apéndice, se recogen, para cada uno de los tipos de documentos, el conjunto de subtipos considerados así como la codificación que le fue asociada.

0 – Aplicación

0	aom
	binary
	cab : x-cabinet : x-cabinet-win32-x86
	font : font-eot : font-tdpfr
	forcedownload
5	futuresplash
	gzip : x-gzip
	hta
	ico
	image
10	jar : java-archive : x-java-archive
	java
	javascript : x-javascript : x-javascrip
	java-vm : x-java-vm
	mac-binhex40
15	metastream

msword
 octet_stream : octet-stream : x-octet-stream
 pdf
 pkix-crl
 20 postscript
 powerpoint : vnd.ms-powerpoint : x-ppt
 rar
 RealNetworksUpgrade
 rtf
 25 save
 smil
 unknown
 vnd.mozilla.xul+xml
 vnd.ms.wms-hdr.asfv1
 30 vnd.ms-excel
 vnd.netfpx
 vnd.rn-realmedia
 vnd.rn-rn_game_package
 vnd.rn-rn_music_package
 35 vndms-pkiseccat
 x-bittorrent
 x-bzip2
 x-cdf
 x-compress
 40 x-compressed
 x-director
 x-executable
 x-flash : x-shockwave : x-shockwave-flash : x-shockwave-flash2-preview
 xhtml+xml
 45 x-httpd-php
 x-incredimail
 x-ipix
 x-java
 x-java-applet
 50 x-java-byte-code
 x-java-jnlp-file
 x-javascript-config
 x-java-vm
 xml
 55 x-mms-framed
 x-msdos-program
 x-msdownload
 x-msmetafile
 x-ms-wmz
 60 x-musicnotes
 x-netcdf
 x-ns-proxy-autoconfig
 x-perl
 x-photoparade
 65 x-pmxy
 x-pointplus
 x-redhat-package-manager
 x-rpm
 x-stuffit
 70 x-swf
 x-tar

x-troff
x-wais-source
x-webshots
75 x-www-form-urlencoded\n\n
x-www-urlformencoded
x-x509-ca-cert
x-xpinstall
zip : x-zip : x-zip-compressed
80 desconocido

1 – Audio

0 basic
midi : mid : x-mid : x-midi
mpeg : x-mpeg
wav : x-wav
x-aiff
5 x-mpegurl
x-ms-wax
x-ms-wma
x-ms-wmv
x-pn-realaudio : x-pn-realaudio : x-realaudio
10 x-pn-realaudio-plugin
x-scpls
desconocido

2 – Imagen

0 bmp :x-bitmap : x-bmp : x-ms-bmp
gif
ico : icon : icons : x-ico : x-icon
jpg : jpeg : pjpeg
nids : nids-mosaic
5 png : x-png
swf
tiff
vnd
vnd.dwg
10 vnd.nok-oplogo-color
vnd.rn-realpix
vnd.wap.bmp
x-binary
x-corelphotopaint
15 x-guffaw
x-hotmedia
x-portable-graymap
desconocido

3 – Mensaje

rfc822

4 – Texto

0 cfg
 css
 gif
 htmd
 html
5 js : javascript : x-javascript
 plain
 rdf
 rtf : richtext
 stylesheet
10 unknown
 vbscript
 vnd.wap.wml
 x-component
 xml
15 x-server-parsed-html
 x-tcl
 desconocido

5 – Vídeo

0 avi
 mpeg : x-mpeg
 quicktime
 unknown
 wmv : x-ms-wmv
5 ms-asf
 x-ms-vídeo : x-msvídeo
 x-ms-wvx
 desconocido