

Índice de Contenidos

Índice de Contenidos	i
Índice de Figuras	v
Índice de Tablas	ix
Relación de Acrónimos	xi

CAPÍTULO 1: Introducción	1
1.1. <i>CACHÉ PROXY</i> WEB	1
1.2. POLÍTICAS DE ACCESO Y DE REEMPLAZO	6
1.2.1. POLÍTICAS DE REEMPLAZO	7
1.2.2. POLÍTICAS DE ACCESO	8
1.3. OBJETIVO DEL PROYECTO Y CONTENIDOS	9

CAPÍTULO 2: Descripción del emulador empleado y mejoras implementadas	11
2.1. DESCRIPCIÓN DEL EMULADOR	11
2.1.1. LENGUAJE EMPLEADO EN SU DESARROLLO	11
2.1.2. INTERFAZ DE USUARIO DEL EMULADOR	14
2.1.2.1. Introducción de parámetros de emulación	14

2.1.2.2. Control de la emulación y botón de ayuda	17
2.1.2.3. Resultados de la emulación	17
2.1.3. ESTRUCTURA INTERNA DEL EMULADOR	20
2.2. OPTIMIZACIÓN DEL EMULADOR	23
2.2.1. ESTRUCTURAS DE ALMACENAMIENTO EN JAVA	23
2.2.2. ANÁLISIS DEL RENDIMIENTO DE LA APLICACIÓN, USO DE <i>JPROFILER</i>	28
2.3. NUEVA POLÍTICA DE REEMPLAZO: LFU-DA	36
2.4. NUEVAS MÉTRICAS IMPLEMENTADAS	42
2.4.1. <i>NOT UNIQUE HIT RATIO</i> (NUHR) Y <i>NOT UNIQUE BYTE HIT RATIO</i> (NUBHR)	43
2.4.2. <i>ACCESS CONTROL HIT RATIO</i> (ACHR) Y <i>ACCESS CONTROL BYTE HIT RATIO</i> (ACBHR)	48
2.5. PROCESAMIENTO ESTRUCTURADO POR LOTES	49
2.5.1. LENGUAJES DE MARCAS	50
2.5.2. DTD (<i>DOCUMENT TYPE DEFINITION</i>)	50
2.5.3. XML (<i>EXtensible Markup Language</i>)	51
2.5.4. IMPLEMENTACIÓN DEL MÓDULO XML EN EL EMULADOR	53
2.5.5. ESTRUCTURA DEL FICHERO XML	56
CAPÍTULO 3: Caracterización del tráfico	59
3.1. MUESTRAS DE TRÁFICO EMPLEADAS	59
3.2. NÚMERO DE PETICIONES Y VOLUMEN DE DATOS PARA CADA TIPO DE DOCUMENTO	61
3.3. LOCALIDAD TEMPORAL: POPULARIDAD DE LOS DOCUMENTOS Y CORRELACIÓN TEMPORAL	63
3.4. RELACIÓN ENTRE LA FRECUENCIA Y LA DISTANCIA ENTRE PETICIONES CON EL TAMAÑO DE LOS DOCUMENTOS	68
CAPÍTULO 4: Rendimiento de la <i>caché</i> según la política de acceso elegida	73
4.1. POLÍTICA DE ACCESO BASADA EN EL ESTABLECIMIENTO DE UN UMBRAL	73
4.1.1. CÁLCULO DEL FACTOR ASOCIADO A CADA DOCUMENTO	73
4.1.2. RESULTADOS	74
4.2. POLÍTICA DE ACCESO BASADA EN PERMITIR EL ACCESO DE LOS SUBTIPOS MÁS SOLICITADOS	78

4.3. POLÍTICA DE ACCESO COMBINADA: ADMISIÓN DE DOCUMENTOS DE SUBTIPOS MÁS SOLICITADOS QUE SUPEREN UN UMBRAL	82
CAPÍTULO 5: Conclusiones y líneas futuras	87
5.1. CONCLUSIONES FINALES	87
5.2. LÍNEAS FUTURAS	89
Bibliografía	91
Apéndice A. Codificación de los subtipos de documentos	95
Apéndice B. Diagramas UML	99

Índice de Figuras

Figura 1.1. Esquema de acceso a Internet a través de servidor proxy para el servicio WWW	5
Figura 2.1. Interfaz de usuario de la aplicación.....	14
Figura 2.2. Ventana de carga de los archivos que contienen las muestras	15
Figura 2.3. Subtipos seleccionables para ser procesados del tipo Imagen	17
Figura 2.4. Mensajes de error	18
Figura 2.5. Estado de la emulación con entrada XML	18
Figura 2.6. Resultados de la emulación.....	19
Figura 2.7. Fichero de salida XML.....	20
Figura 2.8. Diagrama de flujo de “Emuladornuevo”	22
Figura 2.9. Jerarquía de las interfaces del marco de colecciones	24
Figura 2.10. Jerarquía de las clases del marco de colecciones	25
Figura 2.11. Consumo de CPU del emulador por métodos antes de la optimización (LRU)	30
Figura 2.12. Consumo de CPU del emulador por métodos después de la optimización (LRU)	31

Figura 2.13. Evolución del tiempo de ejecución en función del tamaño de la caché para LRU	32
Figura 2.14. Consumo de CPU del emulador por métodos antes de la optimización (GD)	33
Figura 2.15. Consumo de CPU del emulador por métodos después de la optimización (GD*).....	34
Figura 2.16. Tiempo de ejecución para distintos tamaños de caché para LRU.....	34
Figura 2.17. Tiempo de ejecución para distintos tamaños de caché para LFU	35
Figura 2.18. Tiempo de ejecución para distintos tamaños de caché para GDS.....	35
Figura 2.19. Tiempo de ejecución para distintos tamaños de caché para GDSF	36
Figura 2.20. Tiempo de ejecución para distintos tamaños de caché para GD*	36
Figura 2.21. HR para los algoritmos LRU, LFU y LFU-DA en función del tamaño de caché	38
Figura 2.22. BHR para los algoritmos LRU, LFU y LFU-DA en función del tamaño de caché	38
Figura 2.23. Mejora del parámetro RechazadosOk para LRU sin aciertos entre peticiones.....	46
Figura 2.24. Mejora del parámetro RechazadosOk para LRU con aciertos entre peticiones.....	46
Figura 2.25. Mejora del parámetro RechazadosOk para LFU sin aciertos entre peticiones.....	47
Figura 2.26. Mejora del parámetro RechazadosOk para LFU con aciertos entre peticiones.....	48
Figura 2.27. Fichero XML mostrado por el editor de Altova XMLSpy	54
Figura 2.28. DTD generada por Altova XMLSpy.....	55
Figura 3.1. Formato de las peticiones contenidas en los archivos de muestras.....	61
Figura 3.2. Distribución de popularidad.....	65
Figura 3.3. Distancia entre peticiones de documentos que se piden más de una vez (en peticiones).....	66
Figura 3.4. Distancia entre peticiones de documentos que se piden más de una vez (en MBytes)	66
Figura 3.5. Relación entre el tamaño del documento y el número de peticiones múltiples	69

Figura 3.6. Relación entre el tamaño del documento y la distancia (en datos) entre peticiones	72
Figura 4.1. HR y NUHR para distintos umbrales de acceso	76
Figura 4.2. HR y NUHR para $U=0.1$	76
Figura 4.3. BHR y NUBHR para distintos umbrales de acceso	77
Figura 4.4. ACHR para distintos umbrales de acceso	77
Figura 4.5. ACBHR para distintos umbrales de acceso	78
Figura 4.6. HR y NUHR para las políticas de acceso basadas en admitir los subtipos más solicitados.....	79
Figura 4.7. HR y NUHR para admisión del subtipo más solicitado de cada tipo de documento	80
Figura 4.8. BHR y NUBHR para las políticas de acceso basadas en admitir los subtipos más solicitados.....	81
Figura 4.9. ACHR para las políticas de acceso basadas en admitir los subtipos más solicitados.....	81
Figura 4.10. ACBHR para las políticas de acceso basadas en admitir los subtipos más solicitados	82
Figura 4.11. HR y NUHR para las políticas de acceso combinadas	83
Figura 4.12. BHR y NUBHR para las políticas de acceso combinadas	84
Figura 4.13. ACHR para las políticas de acceso combinadas	84
Figura 4.14. ACBHR para las políticas de acceso combinadas.....	85
Figura A.1. Diagrama UML de <i>EmulCacheProxy</i>	100
Figura A.2. Diagrama UML de <i>EmuladorNuevo</i>	101
Figura A.3. Diagrama UML de <i>MiLista</i>	102
Figura A.4. Diagrama UML de <i>ListaLFUDA</i>	103
Figura A.5. Diagrama UML de <i>ConfiguracionAplicacion</i>	104
Figura A.6. Diagrama UML de <i>XMLHandler</i>	104
Figura A.7. Diagrama UML de <i>SAX2Reader</i>	105

Índice de Tablas

Tabla 2.1. Políticas de reemplazo	20
Tabla 2.2. Tipos de documento	20
Tabla 3.1. Número de peticiones y volumen de datos por tipo de documento.....	61
Tabla 3.2. Peticiones y volumen de datos por subtipo para el tipo Imagen	62
Tabla 3.3. Estadísticos por tipo de documento	63
Tabla 3.4. Popularidad de los documentos	64
Tabla 4.1. Valores del parámetro T	74
Tabla 4.2. Peticiones rechazadas para las políticas de acceso de tipo umbral	75
Tabla 4.3. Subtipos más solicitados para cada tipo de documento	79
Tabla 4.4. Peticiones rechazadas para las políticas de acceso basadas en permitir los subtipos más solicitados	79
Tabla 4.5. Peticiones rechazadas para las políticas de acceso combinadas.....	83

Relación de Acrónimos

ACBHR	<i>Access Control Byte Hit Ratio</i>
ACHR	<i>Access Control Hit Ratio</i>
ASCII	<i>American Standard Code for Information Interchange</i>
BHR	<i>Byte Hit Ratio</i>
CPU	<i>Central Process Unit</i>
DOM	<i>Document Object Model</i>
DTD	<i>Document Type Definition</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>
HR	<i>Hit Ratio</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>HyperText Transfer Protocol</i>
IP	<i>Internet Protocol</i>
JAXP	<i>Java API for XML Processing</i>

JPEG	<i>Joint Photographic Experts Group</i>
JVM	<i>Java Virtual Machine</i>
LFU	<i>Least Frequently Used</i>
LFU-DA	<i>Least Frequently Used – Dynamic Age</i>
LRU	<i>Least Recently Used</i>
NLANR	<i>National Laboratory for Applied Network Research</i>
NUBHR	<i>Not Unique Byte Hit Ratio</i>
NUHR	<i>Not Unique Hit Ratio</i>
RAM	<i>Random Access Memory</i>
SAX	<i>Simple API for XML</i>
SGML	<i>Standard Generalized Markup Language</i>
TCP	<i>Transfer Control Protocol</i>
UML	<i>Unified Modelling Language</i>
URL	<i>Uniform Resource Locator</i>
W3C	<i>World Wide Web Consortium</i>
WWW	<i>World Wide Web</i>
XML	<i>eXtensible Markup Language</i>

CAPÍTULO 1: Introducción

1.1. *CACHÉ PROXY WEB*

El continuo crecimiento de la World Wide Web y la aparición de nuevos servicios finales tales como la descarga en tiempo real de contenidos multimedia, hace necesario el uso de tecnologías que reduzcan la latencia, el tráfico y la carga en los servidores Web.

El término *almacenamiento en caché* es empleado para denominar una técnica destinada a trasladar partes de un conjunto de datos a una ubicación más cercana a su lugar de procesamiento. En el caso de la consulta de páginas Web, los documentos se almacenan en memorias intermedias, *caché* Web, ubicadas cerca del usuario final, de forma que las solicitudes referentes a estos documentos son servidas por la *caché*, en lugar de ser atendidas por el servidor origen.

Para que el uso de una *caché* sea conveniente se deben cumplir dos condiciones fundamentales: en primer lugar que las peticiones de los usuarios presenten cierta localidad temporal, y por otro lado, que el coste de almacenar en *caché* el documento sea menor que el coste de traerlo directamente del servidor remoto. La primera condición se comprende fácilmente, no es interesante que un documento que no sea relativamente popular esté en *caché*, haciendo uso de un espacio en memoria que podría ser usado por documentos muy solicitados. Para verificar el cumplimiento de la segunda condición necesitamos comparar el coste de almacenar en *caché* un documento con el coste de no almacenarlo, para ello hay que tener en cuenta numerosos factores, algunos de ellos más fáciles de cuantificar que otros. Para calcular el coste de mantener en *caché* un documento tendremos que considerar, por ejemplo, costes de hardware, de software, y de administración del sistema. También habrá que considerar el tiempo que se ahorra

al descargar un documento de la *caché* en lugar de traerlo del servidor remoto, y el coste en términos de ancho de banda de Internet.

Podemos concluir que las ventajas de usar una *caché* Web son fundamentalmente las siguientes:

- Reducción de la latencia en la descarga de páginas Web.
- Reducción del ancho de banda ocupado.
- Reducción de la carga en los servidores Web (atenderán menos peticiones).

La latencia es el retardo que sufre la información al transmitirse de un punto a otro. La transmisión de información a través de circuitos eléctricos u ópticos está limitada por la velocidad de la luz. En la práctica, los pulsos eléctricos u ópticos viajan a aproximadamente dos tercios de la velocidad de la luz en cables y fibras. Los retardos que se producen en conexiones transoceánicas, por ejemplo, son del orden de unos 100 milisegundos. Pero la latencia no se debe únicamente al retardo que introduce el medio de transmisión, otra fuente importante de latencia son los retardos introducidos por diferentes elementos de la red (*routers, bridges...*), durante el procesamiento de la información. Además, se produce un retardo adicional en las colas de estos dispositivos de encaminamiento, sobretodo cuando los enlaces están próximos al límite de la saturación.

Cuando se usa una *caché* situada cerca de los usuarios, se reduce la latencia significativamente cuando se solicita un documento que se encuentra almacenado en *caché*. Los retardos de transmisión que se producen son mucho menores al encontrarse los sistemas origen y destino de la información próximos entre sí. Si el documento que se solicita no está en *caché*, traerlo del servidor remoto no supondrá un retardo mucho mayor que el que supondría una transferencia directa entre el usuario y el servidor remoto sin pasar por la *caché*.

Por otro lado, cada documento que puede ser obtenido de la *caché* supone un ahorro de ancho de banda en la red. Una *caché* Web reduce significativamente la

cantidad de ancho de banda consumida por el tráfico HTTP, dejando libre todo ese ancho de banda para otras aplicaciones de red.

La tercera ventaja que se obtiene cuando se usa una *caché* Web es la reducción en la carga de tráfico soportado por los servidores remotos de los documentos. El tiempo de respuesta de un servidor se incrementa cuando la tasa de llegada de peticiones aumenta. Un servidor muy ocupado es muy lento, independientemente de las condiciones de la red. Por el contrario, un servidor poco ocupado, al que le lleguen pocas peticiones, es más rápido respondiendo a éstas. Cuando se usa una *caché* Web, se disminuye el número de peticiones que se le hace al servidor remoto, y por tanto, las respuestas serán más rápidas.

Pero el uso de *caché* Web también tiene ciertos inconvenientes. La principal dificultad es garantizar la consistencia ya que la *caché* podría proporcionar un documento no actualizado como respuesta a la petición de un usuario. Para evitarlo se suele asignar un tiempo de expiración a los documentos, pasado el cual, el documento se considera obsoleto, cuando esto ocurre se valida su vigencia mediante una petición al servidor remoto.

Otro inconveniente es que la *caché* complica el análisis del tráfico en Internet. Las peticiones atendidas por la *caché* no son contabilizadas por el servidor remoto, el proveedor del documento, de manera que éste no puede llevar un control fiable de quién ha visitado su página y con qué frecuencia lo ha hecho.

El copyright es también una fuente de polémica con respecto al uso de *caché* Web. Algunos opinan que con estas técnicas se violan los derechos de autor de los documentos, al no permitir el control de la distribución de la obra. HTTP permite a los autores especificar cómo se han de manejar y distribuir sus documentos por los distintos tipos de *caché*, pero el protocolo no trata el copyright de una forma directa.

Por otro lado, hay quien predice que el porcentaje de contenido dinámico y personalizado en Internet crecerá en el futuro. Las respuestas dinámicas normalmente no deben ser almacenadas en *caché* ya que no pueden ser reutilizadas en una próxima petición. El contenido dinámico cambia de una petición a otra, se modifica en función

de la procedencia y características del usuario. Sólo tiene sentido almacenar en *caché* contenido estático. Si la predicción es cierta, el uso de *caché* será menos importante en el futuro. Pero hay estudios que apuntan en el sentido contrario, los contenidos estáticos crecerán más que los dinámicos y por tanto el uso de *caché* será cada vez más efectivo.

En función de su ubicación y de quién gestiona la *caché*, existen tres opciones básicas de desarrollo del sistema:

- a) La *caché* forma parte de la aplicación usada para la consulta de documentos (navegador):

El navegador puede disponer de capacidades para almacenar en el sistema de ficheros local los documentos a los que accede el usuario. Para el usuario del navegador, esta alternativa contribuye a mejorar su experiencia de usuario, sobre todo cuando se accede 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. No logra disminuir la latencia cuando los documentos se cargan por primera vez.

- b) *Caché* ubicada en un *proxy*: *Caché Proxy Web*:

Un *proxy* permite a otros equipos conectarse a una red de forma indirecta a través de él. Cuando un equipo de la red desea acceder a una información o recurso, es realmente el *proxy* quien realiza la comunicación con el servidor y a continuación traslada el resultado al equipo inicial. En la figura 1.1 podemos observar el esquema típico de una arquitectura de red basada en servidor *proxy* para el acceso al servicio Web.

El servidor *proxy* que dispone de una *caché* intercepta las solicitudes HTTP que realizan los clientes de la red local a la que pertenecen. A continuación busca en la memoria intermedia los documentos solicitados, y si los encuentra, los envía al usuario; si no es así los solicita al servidor origen en nombre del usuario, una vez

que los consigue, los transfiere al usuario, quedándose con una copia, si procede según la política de selección seguida.

Normalmente el *proxy* se sitúa en los límites de una red interna, pudiendo ser usado, además, como un cortafuegos de seguridad, prestando servicio a los usuarios de la misma. Otras funciones implementadas por el *proxy* son: traductor de formato de archivos, adaptador de protocolos, etc.

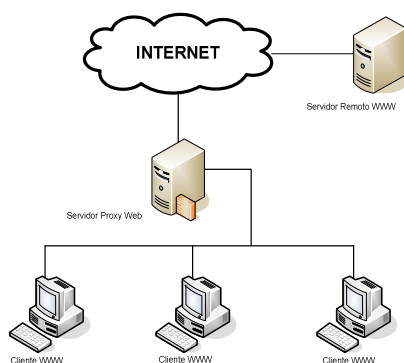


Figura 1.1. Esquema de acceso a Internet a través de servidor *proxy* para el servicio WWW

Esta configuración supone un ahorro apreciable de coste de ancho de banda en las comunicaciones hacia el exterior y además, mejora el tiempo de respuesta, ya que, al estar el *proxy* próximo al usuario, las respuestas sufren menor retardo de propagación en la red. Por otro lado, se aumenta la disponibilidad de los documentos requeridos, debido a que el usuario puede recibir algunas respuestas aunque el servidor origen se encuentre temporalmente inactivo. De este modo, desde el punto de vista del usuario, la red aparenta ser más fiable.

El problema de esta arquitectura basada en un único *proxy* radica en que supone un punto de fallo único en el servicio, si éste no se encuentra disponible, la Web completa aparece como no disponible para todos los usuarios de la red local. Además, existe otro problema relacionado con el potencial de escalado del sistema. A medida que aumenta el número de usuarios, el mismo *proxy* debe continuar dando servicio a todos, por lo que su eficacia se reducirá, llegando incluso a verse desbordado.

c) Conjunto de *cachés* en configuración transparente:

Esta alternativa trata de resolver las limitaciones comentadas anteriormente. En este tipo de configuración las peticiones HTTP son interceptadas y redistribuidas entre un grupo de *cachés* que se encuentran en una misma red local.

1.2. POLÍTICAS DE ACCESO Y DE REEMPLAZO

Cuando una *caché* recibe una petición, lo primero que hace es ver si tiene la respuesta a esa petición almacenada, es decir, si el documento está en *caché*. Si no es así, la petición es dirigida al servidor remoto, y decimos que se ha producido un fallo en *caché*. Los fallos se pueden producir porque el documento no se haya pedido con anterioridad, porque el documento sea de los que no está permitido almacenar, o bien porque aunque haya estado con anterioridad en la *caché*, haya tenido que ser borrado para dejar espacio a nuevos documentos. La decisión de si un documento debe ser almacenado en *caché* o no, corresponde a la política de acceso. La selección de los documentos que hay que eliminar de la *caché* para introducir otros nuevos es competencia de la política de reemplazo.

Si el documento pedido está en *caché*, podríamos estar ante un acierto. Para asegurar esto hay que comprobar que la respuesta almacenada es “fresca”. Se considera que una respuesta es “fresca” cuando el tiempo de expiración asociado al documento no ha sido alcanzado. Si se supera este tiempo, la respuesta se considera obsoleta. Una respuesta “fresca” produce de forma inmediata un acierto, sin embargo, en el caso de una respuesta obsoleta, hay que comprobar, mediante una petición al servidor remoto, si el documento que tenemos en *caché* está actualizado. Si es así, se consideraría acierto, en caso contrario, es un fallo y se trae desde el servidor la versión actualizada del documento, almacenándose una copia en *caché*.

1.2.1. POLÍTICAS DE REEMPLAZO

Los algoritmos de reemplazo se encargan de decidir qué documentos serán sustituidos en *caché* por otros nuevos cuando se ha ocupado todo el espacio disponible. Esto afectará a los aciertos que se producirán en un futuro. Serán eliminados tantos documentos como sean necesarios para la introducción del nuevo documento.

La política de reemplazo en una *caché proxy* se puede implementar de forma que favorezca a aquellos documentos que resulten más costosos, por ejemplo si proceden de enlaces congestionados o que requieran un número de saltos elevado para su obtención.

Para incorporar estas consideraciones, las políticas de reemplazo deben asociar un coste a cada documento y tratar de minimizar el coste total para un determinado flujo de acceso.

Para evaluar la eficiencia de las distintas políticas de reemplazo se suelen usar dos métricas:

- **HR (*Hit Ratio*)**: se define como el número total de peticiones que causan un acierto en *caché* dividido por el total de peticiones procesadas por la *caché*.
- **BHR (*Byte Hit Ratio*)**: se define como el volumen total de información, medido en Bytes, de los documentos que causan un acierto en *caché* dividido por el volumen total de información procesada, medido en Bytes.

El HR proporciona una medida de los documentos que han sido servidos por el proxy *caché* en lugar de por el servidor remoto y da una idea de la reducción de la latencia observada por los usuarios. El BHR, por su parte, proporciona una medida del tráfico servido por la *caché proxy* en lugar de por el servidor remoto y da una idea del ancho de banda ahorrado.

1.2.2. POLÍTICAS DE ACCESO

El propósito más importante de una *caché* es almacenar respuestas que puedan ser usadas en peticiones futuras. Para maximizar el rendimiento de la *caché* es necesario hacer una buena selección de los documentos que pueden acceder a un espacio de memoria que es finito. Por tanto, cuando un nuevo documento llega a la máquina donde está implementada la *caché* hay que decidir si es conveniente introducirlo o no en la memoria intermedia, es lo que denominamos política de acceso a *caché*.

Se han desarrollado multitud de políticas de reemplazo, pero sólo se han propuesto unos pocos algoritmos para el control de acceso. Abrams [Abrams'95] propuso un algoritmo basado en LRU (*Least Recently Used*) llamado *LRU-Threshold*. Mediante este algoritmo, no se admiten aquellos documentos que sobrepasen un tamaño umbral. Esta estrategia tiene el inconveniente de que depende del tipo del tráfico y del tamaño de la *caché*. Para solventar este problema Markatos [Markatos'96] propuso el algoritmo *LRU-Adaptive*, similar al anterior excepto por el hecho de que el umbral es calculado dinámicamente en función de la evolución de la *caché*. Aggarwal [Aggarwal'99], presentó una solución basada en una pequeña *caché* auxiliar LRU en la que se almacena metainformación de los documentos solicitados, por ejemplo, contadores del tiempo desde la última referencia, coste de acceso al servidor remoto, etc. Cuando un documento llega a la *caché*, el control de acceso comprueba si tiene datos de él en la *caché* de metainformación, si es así, se hace un cálculo que tiene en cuenta estos datos y se obtiene un valor, si este valor es superior a un determinado umbral se almacena en la *caché* principal. En otro caso, sólo se tiene acceso a la *caché* auxiliar, es decir, se actualizan los datos del documento, o se introducen por primera vez, si se trata de una primera petición.

En este Proyecto Fin de Carrera se realiza, entre otros, un estudio del control de acceso a *caché* considerando un factor calculado en función del tamaño del documento y del tipo de información que contiene (audio, imagen, video...). Para cada petición se realiza el cálculo de este factor, y a continuación, se comprueba si el resultado supera cierto umbral. Si es así, el documento será almacenado en *caché*, en otro caso será rechazado.

Cuando se usa una política de acceso, conviene usar nuevas métricas que contemplen la conveniencia o no de rechazar ciertos documentos. Analizaremos estas métricas en el siguiente capítulo.

1.3. OBJETIVO DEL PROYECTO Y CONTENIDOS

El objetivo de este Proyecto Fin de Carrera es la evaluación de distintas políticas de acceso a *caché* en servidores *proxy* Web, en función del tipo de documento. Para llevar a cabo el estudio se hace uso de un emulador de *caché* desarrollado en un proyecto anterior al que se le hicieron las modificaciones y optimizaciones oportunas.

Dado el elevado tiempo que requería la ejecución de las emulaciones, y ante la previsión del elevado número de ellas que habría que realizar, el primer objetivo fue optimizar el código de la aplicación.

Además, se tendrían que añadir algunas mejoras en el emulador encaminadas a ampliar los parámetros de emulación configurables, como la posibilidad de elegir múltiples tipos de documentos simultáneamente, seleccionando para cada uno de ellos una política de reemplazo y un porcentaje de memoria. Por otro lado, se añade una nueva política de reemplazo, LFU-DA, para posibilitar estudios comparativos más completos.

Con el objeto de automatizar las simulaciones a realizar, se consideró oportuno incluir la posibilidad de elegir la lectura de parámetros de entrada desde un fichero auxiliar, mediante el uso del estándar para el intercambio de información estructurada XML (*eXtensible Markup Language*).

Para llevar a cabo el estudio del acceso a *caché*, habría que realizar previamente una caracterización de las muestras de tráfico empleadas. Estas muestras fueron tomadas de la página de IRCache [IRCache]. El sistema IRCache consta de un conjunto de diez servidores *proxy* Web que emplean almacenamiento en *caché* y que se encuentran distribuidos a lo largo de los Estados Unidos. El proyecto IRCache

promueve el uso de las cachés Web, proporcionando archivos con muestras de tráfico para su uso por parte de investigadores y de otras organizaciones.

La memoria del Proyecto se estructura en cinco capítulos que se describen a continuación: el primer capítulo es esta introducción, en la que se presentan algunos conceptos preliminares, se explican los objetivos perseguidos y se describe la estructura de la memoria.

En el segundo capítulo se describe el emulador empleado, se detalla el proceso de optimización efectuado y las mejoras implementadas. Además, se presentan las nuevas métricas usadas cuando se controla el acceso a *cache* y se hace un estudio comparativo del rendimiento cuando se usa la nueva política de reemplazo implementada.

En el tercer capítulo se hace un análisis del tráfico. Se realiza un estudio del número de peticiones y del volumen de datos para cada tipo de documento, obteniendo de esta forma la importancia relativa de cada tipo dentro del conjunto de las trazas. También se analizan aspectos como la frecuencia, la distancia entre peticiones y la relación entre la frecuencia de las peticiones y el tamaño de los documentos.

En el capítulo cuarto se estudia el acceso a *cache* según la selección de documentos que se deciden que pueden ser almacenados. Se proponen tres tipos de políticas de acceso distintas, una basada en el establecimiento de un umbral, otra basada en la admisión de documentos cuyo subtipo pertenezca a los más solicitados, y una tercera que combina las dos anteriores.

En el capítulo cinco se comentan las conclusiones en función de los resultados obtenidos en el capítulo anterior y se proponen líneas futuras y posibles mejoras del sistema.

CAPÍTULO 2: Descripción del emulador empleado y mejoras implementadas

En el primer apartado de este capítulo presentamos el emulador empleado. Se trata de una aplicación desarrollada en Java, se hará una breve reseña de las características de este lenguaje de programación y se describirá la interfaz de usuario, centrando la atención en las nuevas funcionalidades añadidas a la versión anterior. En el segundo punto se explica el proceso llevado a cabo para la optimización del código, se describen varias estructuras de almacenamiento de datos en Java y se analiza cuál es la idónea para nuestra aplicación. En los siguientes apartados se explica con detalle las mejoras implementadas, entre ellas la política de reemplazo LFU-DA, las nuevas métricas usadas para tener en cuenta el control de acceso a *caché*, y el uso de XML para automatizar las emulaciones.

2.1. DESCRIPCIÓN DEL EMULADOR

2.1.1. LENGUAJE EMPLEADO EN SU DESARROLLO

El emulador de *caché proxy* Web usado está desarrollado en Java. Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 1990. Es un lenguaje que toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel como punteros.

Sun caracteriza Java diciendo que es simple, orientado a objetos, distribuido, interpretado, robusto, seguro, independiente de las arquitecturas de hardware, portable, potente, multitarea y dinámico. Veamos brevemente cada una de estas características:

- Simple: la sintaxis de Java es similar a la del lenguaje C o del C++, pero omite las características que los hacen confusos y poco seguros, por ejemplo la imposibilidad de que el programador gestione los punteros, la falta de herencia múltiple, etc. Además, Java dispone de un sistema llamado recogida de basura (*garbage collector*), que se ocupa de la destrucción automática de los objetos que ya no se utilizan, con el fin de liberar memoria. Por otro lado, permite la gestión de las excepciones.
- Orientado a objetos (OO): se refiere a un método de programación y de diseño del lenguaje. Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que se usen estén unidos a sus operaciones. Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Un objeto puede verse como un paquete que contiene el “comportamiento” (el código) y el “estado” (datos). El principio es separar aquello que cambia de las cosas que permanecen inalterables. Frecuentemente, cambiar una estructura de datos implica un cambio en el código que opera sobre los mismos, o viceversa. Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software. El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos. Por otro lado, la programación orientada a objetos propicia la creación de entidades más genéricas (objetos) que permiten la reutilización del software entre proyectos, una de las premisas fundamentales de la Ingeniería del Software. Java está provisto de un conjunto de clases (una clase define la forma y comportamiento de un objeto) que permiten manipular todo tipo de objetos (interfaz gráfica, acceso a la red, gestión de la entrada/salida...).
- Distribuido: Java permite la comunicación con servidores remotos a través de TCP/IP, UDP, HTTP o FTP, esto permite realizar desarrollos en arquitectura cliente/servidor. También es posible llamar desde una aplicación a objetos situados en otras máquinas de la red (principio de los objetos distribuidos).

- Interpretado: Los programas escritos en Java son interpretados por la máquina virtual o JVM (*Java Virtual Machine*), lo que hace que sea un poco más lento que otros lenguajes, sin embargo, esto tiene la ventaja de que no hay que recompilar un programa Java para cambiarlo de sistema, tan sólo hay que tener la máquina virtual apropiada instalada en la máquina donde se desee ejecutar el programa.
- Robusto: Gracias al tipo de datos usados y a la gestión de la memoria que realiza Java se evita sobrescribir datos en memoria de forma inoportuna.
- Seguro: El código es verificado en la compilación, y también por el intérprete en el momento de la ejecución, lo que permite evitar los bloqueos del sistema.
- Independiente de las arquitecturas de hardware: El compilador produce código que es independiente de la plataforma. Basta disponer de la JVM adecuada como se ha mencionado antes.
- Portable: Los tipos de datos primarios de Java tienen el mismo tamaño, sea cual sea la plataforma de desarrollo. Las bibliotecas de clases estándar de Java facilitan la escritura de código, que puede portarse fácilmente de una plataforma a otra sin adaptación alguna.
- Potente: Aunque es interpretado, Java incorpora un proceso de optimización de la interpretación del código (JIT, *Just In Time*) que permite obtener prestaciones similares a otros lenguajes no interpretados.
- Multitarea: Permite desarrollar aplicaciones que impliquen la ejecución simultánea de varias hilos.
- Dinámico: El programador no tiene que realizar la edición de los enlaces. Es, por tanto, posible realizar la modificación de una o varias clases sin tener que realizar una actualización de las modificaciones para el conjunto del programa.

La plataforma Java se compone de un entorno lógico que se puede ejecutar sobre numerosas plataformas físicas. Se compone de dos elementos: la máquina virtual Java (JVM), y la interfaz de programación de aplicación (API, *Application Programming Interface*). La JVM se ocupa de la carga del código, de la gestión de la memoria, de la seguridad y de la interfaz con el código nativo. La API Java contiene una colección de componentes lógicos que le proporcionan un cierto número de funcionalidades, como la gestión de la interfaz gráfica, o de los protocolos de intercambio de una red.

2.1.2. INTERFAZ DE USUARIO DEL EMULADOR

2.1.2.1. Introducción de parámetros de emulación

Cuando lanzamos la aplicación en el entorno de ejecución de Java, se muestra en pantalla la interfaz de usuario que se presenta en la figura 2.1. Ésta permite la introducción de los parámetros que definen el comportamiento de la *caché*, y el control de la ejecución de la emulación.

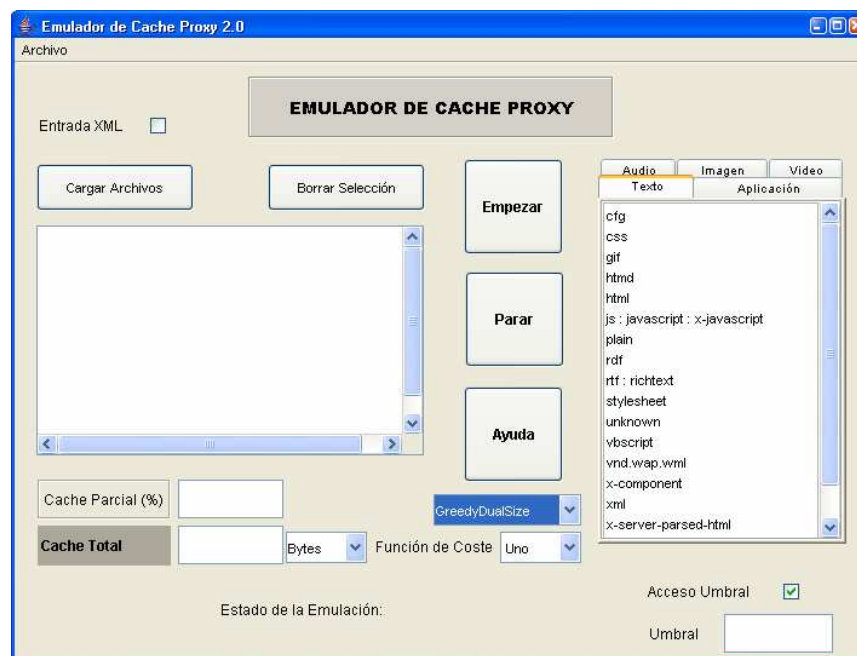


Figura 2.1. Interfaz de usuario de la aplicación

En la parte superior izquierda podemos observar una casilla que señalaremos si queremos que los parámetros sean obtenidos de un archivo externo. Esta opción está

pensada para efectuar consecutivamente varias emulaciones, obteniendo los parámetros de entrada usando el estándar XML.

Para seleccionar las muestras de tráfico a emplear usamos el botón “Cargar Archivos”, cuando se presiona aparece una nueva ventana de exploración que permite buscar la ubicación de los archivos correspondientes. Si nos equivocamos en la elección de los archivos podemos deshacer la selección mediante el botón “Borrar Selección”. En la figura 2.2 podemos observar la ventana de exploración referida en el caso de que estemos trabajando en un sistema Windows.

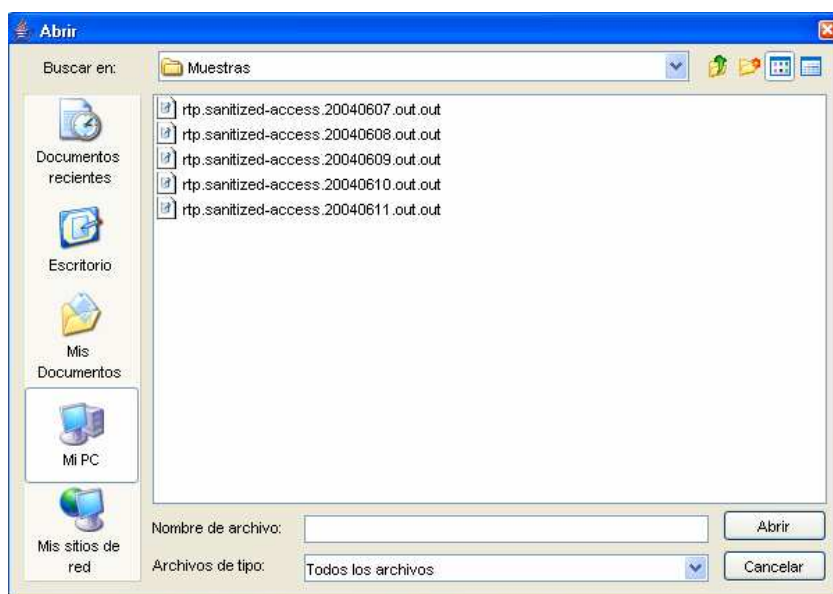


Figura 2.2. Ventana de carga de los archivos que contienen las muestras

Debajo del panel donde se presentan los archivos que contienen las muestras se encuentran dos casillas relacionadas con el tamaño de la *caché*. En “Cache Total” podemos introducir el tamaño total de la *caché*, junto a esta casilla hay un menú desplegable mediante el que podemos seleccionar la unidad de la cifra introducida, podemos elegir entre Bytes, KBytes, MBytes y GBytes. Sobre “Cache Total” tenemos la casilla “Cache Parcial %”, aquí podemos introducir qué porcentaje de la memoria total queremos destinar a cada tipo de documento.

A la derecha de esta zona se encuentra la selección de la política de reemplazo a aplicar, tenemos la posibilidad de seleccionar una política distinta para cada tipo de documento. Las políticas que podemos elegir son: LRU (*Least Recently Used*), LFU

(*Least Frequently Used*), GDS (*Greedy Dual Size*), GDSF (*Greedy Dual Size Frequency*), GD* (*Greedy Dual **) y LFU-DA (*Least Frequently Used with Dynamic Aging*).

Cuando se seleccionan las políticas de reemplazo de tipo *Greedy* aparece debajo un menú desplegable con la etiqueta “Función de Coste”, donde podemos seleccionar la función de coste asociada a la política de reemplazo. La función de coste se utiliza para determinar la forma en la que se calculará la prioridad asociada a cada documento a la hora de ser reemplazado. Las opciones son “Uno” y “Packets”. La primera opción define el coste constante, es decir, se supone que el coste de traer un documento desde un servidor remoto es siempre el mismo. La segunda opción asume que el coste de traer un documento es proporcional a su tamaño. Como para LRU, LFU y LFU-DA no se hace uso de ninguna función de coste ya que la prioridad viene determinada exclusivamente por el tiempo que transcurrió desde la última referencia al documento y por la frecuencia de las referencias, no aparece el menú desplegable para éstas. En el apartado 2.3 se describirá más detalladamente estas dos opciones.

A la derecha de la interfaz de la aplicación aparece un panel donde se podrá seleccionar los tipos y subtipos que se guardarán en *caché*. El panel presenta las pestañas: “Audio”, “Imagen”, “Vídeo”, “Texto” y “Aplicación”, correspondientes a cada uno de los tipos seleccionables. Cuando se selecciona una pestaña aparece la lista de subtipos disponibles, de entre los que podremos elegir los que podrán acceder a la *caché*. En la figura 2.3 se puede ver a modo de ejemplo la lista de subtipos para el tipo “Imagen”, tal como aparece en la aplicación. En la versión anterior del emulador sólo se podían seleccionar subtipos de uno de los tipos, con los cambios efectuados, es posible la selección de subtipos de distintos tipos simultáneamente.

Debajo de este último panel se sitúa una casilla con el título “Acceso Umbral” que se habrá de señalar para indicar al emulador que la política de acceso elegida es de este tipo. Cuando se señala aparece un cuadro de texto donde se introduce el umbral que se quiere usar. En el capítulo 4 se explica con detalle en qué consiste este tipo de política de acceso.

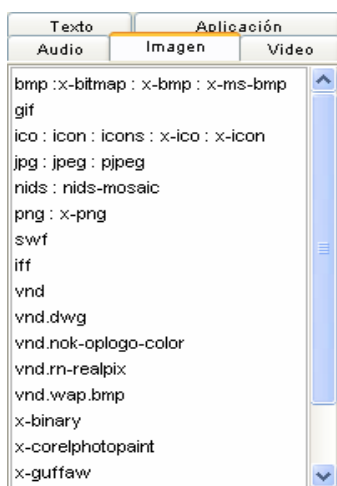


Figura 2.3. Subtipos seleccionables para ser procesados del tipo Imagen

2.1.2.2. Control de la emulación y botón de ayuda

En el centro de la ventana principal se sitúan dos botones que controlan la ejecución de la emulación y un tercero, de ayuda. El botón “Empezar” se usa para comenzar la emulación una vez que todos los parámetros han sido introducidos correctamente, bien manualmente o a través del archivo XML de entrada.

El botón “Parar” se emplea para detener la emulación, dejando en ese momento de procesar los ficheros de muestra y mostrando los resultados.

Debajo de estos dos botones se encuentra un tercer botón, “Ayuda”, al pulsarlo se muestra una ventana con una descripción del funcionamiento de la aplicación.

2.1.2.3. Resultados de la emulación

Cuando se introducen datos incorrectamente, por ejemplo cuando la suma de los porcentajes de *caché* para cada tipo de documento supera el 100 %, se muestra en pantalla un mensaje de error informando de tal circunstancia. Si los archivos con las muestras de tráfico, o el archivo con los parámetros de entrada, no poseen el formato

adecuado, también se presenta un mensaje de error en pantalla. En la figura 2.4 se observan varios ejemplos de mensajes de error.

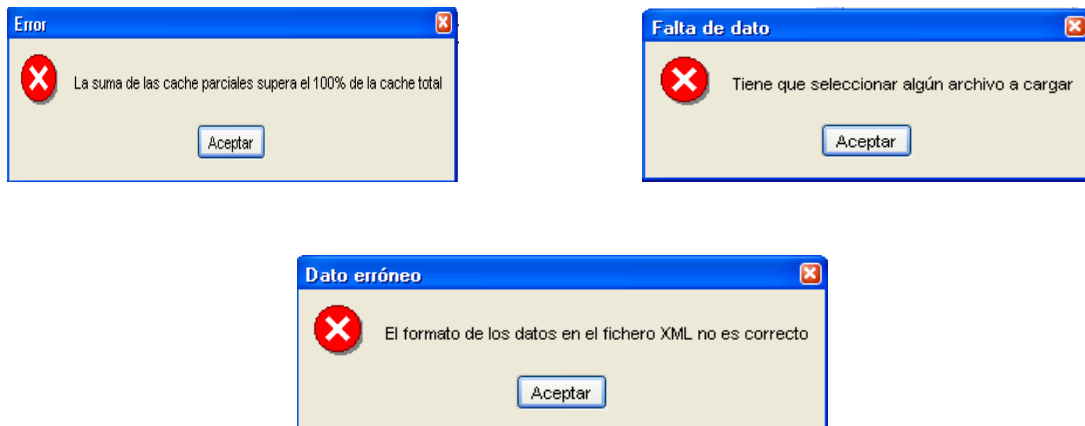


Figura 2.4. Mensajes de error

Mientras se realiza la emulación se muestra información sobre el progreso de ésta. Así, cuando se hace una emulación simple introduciendo los parámetros a través de los menús de la ventana principal, en la parte inferior aparece una línea con las peticiones de documentos realizadas hasta el momento. Cuando la emulación se hace capturando los parámetros de entrada de un fichero XML con varias emulaciones consecutivas, se va mostrando además la información sobre cuál es la emulación que se está ejecutando. En la figura 2.5 podemos observar la ventana que muestra el estado de la emulación en el caso de usar la entrada XML.

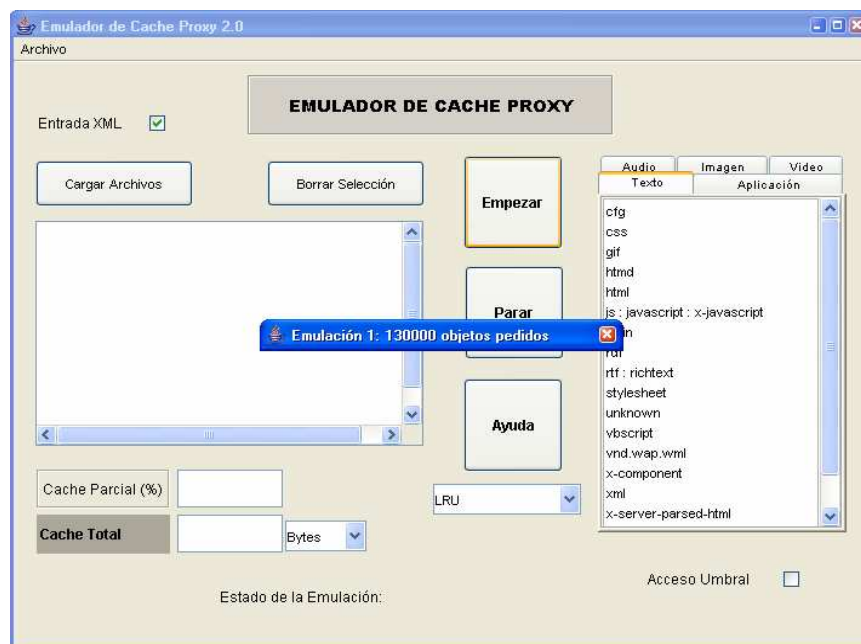


Figura 2.5. Estado de la emulación con entrada XML

Cuando se concluye correctamente la emulación/es se presenta al usuario una ventana en la que se muestra información sobre todas las métricas arrojadas por la aplicación. En la figura 2.6 podemos ver un ejemplo de la ventana de resultados tras ejecutar una emulación de prueba. En la parte superior se muestra el HR y el BHR por tipo de documento, es decir se tienen en cuenta los aciertos producidos por cada tipo, en lugar de los aciertos totales. Debajo se presentan el resto de métricas referidas al conjunto de los documentos. Además, se genera un archivo de resultados, cuyo aspecto, en el caso de entrada XML, se muestra en la figura 2.7. En este archivo se presentan los parámetros de entrada de cada emulación para poder identificarlas correctamente, y a continuación las métricas que produce el emulador al final de su ejecución. En la línea que hace referencia a las políticas de reemplazo usadas, se puede observar un vector de 6 elementos, en este vector la primera posición corresponde al tipo “Aplicación”, la segunda corresponde a “Audio”, la tercera es para “Imagen”, la cuarta posición del vector no se utiliza, el quinto elemento corresponde a “Texto”, y el último se reserva para “Video”.



Figura 2.6. Resultados de la emulación

Para representar las políticas de reemplazo usadas y los tipos y subtipos elegidos se utilizan una serie de códigos, el significado de estos códigos se muestran en las tablas 2.1 -políticas de reemplazo- y 2.2 -tipo de documento-. Las equivalencias para los subtipos, por su extensión, se especifican en el Apéndice A.

```

salidaXML1194208290968 - Bloc de notas
Archivo Edición Formato Ver Ayuda

EMULACIÓN 1
PARAMETROS DE ENTRADA:
El tamaño de la Cache es: 1000000 bytes
Las políticas son: 1 0 4 0 0 0
Lista de los tipos de los objetos almacenados, junto con sus subtipos:
0
  Numero de subtipos pasados: 11
  Lista de subtipos seleccionados: 0 1 2 3 4 5 6 7 8 9 10
4
  Numero de subtipos pasados: 10
  Lista de subtipos seleccionados: 0 1 2 3 4 5 6 7 8 9
=====
SALIDA:
Hit Ratio (%): 9.09
Byte Hit Ratio (%): 0.62
NUHR (%): 14.5
NUBHR (%): 1.92
ACHR (%): 41.7
ACBHR (%): 19.7
////////////////////////////////////
FIN EMULACIÓN 1
*****

EMULACIÓN 2
PARAMETROS DE ENTRADA:
El tamaño de la Cache es: 1000000 bytes
Las políticas son: 0 0 4 0 2 0
Lista de los tipos de los objetos almacenados, junto con sus subtipos:
0
  Numero de subtipos pasados: 11
  Lista de subtipos seleccionados: 0 1 2 3 4 5 6 7 8 9 10
2
  Numero de subtipos pasados: 10
  Lista de subtipos seleccionados: 0 1 2

```

Figura 2.7. Fichero de salida XML

LRU	0
LFU	1
GDS	2
GDSF	3
GD*	4
LFU-DA	5

Tabla 2.1. Políticas de reemplazo

Aplicación	0
Audio	1
Imagen	2
Texto	4
Video	5

Tabla 2.2. Tipos de documento

2.1.3. ESTRUCTURA INTERNA DEL EMULADOR

La interfaz de usuario y el emulador propiamente dicho se ejecutan en hilos de computación distintos, de esta forma podemos controlar la emulación desde la interfaz, parando, por ejemplo, la emulación, o comprobar el progreso de la emulación.

Para emular la *caché* se usa como estructura para almacenar los documentos (se guarda la fecha de la petición, el identificador, el tamaño, el tipo, y el subtipo del documento) una lista ordenada. En la versión anterior de la aplicación, se utilizaba para implementar esta lista en Java la clase *LinkedList* perteneciente al paquete *Java.util*. De cara a optimizar el código, se llegó a la conclusión de que la implementación más conveniente debe hacer uso de la clase *ArrayList* del mismo paquete, pensada para hacer

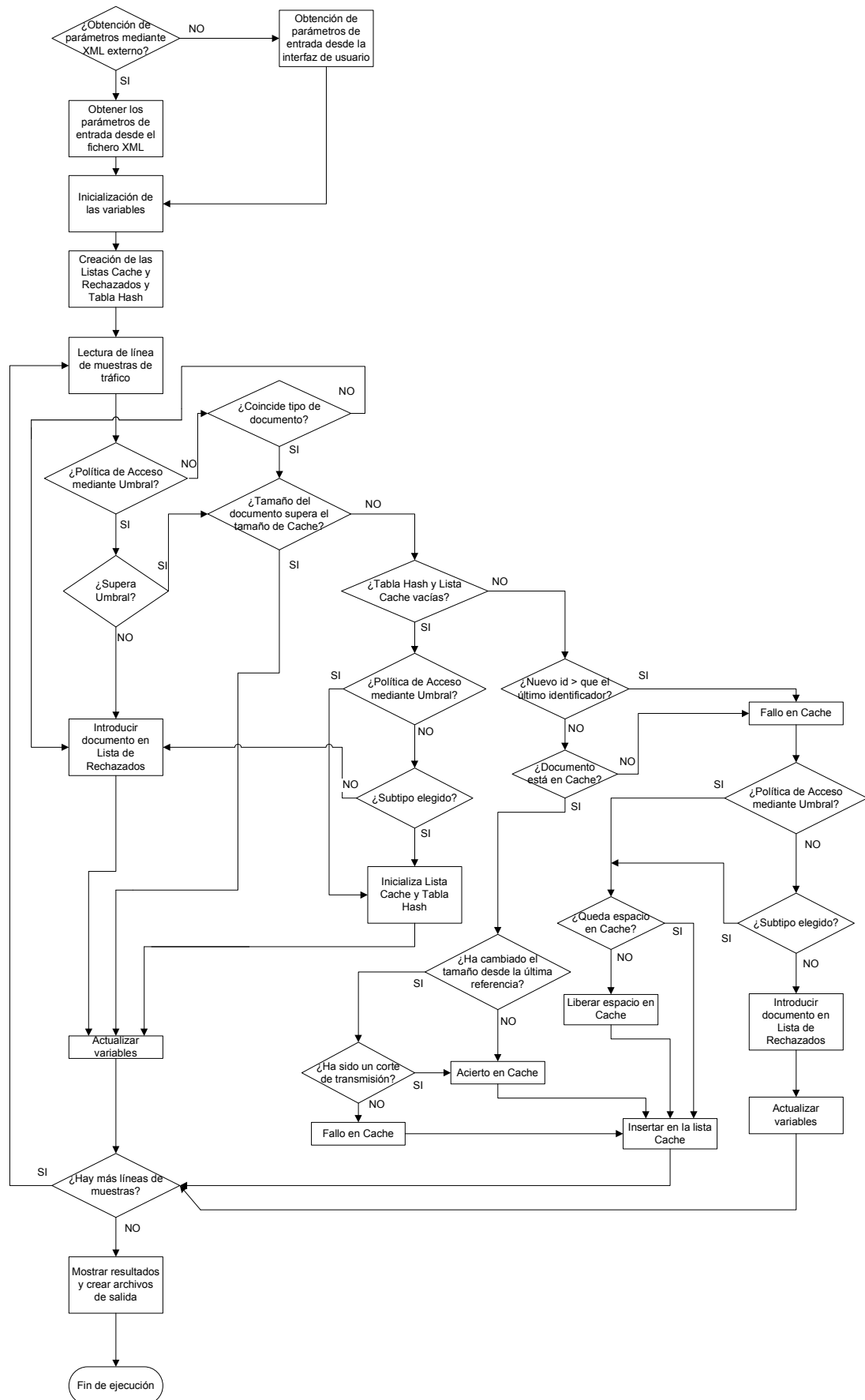
búsquedas rápidas de información. Estos aspectos relativos a la optimización del código serán tratados en el apartado siguiente.

En la lista los elementos se encuentran siempre ordenados de menor a mayor prioridad. Cuando hay que introducir nuevos documentos en *cache* y no hay suficiente espacio, se eliminan tantos documentos como sea necesario del principio de la lista, es decir, se extraen los de menor prioridad. La prioridad que se le da al documento, y por tanto la posición dentro de la lista, viene determinada por la política de reemplazo elegida. Para encontrar el punto de inserción en la lista se usa un algoritmo de búsqueda binaria.

Para determinar si un documento está en la lista o no, se emplea una tabla *Hash*. Una tabla *Hash* es una estructura contenedora asociativa que permite un almacenamiento y posterior recuperación eficientes de elementos, denominados valores, a partir de otros objetos, llamados claves. Las claves son procesadas mediante una función *Hash* que devuelve un código que determina la ubicación del elemento almacenado. La estructura de las tablas *Hash* es lo que les confiere su gran potencial, ya que hace de ellas unas estructuras extremadamente eficientes a la hora de recuperar información almacenada. El tiempo medio de recuperación de información es constante, es decir, no depende del tamaño de la tabla ni del número de elementos almacenados en la misma. En nuestro caso, la clave será el identificador del documento, y el valor, el propio documento.

En la figura 2.8 se muestra el diagrama de flujo que sigue la clase *EmuladorNuevo* que es la clase que se encarga de la ejecución de la emulación, implementando las políticas de reemplazo y de acceso seleccionadas, y realizando los cálculos que proporcionan las distintas métricas.

En el Apéndice B se pueden consultar los diagramas UML de las clases más representativas de la aplicación.

Figura 2.8. Diagrama de flujo de *EmuladorNuevo*

Cuando se recibe una petición de documento, si éste supera la política de acceso a *caché*, se consulta la tabla *Hash* para determinar si está en *caché*, si es así, hay que determinar si efectivamente se trata de un acierto, si el documento que se encuentra en *caché* es el que queremos realmente, para ello se debe comprobar que el tamaño del nuevo documento coincide con el que está almacenado. No obstante, también se considerará acierto si el tamaño del nuevo documento está por debajo del 95% del ya existente, suponemos en este caso que ha habido un corte de la transferencia. En los demás casos, es decir, si el tamaño es mayor, o menor pero con una diferencia menor del 5%, consideraremos que el documento ha sido modificado, y por tanto se produce un fallo en *caché* [Lindemann'02].

Si el documento es rechazado por la política de acceso se almacena en una lista información sobre el documento, identificador y tipo de documento, y sobre algunas variables del entorno en el momento del rechazo. Ésta información se utilizará para determinar si el documento ha sido rechazado correctamente, algo que es necesario para el cálculo de las nuevas métricas descritas en el apartado 2.4.

2.2. OPTIMIZACIÓN DEL EMULADOR

2.2.1. ESTRUCTURAS DE ALMACENAMIENTO EN JAVA

La estructura básica de almacenamiento de un conjunto de datos es el *array*. Un *array* es una estructura que nos permite almacenar datos de un mismo tipo. En Java, el tamaño de los *arrays* se declara en un primer momento y no puede cambiar en tiempo de ejecución como puede pasar en otros lenguajes.

Cuando se necesitan características más sofisticadas para almacenar objetos, que las que proporciona un simple *array*, Java pone a disposición del programador las clases colección. Entre otras características, las clases colección se redimensionan automáticamente, por lo que se puede colocar en ellas cualquier número de objetos, sin necesidad de tener que ir controlando continuamente en el programa la longitud de la colección. Es evidente que para implementar el emulador de *caché* este tipo de estructura de almacenamiento es la más adecuada, ya que a priori no conocemos el

número de documentos que contendrá la *caché*. Conocemos el tamaño de ésta, pero como los documentos tienen tamaños muy diversos, el total de documentos almacenados está indeterminado en principio. Queda ahora elegir la clase de tipo colección más adecuada de entre todas las que pone a nuestra disposición Java.

Una particularidad del uso de las colecciones en Java es que se pierde la información de tipo cuando se coloca un objeto en una colección. Esto ocurre porque lo que se pretende es disponer de una herramienta lo más general posible. De esta manera, las colecciones en Java se han diseñado de forma que manejen directamente objetos de tipo *Object*, que es el objeto raíz de todas las clases en Java. Por este motivo, hay que hacer siempre un *casting* al tipo adecuado antes de utilizar cualquier objeto contenido en una colección.

Java ofrece una amplia arquitectura de clases para manipular colecciones de datos. El marco básico se basa en una serie de interfaces que describen las capacidades soportadas por las distintas implementaciones. La jerarquía de dichas interfaces se muestra en la figura 2.9, y la de las implementaciones de dichas interfaces en la figura 2.10.

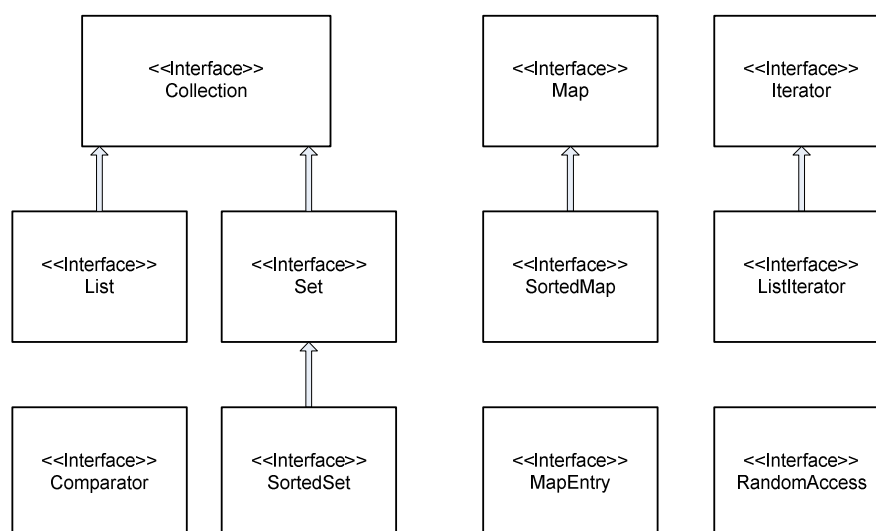


Figura 2.9. Jerarquía de las interfaces del marco de colecciones

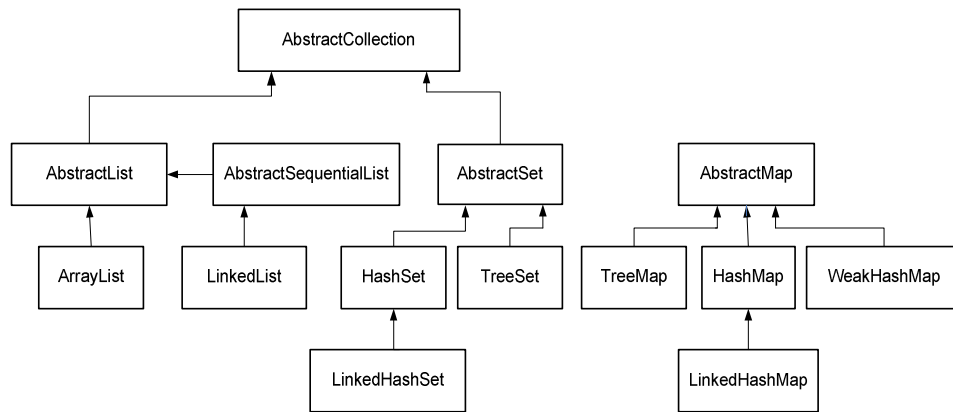


Figura 2.10. Jerarquía de las clases del marco de colecciones

De las interfaces mostradas en la figura 2.9, *Collection* y *Map* son las básicas. Una *Collection* es cualquier grupo de objetos, mientras que *Map* funciona con parejas de valores y claves. Dos interfaces extienden la interfaz *Collection*: *Set* y *List*.

Set es una colección sin duplicados. Para evitar duplicaciones cuando se añaden elementos a un *Set*, se utiliza el método *equals()* al elemento que se ha añadido para establecer la singularidad del objeto. La estructura de datos recomendada para un *Set* de propósito general es *HashSet*. Un *HashSet* es una colección *Set* que se basa en una tabla *Hash*. Un *LinkedHashSet* es similar a un *HashSet*, aunque al desplazarse por el *Set*, devuelve los elementos en el orden en que fueron añadidos. Otra estructura que puede ser usada es *TreeSet*, basada en una arquitectura de tipo árbol. Si el orden es importante, *TreeSet* es la elección adecuada.

La interfaz *List* es la colección básica para posicionamiento. Cuando un elemento se añade a una lista, se puede añadir en una posición específica o al final de ésta. Como se ha explicado en el apartado 2.1.3, donde se describe la estructura interna del emulador empleado, cuando un nuevo documento entra en *caché*, se decide en que posición se insertará en función de la política de reemplazo elegida. Por tanto, *List* es la colección adecuada para la implementación de la estructura de almacenamiento del emulador. A continuación se muestra la definición de la interfaz:

```

public interface List extends Collection {
    public void add(int index, Object element);
    public boolean add(Object obj);
    public boolean addAll(int index, Collection c);
}

```

```
public void clear();
public boolean contains(Object obj);
public boolean containsAll(Collection c);
public boolean equals(Object obj);
public Object get(int index);
public int hashCode();
public indexOf(Object obj);
public boolean isEmpty();
public Iterator iterator();
public int lastIndexOf(Object obj);
public ListIterator listIterator();
public ListIterator listIterator(int startPosition);
public Object remove(int);
public boolean remove(Object obj);
public Object set(int index, Object element);
public int size();
public subList(int fromIndex, int toIndex);
public Object[] toArray();
}
```

Hay dos implementaciones específicas de *List* dentro de *Collection*: *ArrayList* y *LinkedList*. *ArrayList* es una lista volcada en un *array*. Se suele utilizar como almacenamiento de objetos de propósito general, ya que permite un acceso aleatorio muy rápido a los elementos. El inconveniente de esta implementación es que realiza con bastante lentitud las operaciones de insertado y borrado de elementos en medio de la Lista, ya que cada vez que se realizan estas operaciones hay que indexar de nuevo todos los elementos a partir de la posición de borrado o insertado. Esto es especialmente crítico cuando las operaciones se hacen en las posiciones iniciales de la estructura, sobretodo cuando el *ArrayList* adquiere un tamaño importante. Se puede utilizar un *ListIterator* para moverse hacia atrás y hacia delante en la Lista, pero no para insertar y eliminar elementos. *ArrayList* tiene varios constructores, dependiendo de cómo necesitemos construir el *ArrayList*, podemos usar uno u otro. Como ejemplo se muestran a continuación dos de ellos:

- *ArrayList()* construye un *ArrayList* con capacidad cero por defecto, pero crecerá según vayamos añadiendo elementos:

```
ArrayList lista = new ArrayList();
```

- *ArrayList(int initialCapacity)* construye un *ArrayList* vacío con una capacidad inicial especificada:

```
ArrayList lista2 = new ArrayList(5);
```

LinkedList proporciona un óptimo acceso secuencial, permitiendo inserciones y borrado de elementos de cualquier parte de la lista de forma muy rápida. Sin embargo es bastante lento el acceso aleatorio, en comparación con *ArrayList*. Dispone además de los métodos *addLast()*, *getFirst()*, *getLast()*, *removeFirst()* y *removeLast()*, que no están definidos en ninguna interfaz o clase base y que permiten utilizar la lista enlazada como una pila, una cola o una cola doble.

LinkedList es la implementación que se eligió como estructura de almacenamiento de los documentos en la versión inicial del emulador. Efectivamente en la emulación de una *caché* tienen mucha importancia las inserciones y el borrado de elementos, operaciones que se hacen muy eficientemente con una estructura *LinkedList*. Pero el hecho de que el acceso a los datos sea secuencial es un importante inconveniente. En una estructura de este tipo, para acceder por ejemplo a la posición 500, tenemos que pasar por todas las anteriores. Al no permitir posicionarse de manera absoluta, *LinkedList* no resulta apropiado para búsquedas. Y las búsquedas ocupan mucho tiempo de procesador durante la ejecución de la aplicación, como se pudo comprobar mediante el uso de una herramienta software para la optimización de código Java, *JProfiler*. En el siguiente apartado se describe el uso que hemos hecho de esta aplicación.

Como se explicaba en el apartado dedicado a la descripción de la estructura interna del emulador, cada vez que hay que insertar un documento en la *caché*, hay que realizar una operación de búsqueda dicotómica para encontrar el punto de inserción adecuado en función de la política de reemplazo elegida. También se realiza una búsqueda cuando hay que actualizar la fecha y la frecuencia de un documento que está en *caché* tras un acierto. Como *ArrayList* proporciona un acceso directo a los elementos de la lista, la operación de búsqueda se realiza mucho más rápidamente. Usando esta estructura de almacenamiento en lugar de *LinkedList* conseguimos que el tiempo de ejecución se reduzca considerablemente en la mayor parte de las emulaciones, por tanto elegimos *ArrayList* como la implementación más adecuada para nuestra aplicación, sustituyendo la clase elegida en la versión anterior del emulador.

2.2.2. ANÁLISIS DEL RENDIMIENTO DE LA APLICACIÓN, USO DE *JProfiler*

Para evaluar el rendimiento del emulador y extraer conclusiones que permitan una optimización del código se ha elegido la herramienta de análisis *JProfiler* v.4.0.2 [JProfiler]. *JProfiler* es una utilidad para la optimización de aplicaciones Java. Es posible hacer un completo análisis del código, incluyendo monitorizaciones de la memoria, de la CPU, de los hilos de computación activos en cada momento, etc.

Usando las herramientas de monitorización de la CPU se pudo comprobar cuál es el coste computacional, en términos de ciclos de CPU consumidos en relación con el total, de cada uno de los métodos de los que consta el código del emulador en su versión anterior, dónde se hace uso de la estructura *LinkedList* descrita en el apartado anterior. Esto se hizo para diferentes tamaños de *caché* y seleccionando como documentos procesables por la *caché* los del tipo Imagen, que es el más importante tanto en términos de peticiones como de volumen total de información.

Definimos como *caché* infinita aquella que sería capaz de almacenar todas las muestras de que disponemos de un determinado tipo. Así la *caché* infinita para el tipo Imagen es de 14.7 GBytes. De aquí en adelante será habitual referirse al tamaño de *caché* como un porcentaje de la *caché* infinita.

Los resultados del análisis de *JProfiler* se muestran tras seleccionar la aplicación que queremos analizar (en nuestro caso, el emulador) y pasarle una serie de parámetros tales como la ubicación de la máquina virtual de Java que queremos usar y el directorio de trabajo. A continuación seleccionamos las características de análisis CPU mediante el botón correspondiente “CPU views” y pulsamos la opción de iniciar análisis.

JProfiler divide los resultados del análisis en cinco secciones, cada una de ellas puede mostrarse en pantalla seleccionándola desde la barra de menú situada en el lateral izquierdo. Las secciones son las siguientes:

1. *Memory views*: proporciona una monitorización en tiempo real del uso de la memoria y muestra información sobre la localización en memoria de los distintos objetos.
2. *Heap walker*: en esta sección se puede tomar una instantánea de la parte de memoria de asignación dinámica (memoria *heap*).
3. *CPU views*: permite determinar dónde se están consumiendo los recursos del procesador. Es el análisis más útil para reducir el tiempo de ejecución de la aplicación. Es la sección que se ha usado.
4. *Thread views*: desde esta sección se puede controlar el estado de los hilos de computación, lo que permite, por ejemplo, resolver problemas de bloqueos.
5. *VM telemetry views*: permite observar el estado de la máquina virtual de Java.

En la figura 2.11 se muestra los resultados arrojados por *JProfiler* tras la ejecución del emulador, eligiendo como parámetros de entrada los siguientes: Política de acceso: todos los documentos del tipo Imagen; Política de reemplazo: LRU; Tamaño de *caché*: 200 MBytes. Como fichero de muestras se ha elegido *rtp.sanitized-access.20040607.out.out*, que contiene aproximadamente la quinta parte del total de peticiones. Para otros tipos de documento y políticas de reemplazo los resultados son similares.

La ventana de información de la sección CPU muestra un árbol de llamadas a los distintos métodos, cada nodo es una llamada a un método. Para que la información mostrada sea suficientemente clara se puede hacer un filtrado de las clases a mostrar. En cada línea hay una barra porcentual cuyo tamaño es proporcional al tiempo que el procesador ha estado atendiendo al método. A continuación se muestra el porcentaje en forma numérica. Le sigue el tiempo mostrado en milisegundos y el número de invocaciones al método en cuestión. La última información en cada nodo es el nombre del método.

Se puede observar en la Figura 2.11 que la mayor del tiempo (el 72.7% del total) el procesador está ocupado atendiendo el método *Java.util.LinkedList.indexOf* que es llamado por el método *emulcacheproxy.ListaLRU.actualizObjOld* que se encarga de actualizar la fecha y la frecuencia de un documento de la *caché* cuando se ha producido un acierto. *indexOf* proporciona el índice asociado a un objeto que se le pasa como parámetro, para ello tiene que realizar internamente una búsqueda. Dado que, como hemos visto anteriormente, *LinkedList* no es eficiente para esta operación, el coste computacional es elevado.

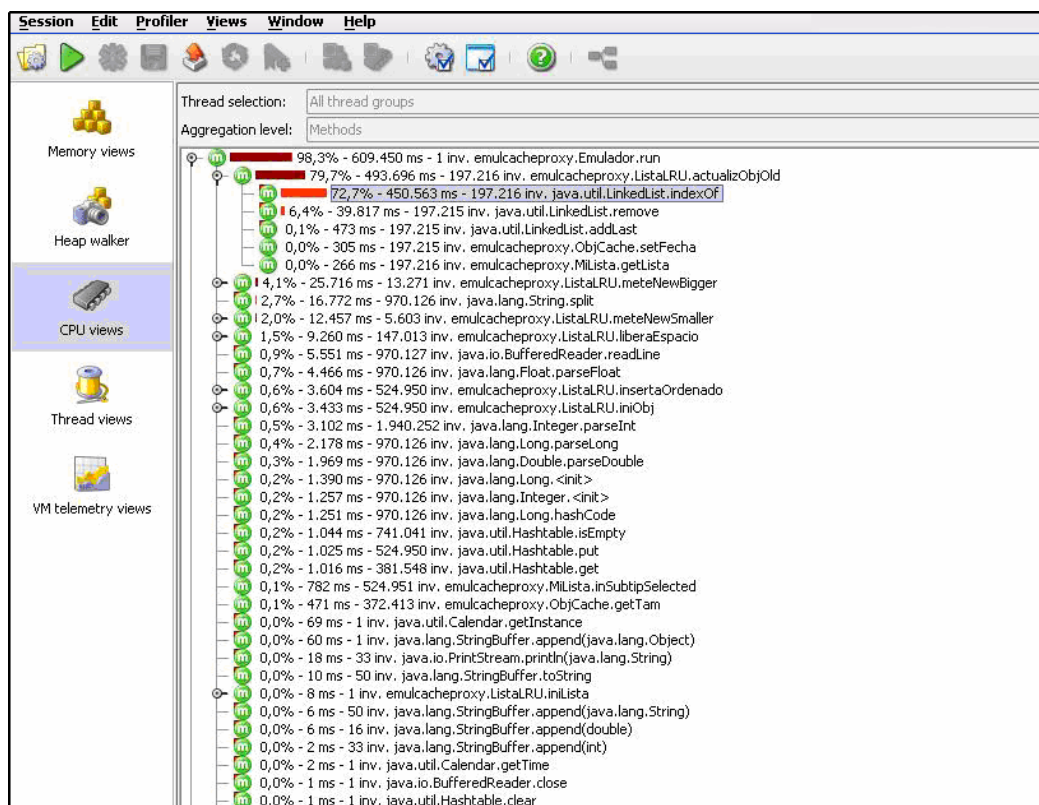


Figura 2.11. Consumo de CPU del emulador por métodos antes de la optimización (LRU)

A la vista de la importancia de la operación de búsqueda en la estructura de almacenamiento se procedió a cambiar *LinkedList* por *ArrayList*, haciendo las modificaciones oportunas en los métodos implicados. En la figura 2.12 podemos ver, para los mismos parámetros de entrada que los de la figura 2.11, como el consumo de CPU en las operaciones de búsqueda es menor.

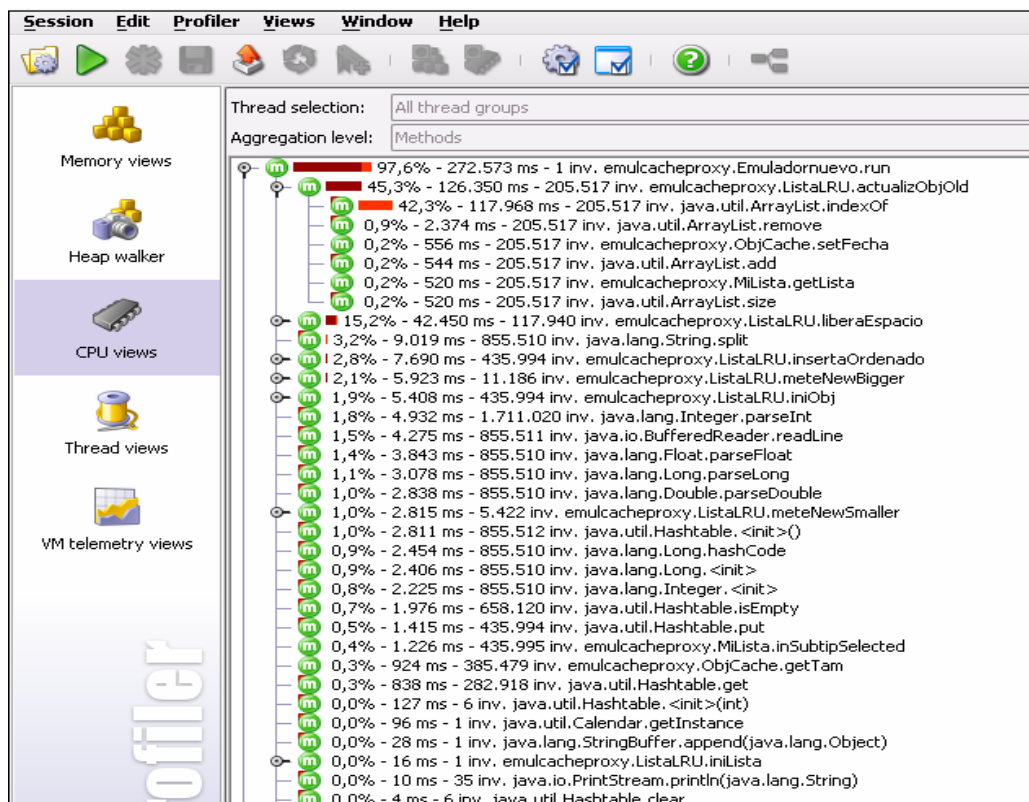


Figura 2.12. Consumo de CPU del emulador por métodos después de la optimización (LRU)

El porcentaje del tiempo de proceso que consume la búsqueda de objetos en la estructura pasa del 72.7% cuando se usaba *LinkedList* al 45.3% después de la optimización, con *ArrayList*. Las operaciones de búsqueda se hacen más rápidamente, disminuyendo así el tiempo necesario para realizar la emulación. Para los parámetros dados, la ejecución completa de la aplicación pasa de realizarse en 640 segundos a hacerlo en algo más de 290 segundos, es decir se ha conseguido que la aplicación sea aproximadamente 2.2 veces más rápida.

En la figura 2.13 se muestra como varía el tiempo de ejecución de la aplicación en función del tamaño de la *caché* para LRU, aceptando todas las peticiones y con todas las muestras de que disponemos. Se puede comprobar como además de ser menor el tiempo de ejecución en el caso del código optimizado, la pendiente de crecimiento de éste con el tamaño de *caché* es mucho menor que en la versión anterior del emulador.

En las pruebas anteriores se ha elegido como política de reemplazo LRU. Con esta política, cuando se introduce un nuevo documento en *caché*, o cuando hay un acierto y

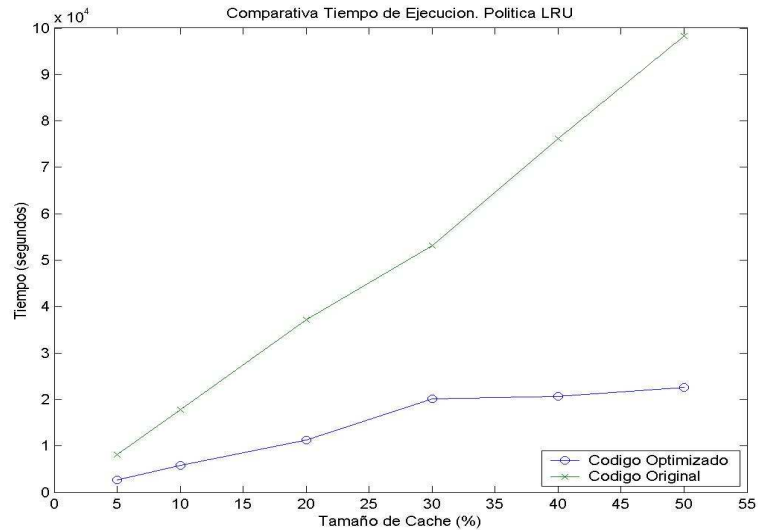


Figura 2.13. Evolución del tiempo de ejecución en función del tamaño de la *caché* para LRU

hay que realojar el documento después de actualizarlo, no hay que realizar una operación de búsqueda para encontrar el punto de inserción ya que los documentos en la lista están ordenados por antigüedad. Los elementos más recientes se colocan al final de la lista. Por tanto la operación de búsqueda se realiza sólo cuando hay que localizar un documento tras un acierto. En el resto de políticas hay que determinar el punto de inserción siempre que hay que introducir un documento en *caché*, es decir, la diferencia entre usar *LinkedList* o *ArrayList* será aún mayor, al realizarse más operaciones de búsqueda. En la figura 2.14 podemos ver un ejemplo para el caso de usar como política de reemplazo GD* antes de la optimización, en la figura 2.15 se muestra el mismo caso tras la optimización. Los parámetros de entrada son iguales a los de las pruebas anteriores, la función de coste, característica exclusiva de las políticas de reemplazo de tipo *Greedy*, es “Uno”. Vemos como en este caso cobra mucha importancia el método *insertaOrdenado*, llamado cada vez que hay que introducir un objeto en la estructura de almacenamiento.

Se comprueba que, como preveíamos, la mejora es aún más apreciable. Pasamos de un tiempo de ejecución total de 9858 segundos a 738 segundos. Para este caso se consigue que la aplicación sea 13.3 veces más rápida.

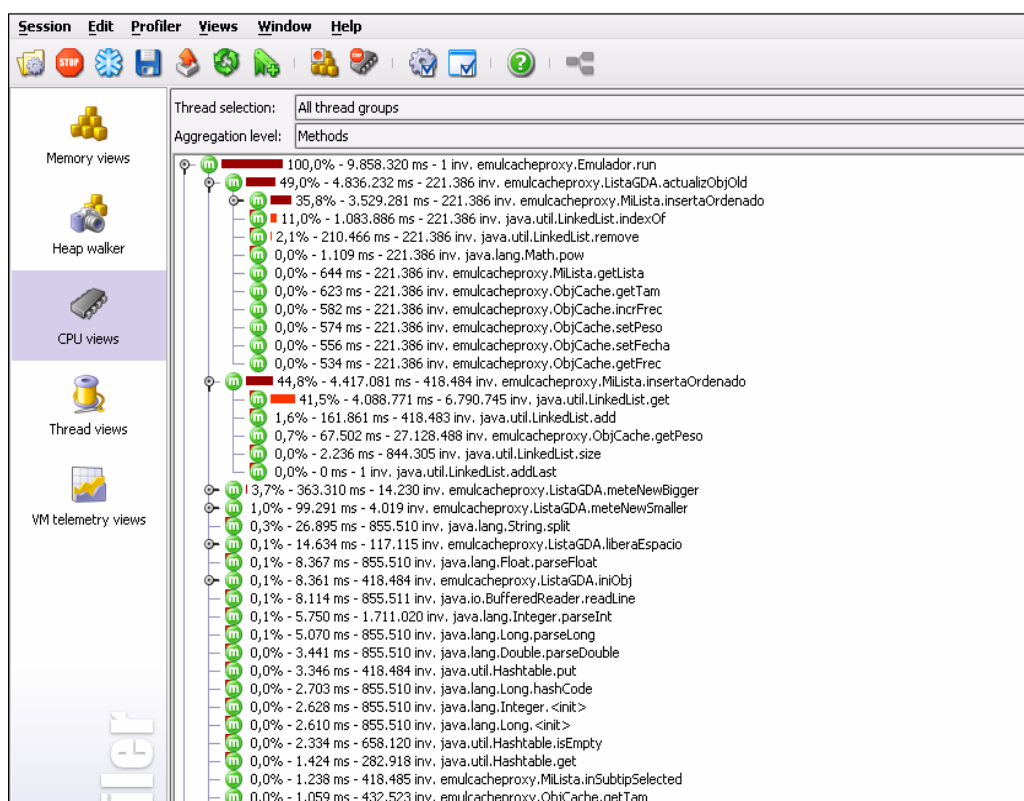


Figura 2.14. Consumo de CPU del emulador por métodos antes de la optimización (GD)

Se realizaron pruebas de rendimiento para todas las políticas de reemplazo (excepto para LFU-DA, que no era incluida por la versión anterior del emulador). En las gráficas de las figuras 2.16, 2.17, 2.18, 2.19, y 2.20 pueden observarse los resultados para distintos tamaños de *caché*. Para estas pruebas se admitieron todas las peticiones del tipo imagen, rechazando las demás. Los tiempos de ejecución de la aplicación se muestran en segundos.

Para todas las políticas de reemplazo se puede apreciar que la primera versión del emulador (código original), que emplea *LinkedList*, sólo es comparable en velocidad para el caso de tamaño de *caché* relativamente pequeño, es decir cuando la lista que hay que manejar no es demasiado grande. En estos casos el hecho de que la búsqueda no sea eficiente no es tan perjudicial, y tiene más peso la particularidad de que las operaciones de inserción y borrado sean más rápidas en una estructura *LinkedList* que en un *ArrayList*. Para tamaños de *caché* superiores a 1 MByte, el código modificado es claramente más eficiente. Teniendo en cuenta que lo habitual en *caché* Web es tener memorias superiores a este límite, queda justificado el uso de *ArrayList* para optimizar el código. Se puede apreciar también en las figuras anteriores que la mejora que se

obtiene para las políticas de reemplazo de tipo *Greedy* es aún más evidente que en el resto de políticas. Además la pendiente de crecimiento, como se había apuntado anteriormente, es mucho menor con el código optimizado.

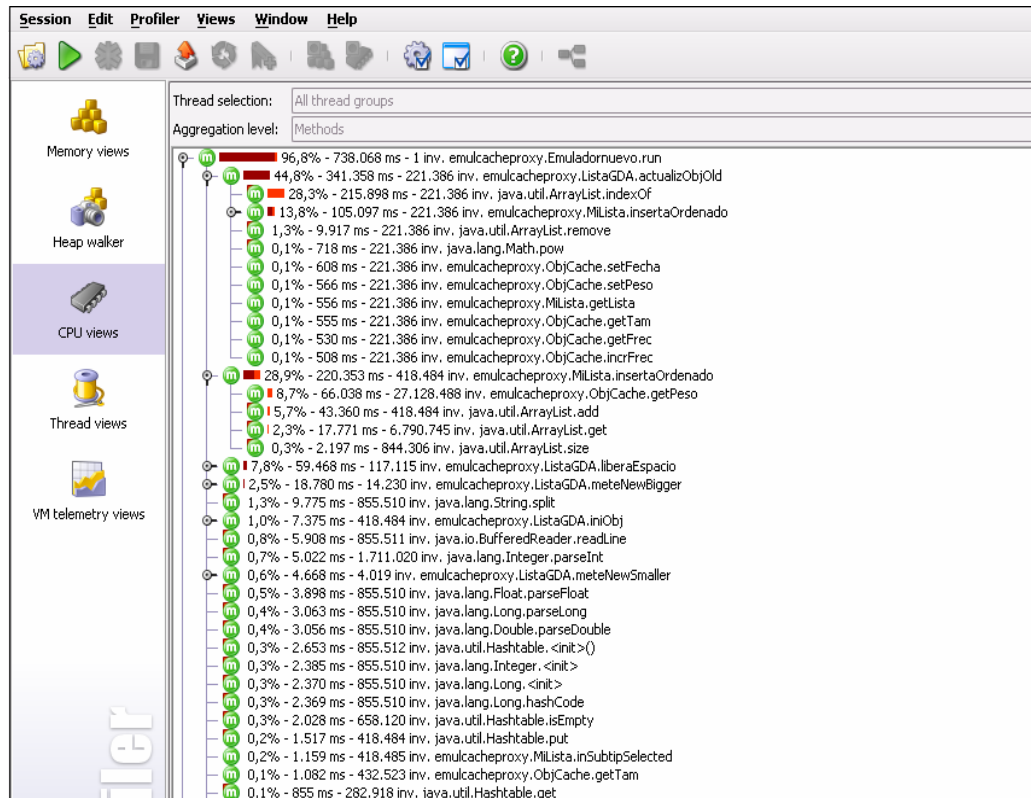


Figura 2.15. Consumo de CPU del emulador por métodos después de la optimización (GD*)

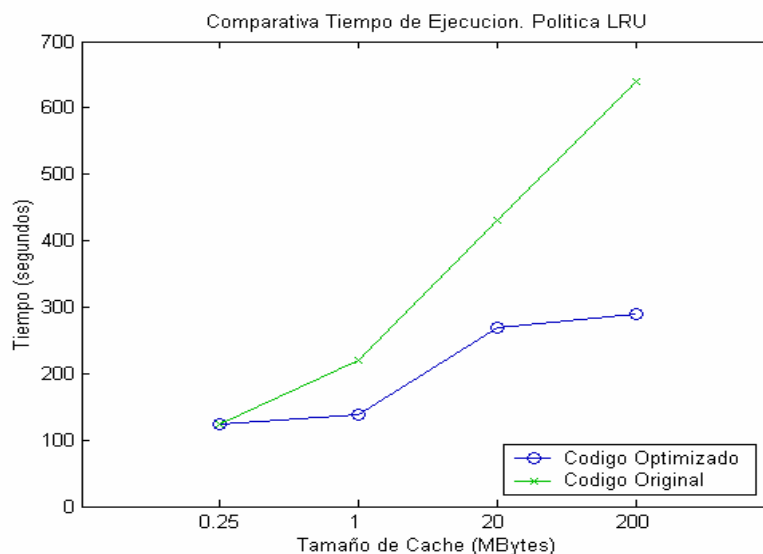


Figura 2.16. Tiempo de ejecución para distintos tamaños de *caché* para LRU

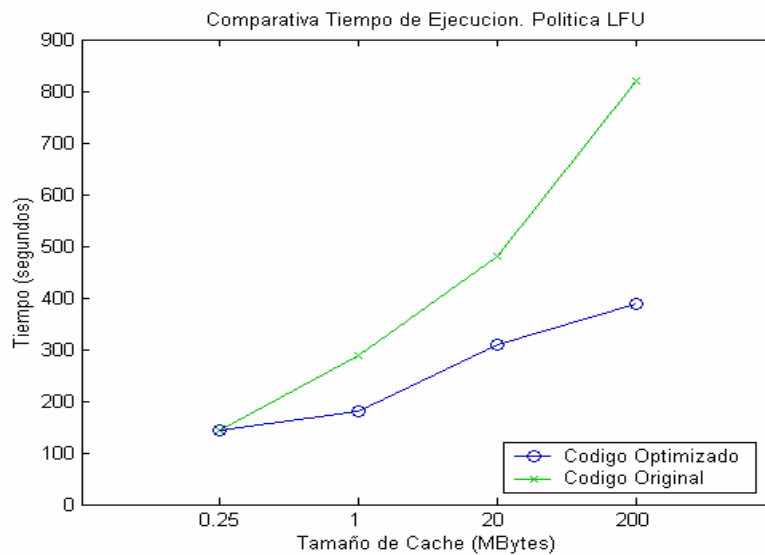


Figura 2.17. Tiempo de ejecución para distintos tamaños de *caché* para LFU

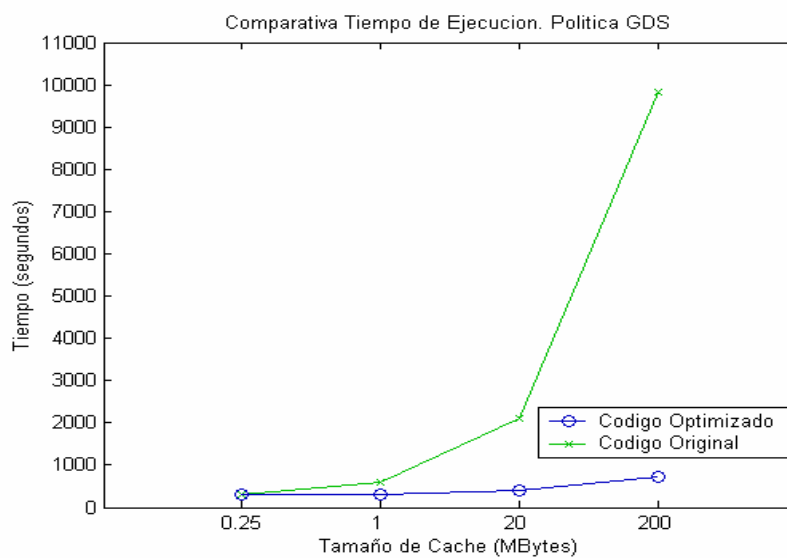


Figura 2.18. Tiempo de ejecución para distintos tamaños de *caché* para GDS

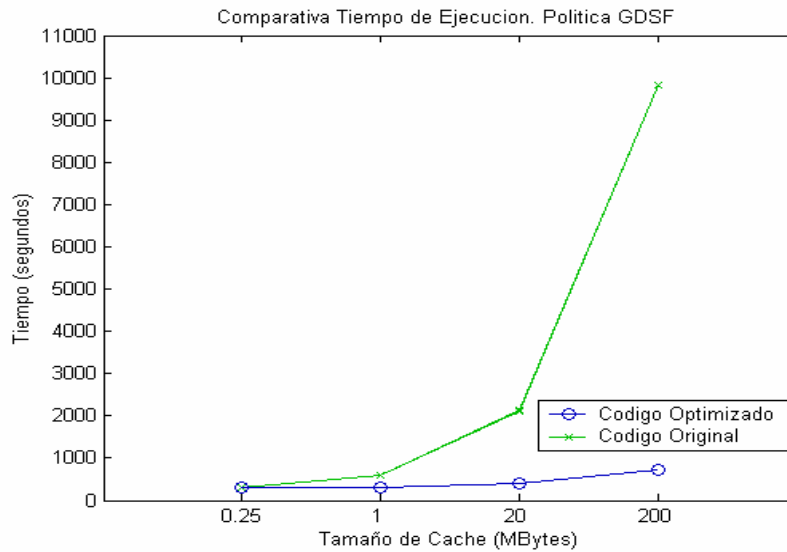


Figura 2.19. Tiempo de ejecución para distintos tamaños de *caché* para GDSF

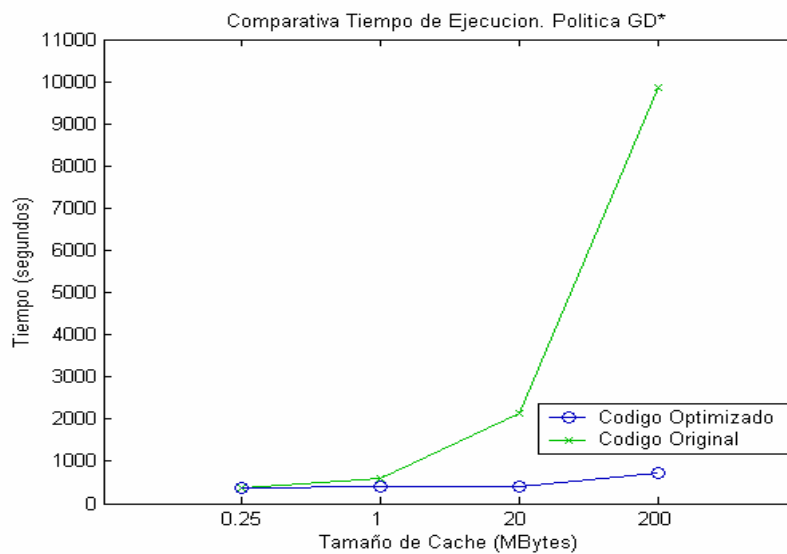


Figura 2.20. Tiempo de ejecución para distintos tamaños de *caché* para GD*

2.3. NUEVA POLÍTICA DE REEMPLAZO: LFU-DA

A continuación se hace una breve descripción de las políticas de reemplazo implementadas en el emulador de *caché proxy* Web empleado en este Proyecto Fin de

Carrera, y se presenta un nuevo algoritmo de reemplazo que ha sido añadido a la aplicación.

- *Least Recently Used* (LRU)

Esta política de reemplazo busca un mejor rendimiento de la *caché* utilizando como parámetro de reemplazo el tiempo que ha permanecido un documento sin haber sido solicitado de nuevo por algún usuario. Es decir el documento que hace más tiempo que fue referenciado es el candidato a ser reemplazado en la *caché*.

- *Least Frequently Used* (LFU)

El criterio para reemplazar un documento en esta política es la frecuencia de acceso al mismo, de entre los que tiene una frecuencia de acceso menor se reemplazará aquel que hace más tiempo que fue referenciado.

Esta estrategia trata de mantener en *caché* los documentos más populares y reemplaza aquellos que apenas se solicitan. Tiene el inconveniente de que puede haber documentos que se piden repetidamente durante un tiempo, incrementando el valor de su frecuencia, y que después no son solicitados más. Estos permanecerán, por tanto, en *caché* sin ser necesario, provocando lo que se ha dado en llamar “contaminación de la *caché*”.

- *Least Frequently Used with Dynamic Aging* (LFU-DA) [Arlitt'99]

Es la nueva política añadida al emulador usado en este Proyecto Fin de Carrera.

Para evitar la contaminación de la *caché*, a la que nos hemos referido anteriormente como un problema de LFU, se utiliza un mecanismo de *aging* (envejecimiento) similar al que utilizan los algoritmos de tipo *Greedy* y que se detalla en el siguiente punto. Mediante este mecanismo agregamos un factor de edad, para evitar que ciertos documentos permanezcan en *caché* por tiempo indefinido.

LFU-DA se muestra muy eficiente en entornos donde la frecuencia es una característica importante pero no es posible usar LFU debido a los problemas de contaminación que presenta.

A continuación, en las figura 2.21 y 2.22, se muestra una comparativa entre las tres políticas anteriores, para las métricas HR y BHR, para distintos tamaños de *caché*. En las emulaciones se ha permitido que todos los documentos del tipo Imagen accedan a *caché*.

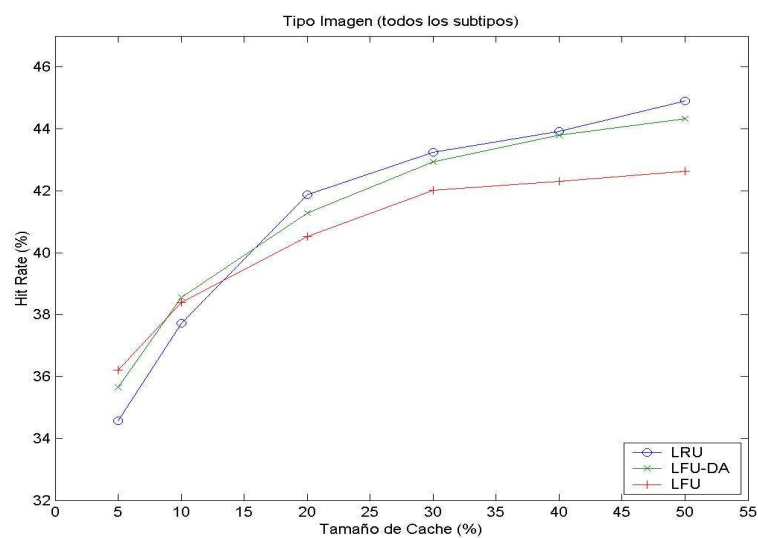


Figura 2.21. HR para los algoritmos LRU, LFU y LFU-DA en función del tamaño de *caché*

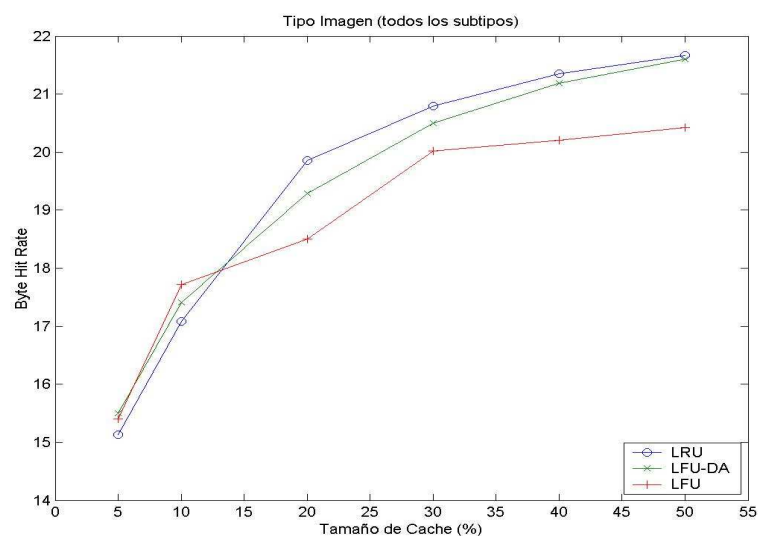


Figura 2.22. BHR para los algoritmos LRU, LFU y LFU-DA en función del tamaño de *caché*

Se comprueba que para tamaños de *caché* inferiores al 15% de la *caché* infinita LFU muestra un mejor comportamiento tanto en el caso del HR como del BHR. A partir de ese punto, la política de reemplazo que mostraba unos peores resultados, LRU, pasa a ser la de mejor comportamiento. La nueva política implementada, LFU-DA se mantiene siempre entre las dos anteriores.

- *Greedy Dual Size* (GDS) [Cao'97]

Los algoritmos *Greedy Dual* [Young'94] están basados en la consideración del hecho de que los documentos presentan un coste asociado a traerlos desde el servidor donde están alojados. Estos algoritmos suponen que todos los documentos tienen el mismo tamaño.

Estos algoritmos asocian un valor H a cada documento almacenado en la *caché*. Inicialmente, cuando un nuevo documento es introducido en la *caché*, H toma el valor del coste de traer el documento desde el servidor remoto. Cuando es necesario reemplazar alguno de los documentos almacenados en la *caché*, el documento que tenga un H menor será eliminado, y el resto de documentos reducen el valor de su H en la cantidad del que tenía el documento eliminado (mecanismo de envejecimiento). Cuando se produce un acierto, el valor H del documento referenciado se reestablece al coste original de traerlo a la *caché*. De esta forma el valor H de los documentos recientemente referenciados mantiene buena parte del coste original, al contrario que aquellos que no han sido pedidos desde hace tiempo.

Greedy Dual Size considera además el tamaño de los documentos, priorizando el reemplazo de los documentos de tamaño mayor. De esta forma, para asignar la prioridad a un documento a introducir en *caché*, H se calcula como se muestra en la ecuación 2.1.

$$H(d) = \text{coste}(d) / \text{tamaño}(d) \quad (2.1)$$

Donde d es el documento en cuestión, *coste* es el coste de traer el documento desde su servidor remoto y *tamaño* es el tamaño del documento medido en Bytes.

El coste se definirá en función del objetivo que persiga el algoritmo. Así, para maximizar el Hit Ratio, se debe fijar el coste como constante a 1, función de coste “Uno” en la interfaz de usuario. De esta forma los documentos más grandes tendrán una H menor independientemente de lo que cueste traerlos de su servidor remoto, serán reemplazados con mayor probabilidad. El coste a 1 favorece a los documentos pequeños, reemplazando los documentos más grandes, sobre todo aquellos poco referenciados.

La estrategia “Packets” establece el coste para cada documento como $2 + \text{tamaño}/1460$. Se considera que este es el número de paquetes que se transmiten tras un fallo en *caché*, es decir, un paquete para la petición, otro paquete para la respuesta y tamaño/1460 paquetes para los datos pedidos, asumiendo 1460 Bytes como el tamaño de un segmento TCP. Definiendo el coste de esta forma se pretende maximizar el *Byte Hit Ratio*. De esta forma se asigna un coste mayor a los documentos de mayor tamaño que a los más pequeños, permitiendo así que éstos sean reemplazados con mayor probabilidad que aquellos. No obstante si un documento de gran tamaño no es referenciado de nuevo en mucho tiempo será reemplazado debido al mecanismo de envejecimiento.

De otra forma, se puede decir que con GDS(Unos) se busca minimizar los fallos en *caché*, permitiendo un gran número de peticiones satisfechas con poca latencia. Por otro lado con GDS(Packets) se intenta minimizar el tráfico de red que causan los fallos en *caché* provocados por las referencias a documentos de gran tamaño sobre todo.

El mecanismo de envejecimiento para los algoritmos de tipo *Greedy Dual* comentado anteriormente supone un coste computacional demasiado elevado al requerir una operación de resta por cada documento que esté almacenado en *caché* cuando se realiza un reemplazo. Para evitarlo se puede añadir un desplazamiento L a la fórmula que proporciona el valor H. Un algoritmo eficiente para esta implementación es el siguiente:

Inicializar L:=0;

Procesar cada petición:

Petición del documento d:

- 1) *si d está en caché*
 - 2) $H(d) := L + \text{coste}(d) / \text{tamaño}(d);$
 - 3) *si d no está en caché*
 - 4) *mientras no haya suficiente espacio en memoria para d*
 - 5) $L := \min_{q \in C} H(q)$
 - 6) *eliminar q tal que $H(q) = L$;*
 - 7) *introducir d en memoria y establecer $H(d) := L + \text{coste}(d) / \text{tamaño}(d);$*
- fin*

- *Greedy Dual Size Frequency (GDSF) [Cherkasova'98]*

El anterior algoritmo puede ser mejorado de forma que se tenga en cuenta además la frecuencia de acceso al documento. Así, esta política de reemplazo tiene en cuenta las características más importantes del documento, es decir: el tamaño, cómo de reciente ha sido su último acceso y la frecuencia de solicitud.

La principal diferencia respecto al anterior algoritmo radica a la hora de hacer el cálculo de H, ahora se tendrá en cuenta la frecuencia con la que ha sido referenciado el documento. La fórmula que establece H se muestra en la ecuación 2.2. El parámetro frecuencia se incrementa en uno por cada petición que provoque un acierto, el valor inicial es 1.

$$H(d) = L + \text{frecuencia}(d) \times (\text{coste}(d) / \text{tamaño}(d)) \quad (2.2)$$

- *Greedy Dual* (GD*) [Lindemann'02]*

En esta otra política *Greedy Dual* se trata de tener en cuenta la popularidad de los documentos a largo plazo y la correlación temporal entre referencias próximas en el tiempo.

La popularidad y la correlación temporal entre peticiones son propiedades que están relacionadas, hay una relación causal entre ambas. Los documentos más populares tienden a ser referenciados más frecuentemente, mostrando, de este modo una mayor localidad temporal que aquellos que no son tan populares. Podemos decir

que la popularidad y la correlación temporal son dos dimensiones que presenta la localidad temporal.

Para extraer los efectos de la popularidad de los documentos a partir de la correlación temporal se considera la distribución de probabilidad de referencias de documentos con un nivel de popularidad similar y se cuantifica el grado de correlación temporal mediante el parámetro β , que es la pendiente de dicha distribución en escala logarítmica. La probabilidad de que el tiempo entre referencias sea igual a t es proporcional a $t^{-\beta}$ a corto plazo. El valor de β depende del tipo de tráfico y suele estar comprendido entre 0 y 1, siendo habitual que este dentro del intervalo 0.3-0.7.

La prioridad asignada al documento cuando entra en *caché*, tiene que reflejar ahora, además de la frecuencia del documento, el grado de correlación temporal. Como las referencias entre peticiones de documentos de similar popularidad sigue una ley $t^{-\beta}$, el tiempo máximo que debe permanecer un documento en *caché* debe ser proporcional a $u(p)^{1/\beta}$, siendo $u(p) = \text{frecuencia}(d) \times (\text{coste}(d) / \text{tamaño}(d))$. Por ello se redefine el parámetro H como se muestra en la ecuación 2.3.

$$H(d) = L + (\text{frecuencia}(d) \times (\text{coste}(d) / \text{tamaño}(d)))^{1/\beta} \quad (2.3)$$

La frecuencia proporciona la popularidad a largo plazo mientras que la correlación temporal está relacionada con la tasa de envejecimiento controlada por el parámetro β . Valores pequeños de este parámetro suponen una débil correlación de las referencias y por tanto los documentos sufren un envejecimiento más lento, valores altos provocan el efecto inverso.

2.4. NUEVAS MÉTRICAS IMPLEMENTADAS

Las métricas HR (*Hit Ratio*) y BHR (*Byte Hit Ratio*), descritas en el capítulo 1 de esta memoria, no son adecuadas cuando usamos una política de acceso a *caché* ya que estas métricas contabilizan todas las peticiones, incluso aquellas que hacen

referencia a documentos que no son admitidos por la *caché* y no son referenciados de nuevo. Para tener en cuenta el efecto que causa en el rendimiento de la *caché* el uso de un control de acceso se utilizan las nuevas métricas que se definen a continuación.

2.4.1. NOT UNIQUE HIT RATIO (NUHR) Y NOT UNIQUE BYTE HIT RATIO (NUBHR)

NUHR es una adaptación de HR para el caso de que exista una política de acceso a *caché* que no se permita el almacenamiento de todos los documentos pedidos. Se define de la siguiente forma (ecuación 2.4):

$$NUHR = \frac{\#Aciertos}{\#Total_Peticones - \#RechazadosOk} \quad (2.4)$$

Donde *#Aciertos* es el número total de peticiones que causan un acierto en *caché*. *#Total_Peticones* es el número total de peticiones procesadas por la *caché* y *#RechazadosOk* es el número de peticiones de documentos que fueron correctamente rechazados.

Por otro lado NUBHR es una modificación de BHR para el caso de que exista una política de acceso. La definición es la siguiente:

$$NUBHR = \frac{Volumen_Aciertos}{Volumen_Total_Peticones - Volumen_RechazadosOk} \quad (2.5)$$

Los términos de la ecuación 2.5 se refieren a volumen de información (en Bytes), de esta forma, *Volumen_Aciertos* es la suma de los tamaños de los documentos referenciados que han causado un acierto en *caché*. *Volumen_Total_Peticones* es la suma de los tamaños de todos los documentos pedidos y *Volumen_RechazadosOk* es la suma de los tamaños de los documentos correctamente rechazados.

Estas dos métricas proporcionan una medida más precisa del rendimiento de la *caché* al tener en cuenta el efecto de la política de acceso. Cuando no se hace uso de una

política de acceso estas métricas arrojan el mismo resultado que HR y BHR ya que *#RechazadosOk* y *Volumen_RechazadosOk* serían nulos.

El principal problema de estas dos métricas es cómo determinar si un documento ha sido correctamente rechazado o no por la política de acceso. Una primera aproximación puede ser considerar que un documento ha sido bien rechazado cuando sólo ha sido pedido una vez, es decir la vez que ha sido rechazado, también cuando ha sido modificado desde la última vez que fue referenciado. En este último caso habría que volver a traerlo desde el servidor remoto y no se consideraría acierto, por eso forma parte del grupo de los correctamente rechazados.

Pero se puede afinar más en la determinación de si un documento ha sido correctamente rechazado en el caso de que se estén usando determinadas políticas de reemplazo. Las políticas de reemplazo para las que se puede hacer esto son LRU y LFU. Esto es posible debido al hecho de que en estas políticas la posición en la que se coloca un nuevo documento cuando entra en *caché* está determinada de antemano. En LRU los nuevos documentos se introducen al final de la lista utilizada como estructura de almacenamiento, en LFU los nuevos documentos se introducen después de los documentos que tienen una frecuencia de referencia igual a uno. En LFU los documentos en la lista están ordenados de menor a mayor frecuencia. En las políticas de tipo *Greedy* un nuevo documento puede ser introducido en cualquier posición de la lista.

Para la política LRU la mejora en el parámetro que mide los documentos rechazados correctamente se hace de la siguiente forma: si entre una petición de un documento de los que se rechazan y la siguiente petición de dicho documento no se han producido aciertos, y la distancia en Bytes entre ambas es mayor que el tamaño de la *caché*, podemos afirmar que el documento se ha rechazado correctamente. Si se producen aciertos, para afirmar que el documento ha sido bien rechazado, la distancia entre peticiones tiene que ser superior al tamaño de la *caché* más el tamaño de los aciertos.

En la figura 2.23 podemos ver un ejemplo que explica cuando podemos considerar correctamente rechazado un documento si entre una petición de éste y la siguiente no se producen aciertos en la *caché*. En el ejemplo, por simplicidad, se ha considerado que todos los documentos tienen el mismo tamaño. Se puede apreciar en la figura que si se introduce el documento con identificador 4 en *caché* y entre una petición y la siguiente hay tantas peticiones de otros documentos como para hacer uso de todo el tamaño de la *caché*, el documento 4 saldrá de *caché*. Cuando se pide de nuevo dicho documento habrá que traerlo del servidor remoto, por tanto no resulta útil introducirlo en *caché*.

En la figura 2.24 se explica, mediante un ejemplo, el caso de que medien aciertos entre una petición y la siguiente del documento que estamos considerando rechazar (documento con identificador 4). Se puede observar que cuando hay aciertos en documentos que han sido introducidos después del documento 4, la posición de este último en la *caché* no varía (instante 4) de la figura 2.24). En cambio, se va acercando a la salida de la *caché* cuando los aciertos se producen en documentos que están delante de él (instante 5)) y cuando se introducen nuevos documentos (instante 6)). El caso peor lo constituye la situación en la que todos los aciertos se produzcan en documentos que hayan sido introducidos con posterioridad (los situados detrás de él). Por tanto para asegurar que se ha rechazado correctamente un documento en esta situación la distancia entre sus referencias tiene que ser superior al tamaño de la *caché* más el de los documentos que han constituido aciertos.

En el caso de LFU hay que tener en cuenta que un nuevo elemento sería añadido al final de la lista de elementos con el peso más bajo, en LFU el peso lo determina directamente la frecuencia de referencia al documento. La forma de determinar si ha sido conveniente rechazar un documento será como en LRU, salvo que en este caso sustituiremos el tamaño de *caché* por un término que sería la suma del espacio vacío en *caché* más el propio elemento más todos los elementos que hay por delante de él.

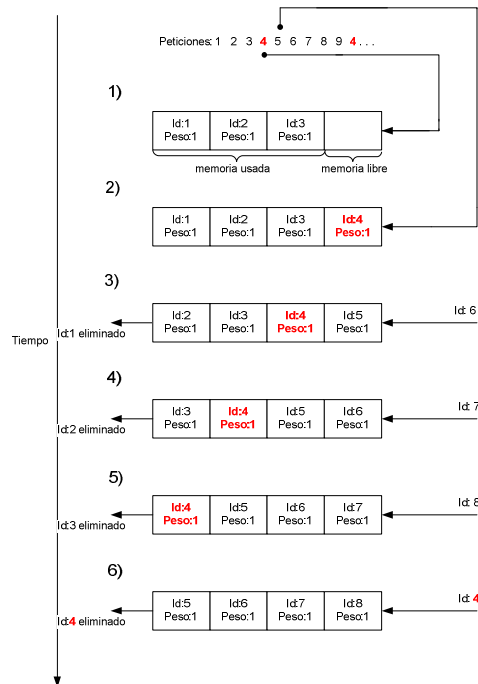


Figura 2.23. Mejora del parámetro *RechazadosOk* para LRU sin aciertos entre peticiones

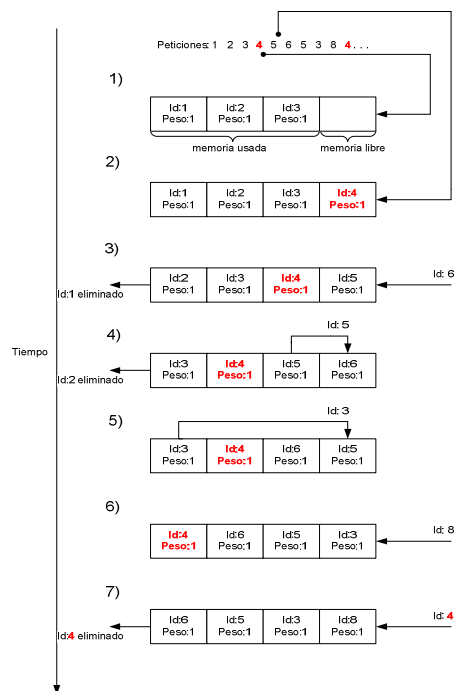


Figura 2.24. Mejora del parámetro *RechazadosOk* para LRU con aciertos entre peticiones

Se puede entender mejor esto último mediante los ejemplos de las figuras 2.25 y 2.26. En la figura 2.25 se muestra el caso de que no haya aciertos entre una petición y la siguiente del documento con identificador 4. Como en LFU cuando se introduce un documento no se hace al final de la lista no es necesario esperar a que haya un número

de peticiones equivalentes al tamaño de la *caché* para considerar si habría sido correctamente rechazado. Se observa que para que el documento 4 sea eliminado de la *caché* se tiene que introducir 4 nuevos documentos, es decir el espacio correspondiente a el mismo más todo lo que hay delante de él más la memoria libre.

En la figura 2.26 se muestra el caso de que haya aciertos entre dos peticiones consecutivas del documento que estamos considerando. Se observa que, como en el caso LRU, el documento 4 sólo cambia de posición cuando los aciertos se producen en documentos situados delante de él y cuando se introducen nuevos documentos. La posición queda inalterada siempre que los aciertos se produzcan en documentos introducidos con posterioridad o en documentos referenciados en más de una ocasión, es decir en documentos situados detrás de él.

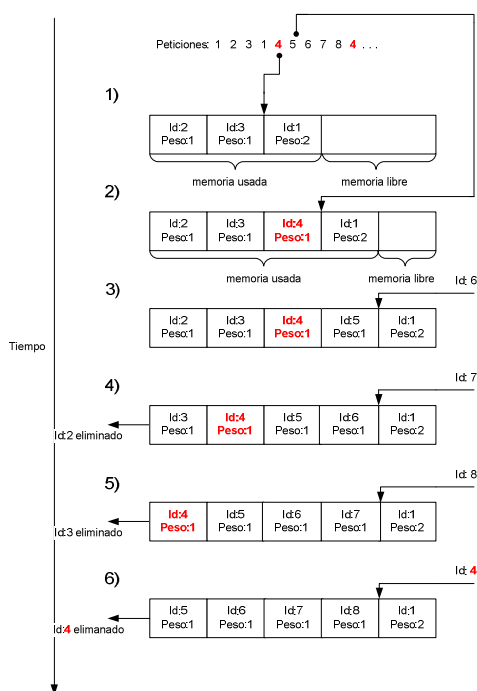


Figura 2.25. Mejora del parámetro *RechazadosOk* para LFU sin aciertos entre peticiones

Para el resto de políticas de reemplazo y en los casos en los que no se cumplen las condiciones anteriores, asumiremos que un documento es correctamente rechazado si sólo es pedido una vez o cuando ha sido modificado desde la última vez que fue referenciado.

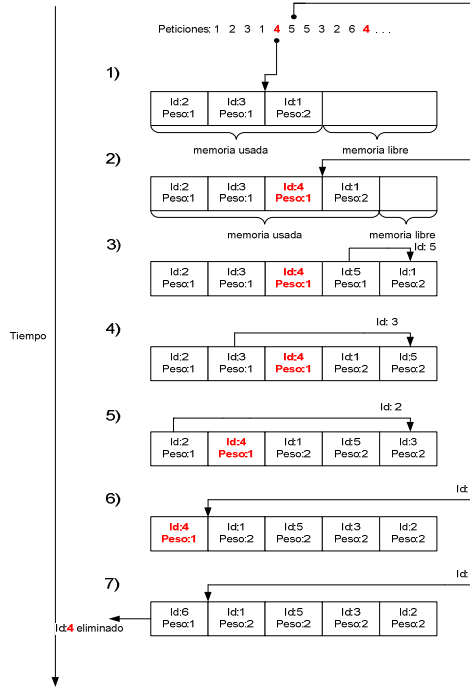


Figura 2.26. Mejora del parámetro *RechazadosOk* para LFU con aciertos entre peticiones

2.4.2. ACCESS CONTROL HIT RATIO (ACHR) Y ACCESS CONTROL BYTE HIT RATIO (ACBHR)

Estas dos métricas tratan de dar una idea del rendimiento de la política de acceso implementada. Las expresiones que definen estas métricas (ecuaciones 2.6 y 2.7) se dividen en dos partes, se mide la tasa de documentos correctamente rechazados por un lado, y la tasa de documentos correctamente admitidos por otro.

$$ACHR = \sqrt{\frac{\#RechazadosOk}{\#Total_Rechazados} \cdot \frac{\#AceptadosOk}{\#Total_Aceptados}} \quad (2.6)$$

$$ACBHR = \sqrt{\frac{Volumen_Rechazados_Ok}{Volumen_Total_Rechazados} \cdot \frac{Volumen_AceptadosOk}{Volumen_Total_Aceptados}} \quad (2.7)$$

Los nuevos términos que aparecen en estas ecuaciones son los siguientes: *#Total_Rechazados*, es el número total de peticiones de documentos rechazados por la caché; *#AceptadosOk*, es el número de peticiones que causan un acierto en la caché; *#Total_Aceptados* es el número de peticiones de documentos que pueden entrar en

caché; *Volumen_Total_Rechazados* es la suma del tamaño de todos los documentos rechazados; *Volumen_AceptadosOk* es la suma del tamaño de los documentos referenciados que han causado un acierto; por último, *Volumen_Total_Aceptados* es la suma del tamaño de todos los documentos aceptados por la política de acceso a *caché*.

Estas métricas proporcionan una idea de la bondad de la política de acceso a *caché*, considerando tanto los documentos rechazados como los aceptados y valorando la conveniencia de incluirlos en uno u otro grupo. Así, una buena política de acceso debería maximizar el número de peticiones correctamente aceptadas, las que dan lugar a un acierto, y el número de peticiones correctamente rechazadas, las que corresponden a documentos que no causarían aciertos mientras estuvieran almacenados en *caché*.

Los valores que arrojan las ecuaciones descritas anteriormente se multiplican por 100 para obtener así valores porcentuales.

2.5. PROCESAMIENTO ESTRUCTURADO POR LOTES

Se ha implementado en el emulador una opción que permite la adquisición de los parámetros de entrada desde un fichero, permitiendo un procesamiento por lotes. Se conoce como procesamiento por lotes el modo de funcionamiento de un programa que se ejecuta de forma no interactiva sobre una gran cantidad de datos. De esta forma podemos realizar múltiples emulaciones sucesivas sin tener que ingresar manualmente nuevos parámetros de entrada cada vez que termine una emulación.

El principal inconveniente de la ejecución por lotes frente a la ejecución interactiva es que hay que conocer y planificar cuidadosamente todo lo que hay que hacer, ya que al ser generalmente tareas que se ejecutan sin supervisión, los resultados pueden ser inútiles o simplemente inexistentes debido a un error de previsión.

2.5.1. LENGUAJES DE MARCAS

El término “marca” alude a todo aquello que en un documento no constituye el contenido en sí mismo. El marcaje de documentos electrónicos puede ser de dos tipos: marcaje procedural y marcaje descriptivo.

El marcaje procedural consiste en una serie de códigos de formateo que se intercalan el texto del documento y que se refieren al modo de presentarlo. Por ejemplo, códigos para indicar el tamaño de la letra o el tipo de fuente. El marcaje procedural no proporciona la capacidad de definir automáticamente la apariencia de la información en otros tipos de formato distintos a aquel para el ha sido definido. Si se requiere un cambio de estilo, hay que volver a formatear masivamente el documento.

El marcaje descriptivo describe la función de un texto dentro del documento del que constituye una parte y no su apariencia. Se basa en identificar los diferentes elementos que integran la estructura del documento y utilizar una notación para indicar a qué elemento corresponde cada parte del texto. De este modo, se facilita que la misma información sea presentada en diferentes formatos.

SGML (*Standard Generalized Markup Language*) es una norma internacional (ISO 8879) publicada en 1986 que especifica un método normalizado para describir la estructura de un documento. El marcaje de documentos que se realiza mediante SGML es, por tanto, de tipo descriptivo. Cada elemento de la estructura se marca con una etiqueta que indica su función. Las etiquetas se identifican porque comienzan por el símbolo “<” y terminan con “>”.

2.5.2. DTD (*DOCUMENT TYPE DEFINITION*)

La DTD describe las reglas formales que deben cumplir los componentes de un tipo concreto de documento y las relaciones entre los mismos, por ejemplo el orden en que deben aparecer dichos componentes. La estructura del documento es descrita de

forma similar a cómo se definen en una base de datos los campos de cada registro la relación entre ellos.

La DTD acompaña siempre al documento escrito en el lenguaje de marcas. Los documentos suelen comenzar con una línea de declaración de tipo de documento, que puede incluir una referencia a una DTD. Dicha referencia puede ser interna, si el documento incluye la DTD, o externa, cuando hace alusión a una DTD contenida en otro fichero.

2.5.3. XML (*EXTENSIBLE MARKUP LANGUAGE*)

Se ha elegido XML (*eXtensible Markup Language*) como formato para el fichero de entrada ya que proporciona una forma ordenada y elegante de representar la información.

XML es un metalenguaje extensible de etiquetas desarrollado por el *World Wide Web Consortium* (W3C). Es una simplificación y adaptación del SGML que permite definir la gramática de lenguajes y puede ser usado para el intercambio de información.

XML se caracteriza por no poseer etiquetas prefijadas con anterioridad, sino que es el propio diseñador del documento quien las crea, dependiendo de cual vaya a ser el contenido.

Un documento XML está formado por las siguientes partes y entidades:

- **Prólogo:** aunque no es obligatorio, los documentos XML pueden empezar con unas líneas que describen la versión XML y el tipo de documento.
- **Cuerpo:** a diferencia del prólogo, el cuerpo no es opcional en un documento XML, el cuerpo debe contener un elemento raíz, característica indispensable también para que el documento esté “bien formado”. Se define más adelante qué se entiende por documento “bien formado”.

- **Elementos:** los elementos XML pueden tener contenido (más elementos, caracteres, o ambos), o bien estar vacíos.
- **Atributos:** los elementos pueden tener atributos, que son una manera de incorporar características o propiedades a los elementos de un documento.
- **Entidades predefinidas:** entidades para representar caracteres especiales y que no sean interpretados como marcaje en el procesador XML.
- **Comentarios:** comentarios a modo informativo para el programador que han de ser ignorados por el procesador. Los comentarios tienen el siguiente formato:
`<!-- Esto es un comentario -->`.

Los documentos XML se almacenan en formato de texto ASCII, generalmente en archivos con la extensión “.xml”.

Un intérprete de documentos XML toma el archivo .xml y analiza sintácticamente el documento para verificar que se ajusta a la gramática XML y a las normas del DTD. Esta tarea es ejecutada por un analizador sintáctico o *parser*. Éste puede operar básicamente de dos formas distintas: *parser* DOM (*Document Object Model*) y *parser* SAX (*Simple API for XML*).

DOM opera generando internamente una estructura en forma de árbol jerárquico que organiza los elementos descritos por las etiquetas. Este árbol jerárquico de información en memoria permite que a través del *parser* sea manipulada la información.

SAX procesa la información por eventos. Procesa la información XML conforme la va leyendo del fichero correspondiente. SAX es un *parser* ideal para manipular archivos de gran tamaño, ya que no hace tanto uso de memoria como DOM al no generar un árbol en memoria. Además, es más rápido y sencillo que utilizar DOM. El inconveniente de SAX es que al funcionar por eventos, no es posible manipular la información una vez procesada. En DOM no existe esta limitación ya que es posible ir a los nodos del árbol jerárquico para modificarlos.

Se dice que un documento XML está “bien formado” cuando cumple las especificaciones de XML, por ejemplo, cuando los anidamientos de los elementos son correctos. Por otra parte, un documento XML se considera “válido” cuando cumple con las reglas de una DTD determinada, por ejemplo, cuando los atributos que contiene un determinado elemento tiene valores especificados como válidos por la DTD.

2.5.4. IMPLEMENTACIÓN DEL MÓDULO XML EN EL EMULADOR

En primer lugar se ha usado Altova XMLSpy 2007 Enterprise Edition [AltovaXMLSpy] para diseñar la estructura que deben tener los ficheros XML que proporcionan los parámetros de entrada al emulador, también para crear la DTD. Altova XMLSpy es un entorno de desarrollo XML con diversas herramientas para la edición y definición de documentos XML.

A partir del XML generamos la DTD con Altova, ya que este entorno de desarrollo ofrece esta posibilidad. En la figura 2.27 se muestra, a modo de ejemplo, el aspecto de un fichero válido XML en el editor de esta aplicación. Se puede apreciar que se implementan dos *batch*. En el primero se admiten los documentos de tipo imagen y de tipo sonido, para los primeros la política de reemplazo a utilizar es GD* y para los segundos, LRU. Se introduce también el tamaño de la *caché* usado por cada uno de ellos. En el segundo *batch* se admiten sólo los documentos de tipo imagen utilizándose como política de reemplazo LFU-DA.

Una vez tenemos el esquema que tienen que seguir los ficheros XML de entrada y generada la DTD tenemos que implementar las clases en Java que se encargan del procesamiento del documento XML. En la figura 2.28 se muestra la DTD generada, de las distintas vistas que proporciona Altova se ha elegido el formato *Grid* por constituir una forma clara de presentar la información.

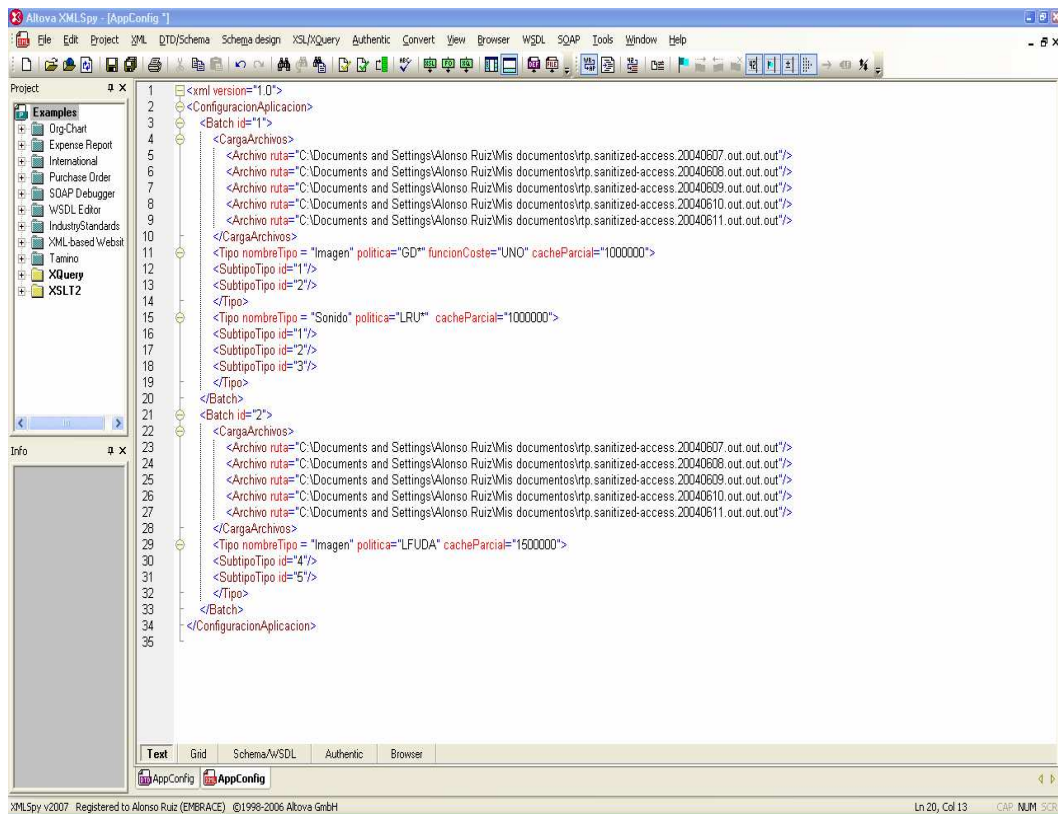


Figura 2.27. Fichero XML mostrado por el editor de Altova XMLSpy

Para este proceso se ha elegido un *parser* de tipo SAX por su facilidad de uso. Se trata de un *parser* incluido en JAXP (*Java API for XML Processing*) proporcionado por el entorno de desarrollo Java que hemos usado: *Borland JBuilder X Enterprise* [BorlandJBuilder]. JAXP también proporciona un *parser* DOM y otras herramientas para el tratamiento de documentos XML.

La interfaz *ContentHandler* es el centro de todo proceso de XML con SAX. Es la que define todos los eventos que se producen a lo largo del procesamiento del documento. Las clases que necesitamos tienen que implementar esta interfaz para que el *parser* vaya pasando los parámetros de entrada al emulador desde el documento XML.

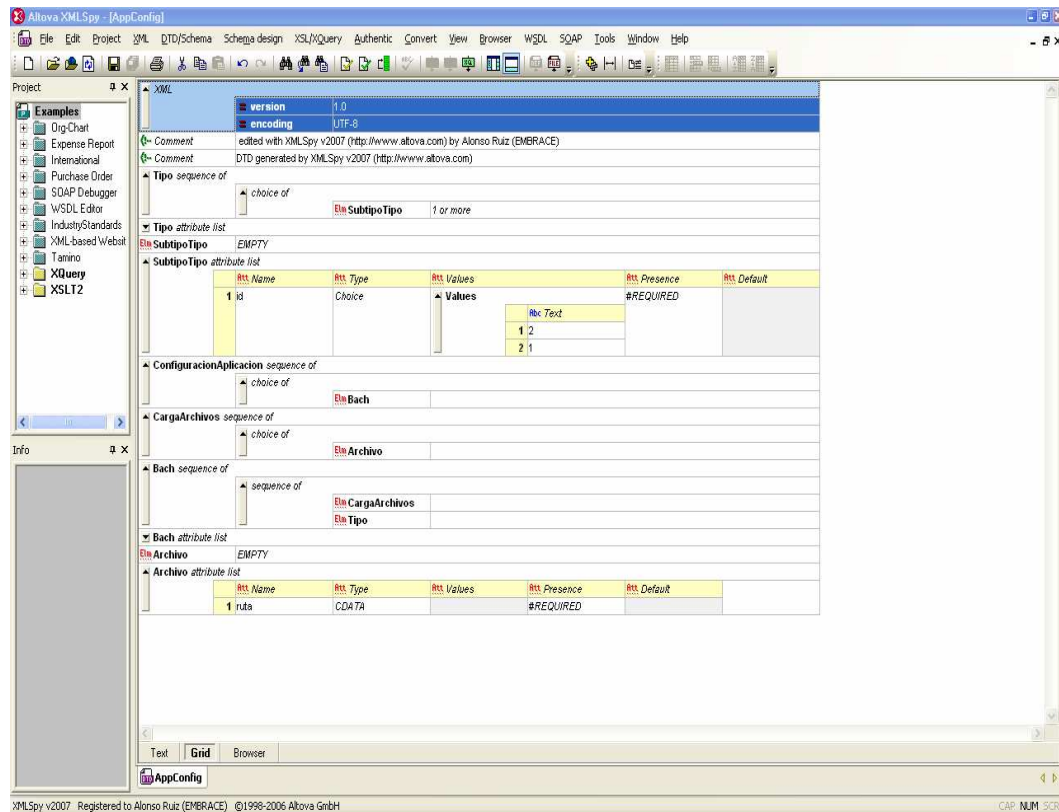


Figura 2.28. DTD generada por Altova XMLSpy

Existe una clase que implementa esta interfaz, se trata de la clase *DefaultHandler*. Es una clase que proporciona una serie de métodos vacíos, nosotros la extendimos (*XMLHandler extends DefaultHandler*) y sobrescribimos los métodos necesarios.

Como se ha dicho con anterioridad, SAX se basa en eventos, eventos que se producen durante el procesamiento del fichero XML, del estilo de inicio/fin del documento, inicio/fin de un elemento, datos entre etiquetas, etc., así pues los métodos de *ContentHandler* son del estilo *startDocument*, *endDocument*, *startElement*, *endElement*, *characters*, etc., que son llamados cuando ocurre el suceso correspondiente. Además a la clase extendida *XMLHandler* se le añadió los métodos necesarios para capturar la información contenida en el fichero XML, y para manejar las excepciones que se producen cuando el fichero XML no es válido.

2.5.5. ESTRUCTURA DEL FICHERO XML

A continuación se describe cual debe ser la estructura que debe seguir un documento XML “bien formado” y “válido” para poder ser usado como fuente de parámetros de entrada.

Lo primero que hay que observar es la primera línea. Con ella deben empezar todos los documentos XML, ya que es la que indica que lo que sigue es XML. Aunque es opcional, es recomendable incluirla. Cuando se incluye, hay que tener en cuenta que el atributo *version* es obligatorio, indica la versión de XML usada en el documento. El resto de atributos que puede contener esta línea son opcionales, uno de ellos es *encoding*. Éste especifica la forma en que se ha codificado el documento.

El elemento raíz, del que dependen todos los demás, está delimitado con las etiquetas de inicio `<ConfiguracionAplicacion>` y de fin `</ConfiguracionAplicacion>`. Este elemento contiene los *batch*, se pueden incluir tantos como sea necesario. La etiqueta de inicio de cada *batch* ha de incluir obligatoriamente un atributo *id* que identifica numéricamente el *batch*, el valor de este atributo se define como CDATA. Mediante esta construcción se permite especificar datos, utilizando cualquier carácter, especial o no, sin que se interprete como marcado XML.

Cada *batch* contiene obligatoriamente un elemento *CargaArchivos* y tantos elementos *Tipo* como tipos de documentos permita la política de acceso a *caché*. *CargaArchivos* contiene elementos de tipo *Archivo*, tantos como archivos de muestras de tráfico queramos cargar. Las rutas de los distintos archivos de muestras se introducen a través del atributo *ruta* de este último elemento mediante una cadena de caracteres definida como CDATA. *Archivo* es un tipo de elemento de tipo vacío, no contiene otros elementos, esto quiere decir que sólo necesita una etiqueta en la que aparece el nombre del elemento, *Archivo*, y el atributo, *ruta*. Al no tener una etiqueta de cierre que delimite un contenido, se utiliza la forma `<etiqueta/>`.

El elemento *Tipo* se define mediante la serie de atributos siguiente: *nombreTipo*, *cacheParcial*, *politica*, *funcionCoste*. Además contiene tantos elementos *SubtipoTipo* como subtipos permita la política de acceso a *caché*. El atributo *nombreTipo* identifica el tipo de documento, puede ser uno de entre los siguientes: *Video*, *Audio*, *Texto*, *Aplicación*, *Imagen*. El atributo *cacheParcial* ha de contener el tamaño reservado en *caché* para el tipo de documento correspondiente, se expresa en Bytes. La política de reemplazo se introduce mediante el atributo *politica*, los valores permitidos son: *LRU*, *LFU*, *LFUDA*, *GDS*, *GDSF* y *GD**. Por último, la función de coste se introduce mediante el atributo *funcionCoste*, es un atributo opcional que admite dos valores: *UNO* y *PACKETS*.

SubtipoTipo es un elemento vacío que contiene un atributo que identifica numéricamente un subtipo permitido para el *Tipo* del que forme parte. La etiqueta de este elemento tiene el siguiente aspecto: `<SubtipoTipo id="1" />`.

CAPÍTULO 3: Caracterización del tráfico

En este capítulo se estudian las características de las muestras de tráfico Web usadas en este Proyecto Fin de Carrera. En el primer apartado se describe la procedencia de las muestras usadas y la estructura de los archivos que las contienen. A continuación se analiza la composición del tráfico en cuanto a número de peticiones y volumen de información solicitada para cada tipo de documento. En el tercer apartado se realiza una caracterización de la localidad temporal de las referencias y de la popularidad de los documentos solicitados, para ello se estudia la frecuencia y la distancia entre peticiones de documentos que se piden en más de una ocasión. Por último se trata de establecer una relación entre la frecuencia, la distancia entre peticiones y el tamaño de los documentos.

3.1. MUESTRAS DE TRÁFICO EMPLEADAS

Como se mencionó en el primer capítulo, las muestras empleadas fueron tomadas de la página Web del proyecto IRCache [IRCache]. De entre las muestras de tráfico que proporciona esta página se han usado para este estudio muestras tomadas del NLANR (*National Laboratory for Applied Network Research*) [NLANR], de una *caché proxy* de primer nivel situada en el *Research Triangle Park*, en Carolina del Norte (Estados Unidos).

Los archivos descargados de IRCache fueron sometidos a un preprocesado para acondicionar las muestras al tipo de estudio que había que realizar. En primer lugar esto supuso quedarse sólo con las respuestas a peticiones GET. GET es un método definido en el protocolo HTTP que permite solicitar un documento al servidor remoto que se devuelve como cuerpo de entidad en la consiguiente respuesta. De éstas, se eliminaron

las peticiones correspondientes a comunicaciones entre las distintas *caché* de la jerarquía del proyecto IRCache. Estas peticiones se distinguen porque presentan la cadena “:3128” ya que la comunicación se realiza a través del puerto 3128. También se eliminaron aquellas entradas que presentaban las cadenas “.cgi”, “.cgi-bin” o “?”, por corresponder a documentos dinámicos.

En los archivos que contienen las muestras de tráfico, cada línea representa una petición de un documento. De entre todos los campos que, originalmente, presenta cada línea, se seleccionan para este estudio los siguientes:

- Marca de tiempo: hora a la que el *socket* del usuario es cerrado. Tiene formato Unix y se expresa en milisegundos.
- Tamaño: número de Bytes enviados al usuario.
- URL: el URL (*Uniform Resource Locator*) solicitado.
- Tipo de contenido: el campo *Content-type* procedente de la respuesta HTTP.

Por último, el campo URL se sustituyó por un número único para cada URL, que constituye el identificador de cada documento pedido. El campo de tipo de contenido fue separado en dos campos. El formato del tipo de contenido es “tipo/subtipo”, la primera parte pasó a ser el nuevo campo tipo y la segunda, el campo subtipo. La codificación asignada a cada tipo se puede observar en la tabla 2.2 del capítulo anterior, y la de los distintos subtipos, en el Apéndice A.

Por tanto cada línea en los archivos de muestras pasa a tener un formato de cinco campos: tiempo, tamaño, identificador, tipo y subtipo de documento. El aspecto de estas líneas se puede apreciar en la figura 3.1 donde se muestra un extracto de uno de los archivos de muestras. En la primera línea, por ejemplo, se pide en el instante temporal determinado por el primer campo el documento con identificador 29, de 617 Bytes, de tipo 2, es decir Imagen, y de subtipo 1, es decir GIF.

```

...
1086566419.846 617 29 2 1
1086566419.923 2455 30 2 3
1086566419.952 1861 31 2 3
1086566419.986 1121 32 4 4
1086566420.007 6679 3 4 6
...

```

Figura 3.1. Formato de las peticiones contenidas en los archivos de muestras

3.2. NÚMERO DE PETICIONES Y VOLUMEN DE DATOS PARA CADA TIPO DE DOCUMENTO

El número total de peticiones contenidas en los archivos de muestras tras el preprocesado es de 4040036 y el volumen total de datos pedidos es de 40.4 GBytes. En la tabla 3.1 se muestra la distribución del número de peticiones y del volumen de datos pedidos por tipo de documento.

	Peticiones	%	Volumen Datos (KBytes)	%
Peticiones de tipo Aplicación :	326589	8.08	13.5E+06	33.39
Peticiones de tipo Audio :	11450	0.28	1.40E+06	3.46
Peticiones de tipo Imagen :	3051832	75.54	14.7E+06	36.36
Peticiones de tipo Texto :	637249	15.77	9.35E+06	23.13
Peticiones de tipo Video :	4964	0.12	1.29E+06	3.19
Total:	4032084	99.80	40.2E+06	99.54
Otros tipos:	7952	0.19	0.18E+06	0.46

Tabla 3.1. Número de peticiones y volumen de datos por tipo de documento

Se puede observar que la mayor parte de peticiones corresponde a documentos de tipo Imagen, de hecho constituyen más del 75% de las peticiones y más del 36% del volumen total de información solicitada. Después de éstos, los documentos más solicitados son los de texto con casi un 16% del total de peticiones, les siguen las aplicaciones con algo más de un 8%. Los tres tipos suponen más del 99% del total de peticiones y prácticamente el 93% del tráfico total solicitado. Entre las muestras usadas se encuentran peticiones que corresponden a tipos que no se han tenido en cuenta en el estudio debido a la poca relevancia que tienen respecto al total de peticiones, todas ellas

apenas suponen un 0.2% del total de peticiones y un 0.5% del total del tráfico solicitado.

En la tabla 3.2 se muestra el número de peticiones y el volumen de información solicitada por subtipo de documento para el tipo Imagen por ser el tipo predominante tanto en número de peticiones como en volumen de tráfico generado.

Imagen (Subtipos)	Peticiones	%	Volumen Datos (Bytes)	%
bmp:x-bitmap-ms	2287	0.07	50700000	0.35
gif	2147969	70.38	5820000000	39.64
ico:icon:icons:x-ico	1666	0.05	4191984	0.03
jpg:jpeg:pjpeg	886629	29.05	8718929039	59.38
nids:nids-mosaic	195	0.01	6100345	0.04
png:x-png	12603	0.41	79373583	0.54
swf	30	≈ 0	753303	0.01
iff	13	≈ 0	163444	≈ 0
vnd	0	0	0	0
vnd.dwg	1	≈ 0	635480	≈ 0
vnd.nok-oplogo-color	0	0	0	0
vnd.rn-realpix	1	≈ 0	768	≈ 0
vnd.wap.bmp	0	0	0	0
x-binary	0	0	0	0
x-corelphotopaint	115	≈ 0	176948	≈ 0
x-guffaw	1	≈ 0	5202	≈ 0
x-hotmedia	2	≈ 0	712082	≈ 0
x-portable-graymap	0	0	0	0
desconocido	320	0.01	735034	0.01
TOTAL:	3051832	100	1.47E+10	100

Tabla 3.2. Peticiones y volumen de datos por subtipo para el tipo Imagen

El subtipo más común es el formato GIF (*Graphics Interchange Format*), es un formato gráfico utilizado ampliamente en la World Wide Web tanto para imágenes como para animaciones. Aunque, como se puede observar, el 70% de peticiones del tipo Imagen son de este subtipo, no constituye el subtipo que genera un mayor tráfico de datos, ya que el volumen de datos generado por las peticiones de este subtipo supone algo más del 39% del total de datos frente al subtipo JPEG (*Join Photographic Experts Group*) y derivados que representa más del 59% del total del volumen de datos, aunque el número de peticiones de estos subtipos supone el 29% del total de peticiones.

GIF es un formato sin pérdida de calidad para imágenes con hasta 256 colores. Por ese motivo, con imágenes con más de 256 colores (profundidad de color superior a 8 bits), la imagen debe adaptarse reduciendo sus colores, produciendo la consecuente pérdida de calidad. Por su parte, JPEG es capaz de utilizar una paleta de colores de más de un millón de tonalidades, por lo que es el formato más conveniente para imágenes fotográficas o diseños con gran cantidad de información. Es por este motivo que los documentos de este subtipo suelen ser mucho más pesados en términos de tamaño que los del subtipo GIF.

En la tabla 3.3 se muestran los estadísticos básicos para cada uno de los tipos de documentos. Se puede observar que los documentos del tipo Imagen son los que tienen un tamaño medio menor, 4 KBytes, y también los que presentan una desviación típica menor, 21 KBytes.

	Aplicación	Audio	Imagen	Texto	Video
Media (KBytes)	41	123	4	14	260
Mediana (KBytes)	3	7	1	3	573
Desv. Típica (KBytes)	761	948	21	104	974

Tabla 3.3. Estadísticos por tipo de documento

3.3. LOCALIDAD TEMPORAL: POPULARIDAD DE LOS DOCUMENTOS Y CORRELACIÓN TEMPORAL

La localidad temporal en las peticiones Web establece que un documento referenciado en un determinado instante tenderá a ser referenciado en un futuro próximo. Está caracterizada por dos propiedades: la popularidad de los documentos y la correlación temporal entre las peticiones. Cuantificar ambas fuentes de la localidad temporal es importante porque permite usar políticas de reemplazo adaptativas que

tienen en consideración ambas propiedades, como GD*, y tomar decisiones acerca de a qué documentos conviene permitir el acceso a *cache*.

Como se puede observar en la tabla 3.4, las peticiones hacen referencia a un total de 2185260 documentos distintos. La mayor parte de ellos se piden una sola vez, casi el 82% de ellos, y prácticamente un 6% son referenciados más de tres veces. Se ha estudiado por separado la popularidad para los documentos de los dos tipos más solicitados, el tipo Imagen y el tipo Texto. Como se puede apreciar, la distribución que resulta es similar a la que proporciona contabilizar todas las peticiones.

	Del Total	%	De Tipo Imagen	%	De Tipo Texto	%
Documentos que se piden 1 vez:	1789740	81.90	1389757	82.17	291155	81.81
Documentos que se piden 2 veces:	196157	8.97	148966	8.81	34337	9.65
Documentos que se piden 3 veces:	71947	3.29	54853	3.24	12084	3.39
Documentos que se piden más de 3 veces:	127416	5.83	97785	5.78	18329	5.15
Total de Documentos	2185260	100	1691361	100	355905	100

Tabla 3.4. Popularidad de los documentos

Otra forma de caracterizar la popularidad es relacionar el número de referencias de un documento con el rango de popularidad del mismo. El documento más pedido se situaría en el ranking de popularidad en la primera posición, tendría un rango de popularidad igual a 1. Diversos estudios han concluido que esta relación sigue aproximadamente una ley de tipo Zipf [Barford'99] [Cao'99] [Bestavros'99]. Esta ley, desarrollada por George Kingsley Zipf (1902-1950), fue aplicada inicialmente al análisis estadístico de diferentes lenguas y afirma que un pequeño número de palabras son utilizadas con mucha frecuencia, mientras que frecuentemente ocurre que un gran número de palabras son poco empleadas. Los trabajos de Zipf pueden ser utilizados para estudiar las propiedades estadísticas de grandes conjuntos de datos, como el tráfico de datos en la World Wide Web.

Según esta ley, la probabilidad de pedir un documento con rango de popularidad i es proporcional a $i^{-\alpha}$. El parámetro α es la pendiente de la representación, en escala logarítmica de ambos ejes, del número de referencias de los documentos en función del

rango de popularidad u orden de frecuencia de los mismos. Para tráfico Web tiene un valor típico comprendido entre 0.6 y 0.7. En la figura 3.2 se muestra dicha distribución de popularidad para las muestras de tráfico usadas. Para el cálculo de la pendiente de la recta de regresión se ha usado el método de mínimo error cuadrático medio, obteniéndose un resultado de $\alpha=0.64$.

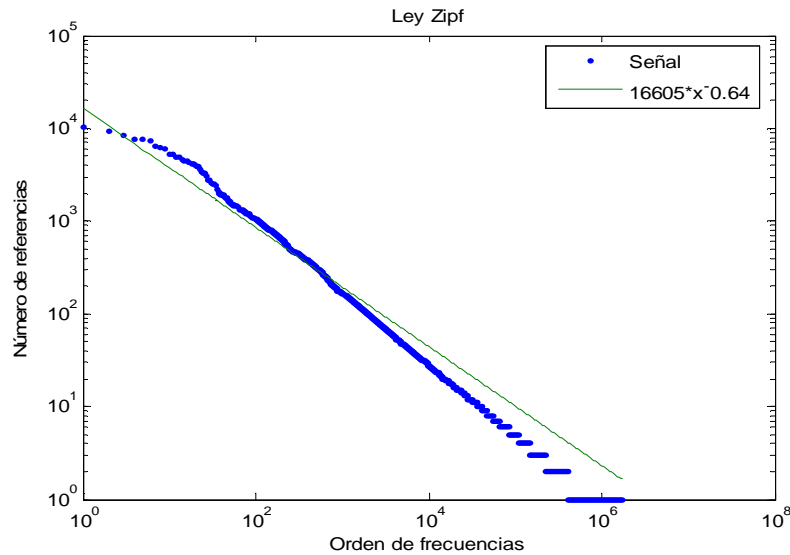


Figura 3.2. Distribución de popularidad

Por otro lado, la localidad temporal puede ser obtenida relacionando la probabilidad de que se produzca una referencia con el tiempo que ha transcurrido desde que se produjo la última. Breslau llegó a la conclusión de que la probabilidad de que un documento sea pedido un tiempo t después de la última referencia es proporcional a $1/t$ [Breslau'99].

Para obtener una medida de la localidad temporal en las muestras de tráfico que tenemos se ha medido la distancia entre referencias a un mismo documento, tanto en términos de peticiones como en volumen de datos entre ellas (método propuesto por Breslau). Los resultados obtenidos se pueden observar en las gráficas de las figuras 3.3 y 3.4.

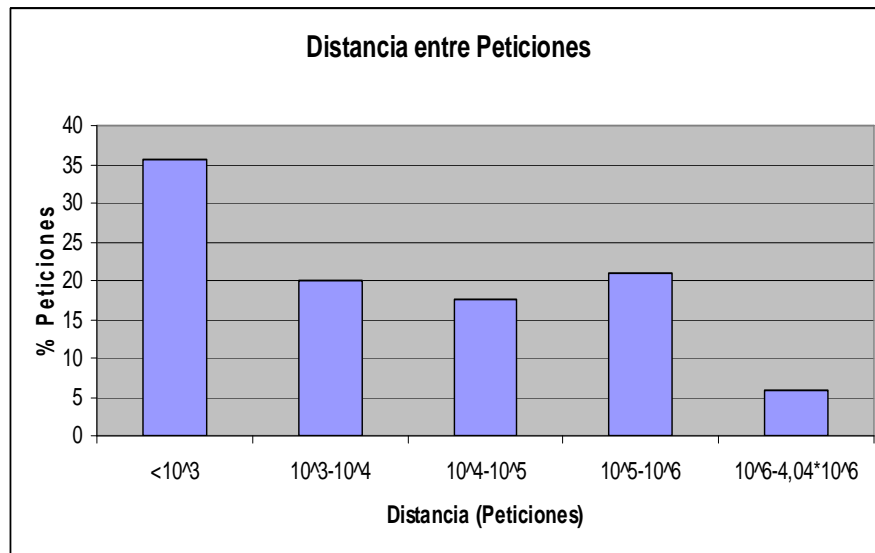


Figura 3.3. Distancia entre peticiones de documentos que se piden más de una vez (en peticiones)

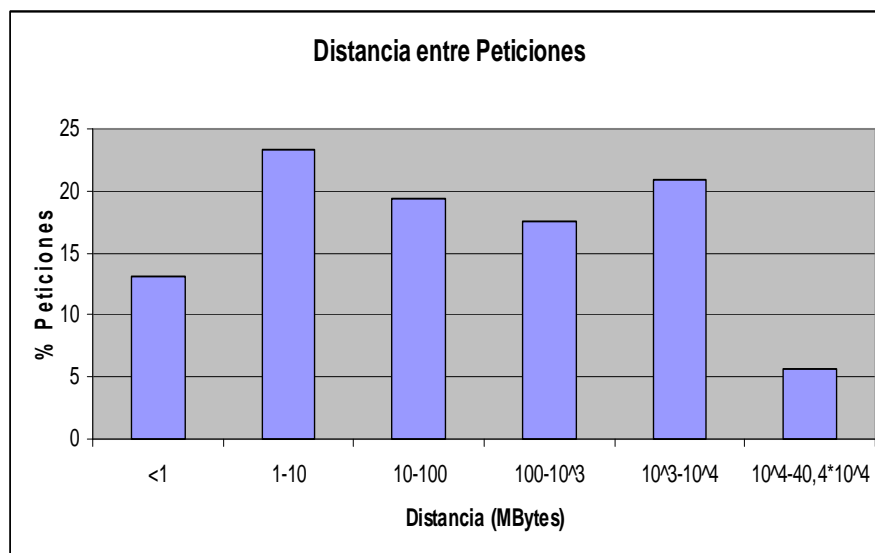


Figura 3.4. Distancia entre peticiones de documentos que se piden más de una vez (en MBytes)

Estas gráficas muestran la probabilidad de que un documento sea referenciado de nuevo en función de la distancia a la última referencia. Se puede observar que la mayor parte de peticiones correspondientes a documentos que se piden en más de una ocasión ocurren antes de las mil peticiones, aproximadamente un 36%. Por otro lado, aproximadamente para el 24% de las peticiones de este tipo la referencia anterior se encuentra a una distancia, en volumen de datos solicitados, de entre 1 y 10 MBytes.

En la figura 3.3 se puede apreciar como la probabilidad de pedir de nuevo un determinado documento aproximadamente decrece a medida que nos alejamos de la referencia, este resultado está en concordancia con el modelo propuesto por Breslau.

Estas medidas hacen referencia a la localidad temporal y por tanto en ellas están implícitas tanto la popularidad de los documentos como la correlación temporal de las peticiones. Los documentos más populares tienden a ser referenciados frecuentemente y de esta forma mostrarán una distancia entre peticiones menor que aquellos documentos menos populares y, por tanto, menos referenciados.

Shudong Jin y Azer Bestavros [Bestavros'99] determinaron que el peso que tiene la popularidad en la localidad temporal es mucho mayor que el que tiene la correlación temporal en el modelo propuesto por Breslau. Para ello midieron la distancia entre referencias desordenándolas previamente, con esto pretendían independizar la localidad temporal de la correlación temporal, sin embargo los resultados obtenidos fueron similares a los que se obtienen sin desordenar las referencias.

Para obtener la correlación temporal se debe eliminar el efecto de la popularidad sobre la localidad temporal. Para ello, se ha de medir la distancia entre peticiones de documentos de igual popularidad. El grado de correlación temporal puede ser cuantificado por la pendiente de la distribución que resulta (con ejes en escala logarítmica), β . Parámetro que, como se vio en el capítulo anterior, es usado por la política de reemplazo GD*.

Para las muestras de tráfico de que disponemos se ha seleccionado para el cálculo de β los documentos que se encuentran en la posición 8 del ranking de popularidad, aunque se han obtenido valores de β similares para otros valores de popularidad. El cálculo de la pendiente, y por tanto de β , se realiza mediante la obtención de la recta de regresión, obteniéndose un resultado de $\beta=0.46$.

Se puede concluir que la localidad temporal puede ser caracterizada usando el par (α, β) , donde α es el parámetro de la ley que caracteriza la popularidad de los documentos y β es el parámetro de la ley que caracteriza la correlación temporal de las peticiones de documentos de igual popularidad.

3.4. RELACIÓN ENTRE LA FRECUENCIA Y LA DISTANCIA ENTRE PETICIONES CON EL TAMAÑO DE LOS DOCUMENTOS

Las figura 3.5 muestra la relación que existe entre el tamaño del documento y la probabilidad que existe de pedirlo en más de una ocasión. Se ha visto conveniente diferenciar entre los tres tipos de documentos que se piden más frecuentemente (Imagen, Texto y Aplicación), por si se puede extraer alguna conclusión que dependa del tipo. Cada una de las columnas de la figura representa el porcentaje sobre el total de peticiones de documentos del tamaño y el tipo correspondiente que hacen referencia a documentos que se piden más de una vez.

Se puede comprobar que, aunque hay ciertas oscilaciones, el número de peticiones que corresponden a documentos que se piden en más de una ocasión decrece a medida que el tamaño del documento crece. Así, para los tres tipos estudiados, más del 90% de las peticiones de documentos de menos de 100 Bytes corresponden a documentos referenciados con anterioridad.

El tipo Texto es el que presenta más claramente una monotonía decreciente del número de peticiones múltiples (denominamos así a las peticiones correspondientes a documentos pedidos en más de una ocasión) con respecto al tamaño de los documentos hasta los dos últimos intervalos (documentos de más de 100 KBytes), para los que experimenta un ligero crecimiento.

Por otro lado, el tipo Aplicación es el que experimenta unas oscilaciones más pronunciadas del número de peticiones múltiples, llegando, en el intervalo de 1 KByte a 5 KBytes, a representar el 75% del total de las peticiones de este tipo.

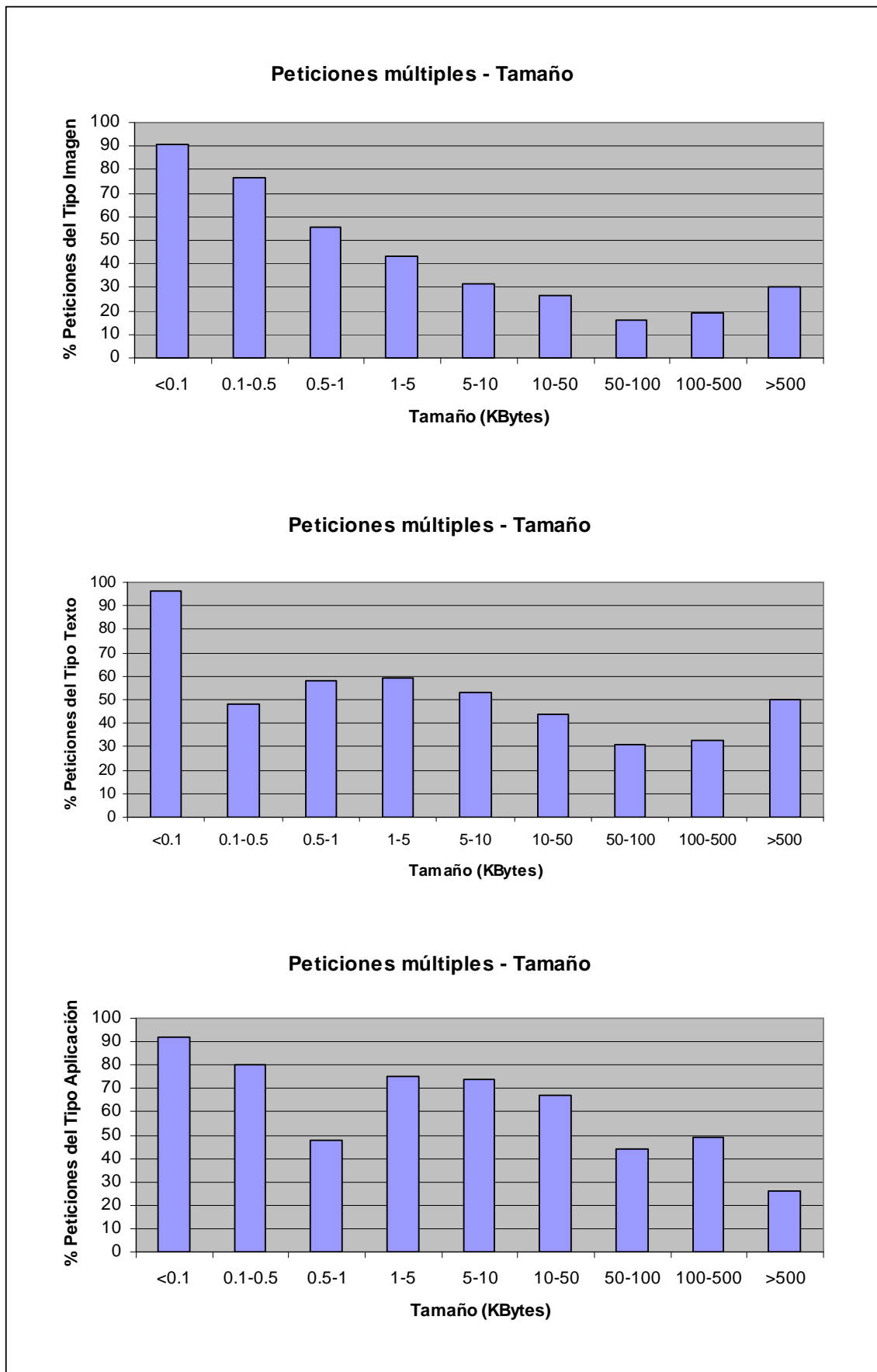


Figura 3.5. Relación entre el tamaño del documento y el número de peticiones múltiples

A continuación se trata de determinar si existe alguna relación entre el tamaño de los documentos y la distancia entre peticiones.

En la figura 3.6 se muestran gráficas, para cada uno de los tres tipos de documentos mencionados, en las que se puede observar la distribución sobre el total de peticiones correspondientes a documentos pedidos en más de una ocasión, de peticiones que se han realizado dentro del margen correspondiente de distancia medida en volumen de datos.

Se pueden apreciar diferencias entre los distintos tipos de documentos. Así, en el caso de documentos de tipo Imagen, para la mayoría de rangos de tamaño, la mayor parte de las peticiones se producen entre 1 GB y 10 GB después de que se produjera la anterior. Sin embargo, para documentos relativamente pequeños, menores de 500 Bytes, la distancia entre peticiones para la mayor parte de documentos que se piden más de una vez se reduce al intervalo entre 1 MB y 10 MB. Ocurre lo mismo para los documentos más grandes, mayores de 500 KBytes.

Para el tipo Texto se puede observar que la mayoría de peticiones (un 55% de media) de documentos referenciados más de una vez ocurren con una distancia entre ellas que va de de 1 MByte a 100 MBytes para todos los rangos de tamaño. Por tanto, para este tipo de documentos no se puede extraer una relación entre la distancia entre peticiones y el tamaño de los documentos.

Se puede observar que el tipo Aplicación presenta una distribución similar a la del tipo Texto hasta tamaños de documentos de 50 KBytes, a partir de entonces la distancia entre peticiones va aumentando. Para documentos de más de 500 KBytes la mayoría de peticiones se producen después de los 100 MBytes de haber ocurrido la anterior.

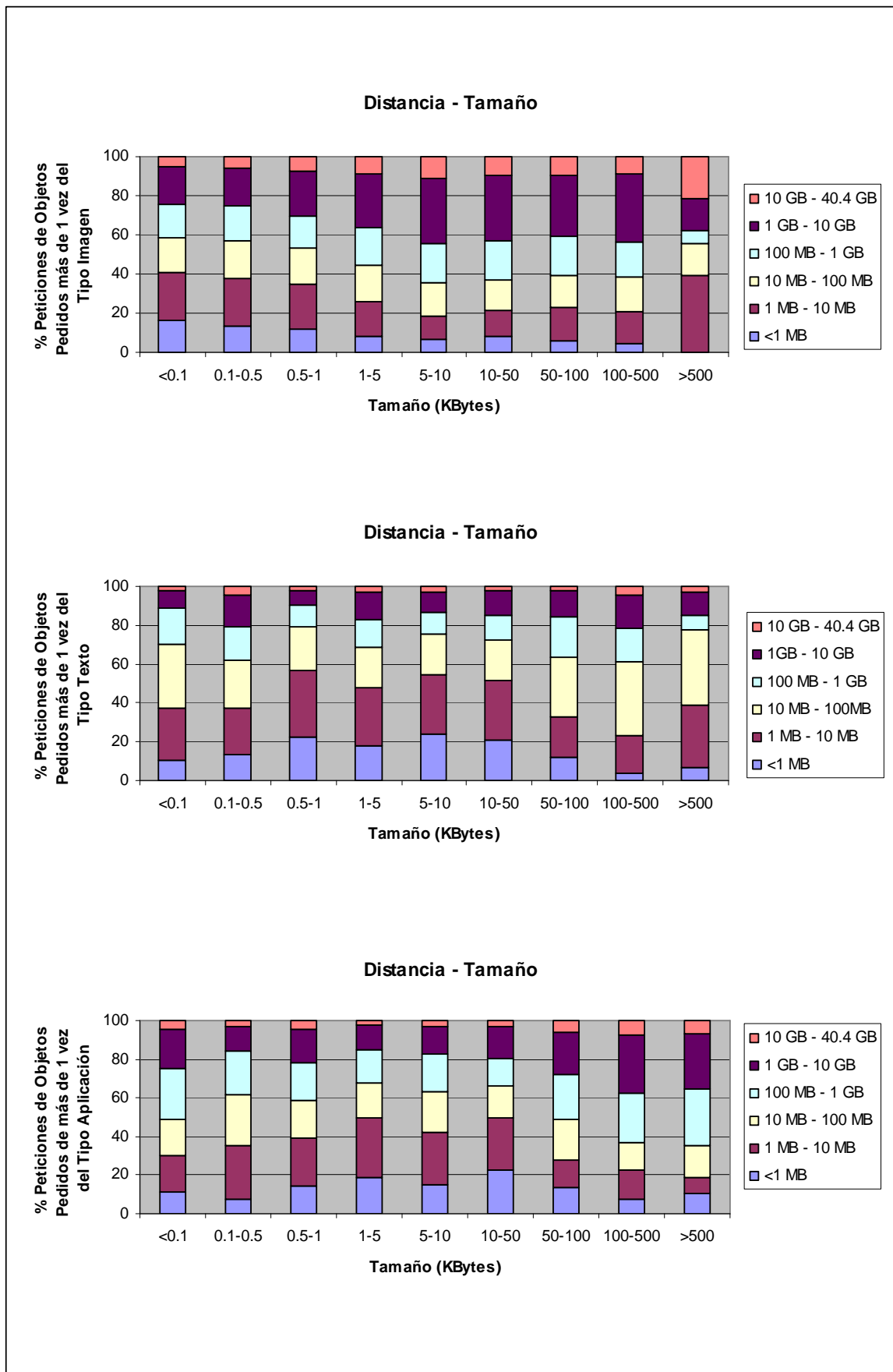


Figura 3.6. Relación entre el tamaño del documento y la distancia (en datos) entre peticiones

CAPÍTULO 4: Rendimiento de la *caché* según la política de acceso elegida

En este capítulo se estudia cómo influye en el rendimiento de la *caché* la elección de los documentos que pueden ser almacenados. Para ello se estudia el comportamiento de tres políticas de acceso diferentes. En primer lugar se usa una política de acceso basada en el establecimiento de un umbral. Para cada documento solicitado se calcula un factor, si éste supera el umbral, el documento es almacenado en *caché*, en caso contrario se rechaza. A continuación se opta por almacenar los documentos de los subtipos más solicitados de cada uno de los tipos. Por último se combinan las políticas de acceso anteriores, es decir, se determina si los documentos de los subtipos más habituales superan cierto umbral. Para evaluar el rendimiento de la *caché* y de la política de acceso se usan las nuevas métricas descritas en el capítulo 2.

4.1. POLÍTICA DE ACCESO BASADA EN EL ESTABLECIMIENTO DE UN UMBRAL

4.1.1. CÁLCULO DEL FACTOR ASOCIADO A CADA DOCUMENTO

Como se puso de manifiesto en el capítulo anterior, los documentos más solicitados son aquellos de menor tamaño. Por otro lado, algunos tipos de documentos son más populares que otros. Por ejemplo, la mayor parte de las peticiones corresponden a documentos de tipo Imagen.

Para calcular un factor asociado a cada documento se han tenido en cuenta los dos aspectos anteriores, el tamaño y el tipo. Así, se dará prioridad para entrar en *caché* a los documentos pequeños y pertenecientes a los tipos más solicitados. La ecuación 4.1 es la

que se propone para calcular dicho factor para cada documento d . En ella, $T(d)$ es un parámetro que depende del tipo de documento, siendo su valor el que se muestra en la tabla 4.1. $T(d)$ tiene como objeto proporcionar un valor más alto de F a los documentos que pertenecen a los tipos más solicitados. Así, se ha optado por dar un valor de $T=1$ a los documentos del tipo Imagen, el tipo más solicitado, e ir decrementando una décima esta cantidad para cada uno de los demás tipos. $TAM(d)$ es el tamaño del documento d medido en Bytes. Se utiliza el logaritmo del tamaño, en lugar del tamaño, para que el valor resultante no sea excesivamente bajo, quedando los valores mejor distribuidos en el intervalo. El valor resultante queda en el intervalo $[0,1]$, ocupando posiciones en la parte baja los documentos más grandes y de tipos poco solicitados, y en la parte alta, los documentos más pequeños y de tipos muy solicitados.

$$F(d) = T(d) \cdot \frac{1}{\log_{10}[TAM(d)] + 1} \quad (4.1)$$

	T
Imagen	1
Texto	0.9
Aplicación	0.8
Audio	0.7
Video	0.6

Tabla 4.1. Valores del parámetro T

4.1.2. RESULTADOS

Para estudiar el rendimiento de la *caché* con la política de acceso presentada se obtienen las métricas mediante el uso del emulador desarrollado. Para ello se señala la casilla correspondiente a Acceso Umbral y se introduce el umbral elegido. El uso de una política de acceso en el emulador es compatible con la adquisición del resto de parámetros a través de fichero XML.

Las simulaciones se han realizado para tamaños de *caché* del 2%, 5%, 10%, 30%, 50% y 70% de la *caché* infinita para el total de documentos, 22.3 GBytes. Como en el emulador implementado es necesario indicar el porcentaje de la memoria total dedicada a cada tipo de documento, se han asignado porcentajes proporcionales al peso que tiene cada tipo de documento en el volumen total de datos solicitados. Así, para el tipo Imagen y Aplicaciones se reserva un 35% de la memoria total para cada uno de ellos, y un 20%, un 5% y un 5% para los tipos Texto, Audio e Imagen, respectivamente. Como política de reemplazo se ha elegido LRU.

Se han elegido los siguientes umbrales de admisión: $U=0$, $U=0.04$, $U=0.06$, $U=0.08$ y $U=0.1$. Cuando se solicita un documento d se evalúa su factor asociado $F(d)$ y si éste supera el umbral establecido se almacena en la *caché*, en otro caso se rechaza. Para $U=0$ no se está aplicando ninguna política de acceso ya que todos los factores asociados a las peticiones de los documentos superarán dicho umbral. A medida que el umbral crece se comienza a rechazar documentos. En la tabla 4.2 se muestra el número de peticiones rechazadas para cada uno de los umbrales elegidos.

Política de Acceso	Peticiones Rechazadas
$U=0.04$	8.250
$U=0.06$	19.309
$U=0.08$	209.604
$U=0.1$	1.026.647

Tabla 4.2. Peticiones rechazadas para las políticas de acceso de tipo umbral

En la figura 4.1 se ha representado el HR y el NUHR para el caso de $U=0$ (ninguna política de acceso) y el resto de umbrales mencionados. Se puede observar que tanto el HR como el NUHR son peores cuando se aplica una política de acceso salvo para $U=0.06$, en este caso estas métricas son ligeramente más altas. Ambas métricas arrojan resultados similares salvo para el umbral más restrictivo, para $U=0.1$. Para este umbral el número de documentos rechazados es el mayor y el número de documentos rechazados correctamente se incrementa de tal manera que influye decisivamente en el aumento del valor del NUHR. La diferencia entre estas dos métricas para este umbral se aprecia más claramente en la figura 4.2.

En la figura 4.3 se muestra el BHR y NUBHR para los distintos umbrales de acceso. Se observa como el BHR cae significativamente cuando se empieza a rechazar documentos. Esto ocurre porque se está dando preferencia a los documentos pequeños frente a los de mayor tamaño a la hora de acceder a la *caché*, decrementándose la cantidad de Bytes acertados frente a los pedidos. El NUBHR, diseñado específicamente para medir el rendimiento de una *caché* con control de acceso, proporciona unos valores ligeramente más altos, y para $U=0.04$ se obtiene un valor algo más alto que cuando no se usa política de acceso.

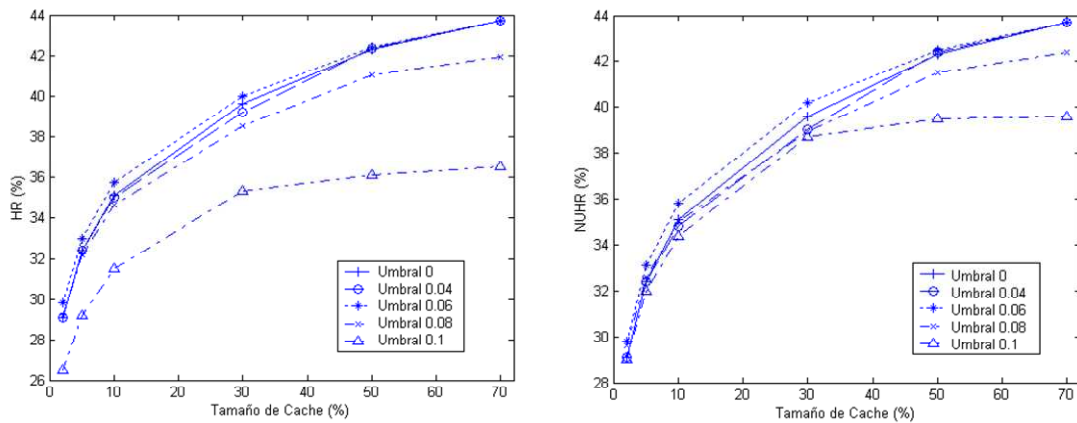


Figura 4.1. HR y NUHR para distintos umbrales de acceso

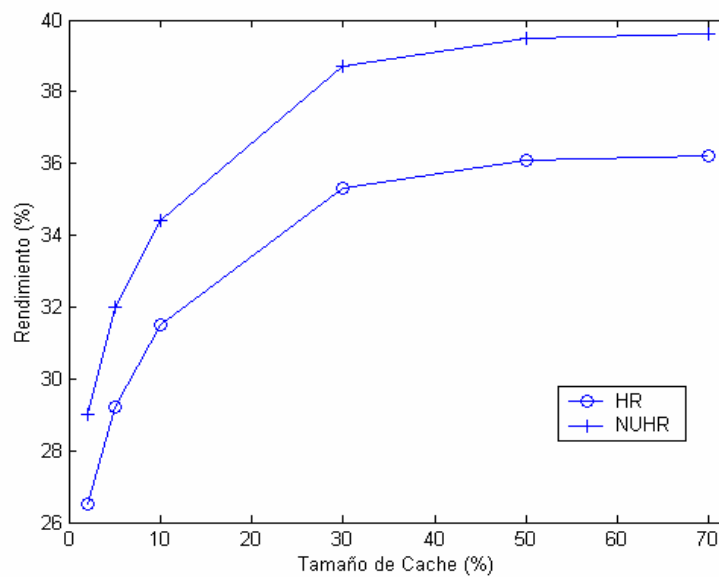


Figura 4.2. HR y NUHR para $U=0.1$

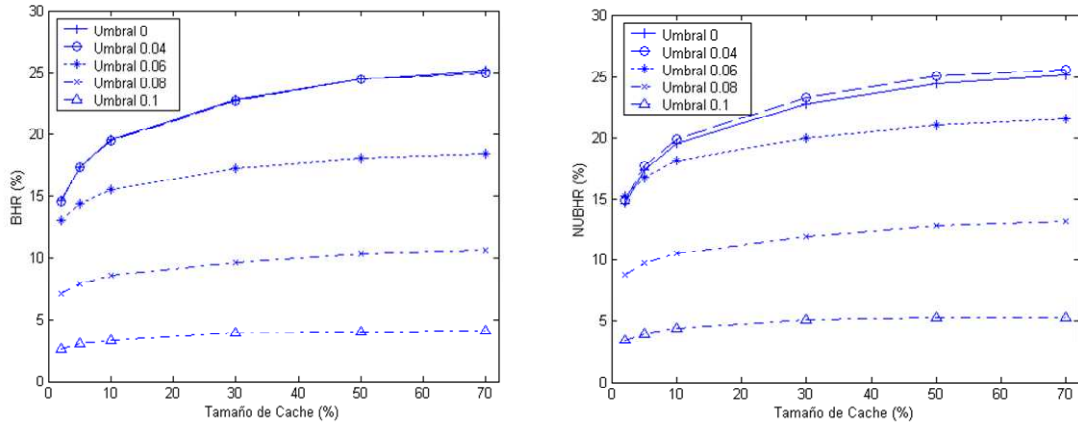


Figura 4.3. BHR y NUBHR para distintos umbrales de acceso

La figura 4.4 muestra el ACHR y la 4.5, el ACBHR, las métricas que evalúan el rendimiento de la política de acceso implementada. Se observa que la mejor opción es establecer el umbral en $U=0.1$ para el caso del ACHR, para $U=0.08$ y $U=0.06$ el resultado es prácticamente similar. Los valores más bajos se obtienen para 0.04, la política que menos documentos rechaza. En cuanto al ACBHR, se puede decir que son los umbrales más bajos los que proporcionan mejores resultados, excepto $U=0.1$, que es algo mejor que $U=0.08$.

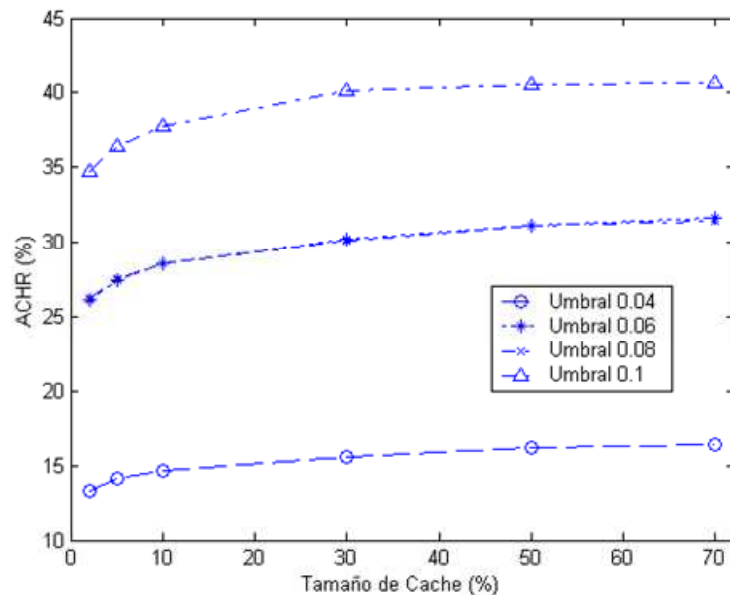


Figura 4.4. ACHR para distintos umbrales de acceso

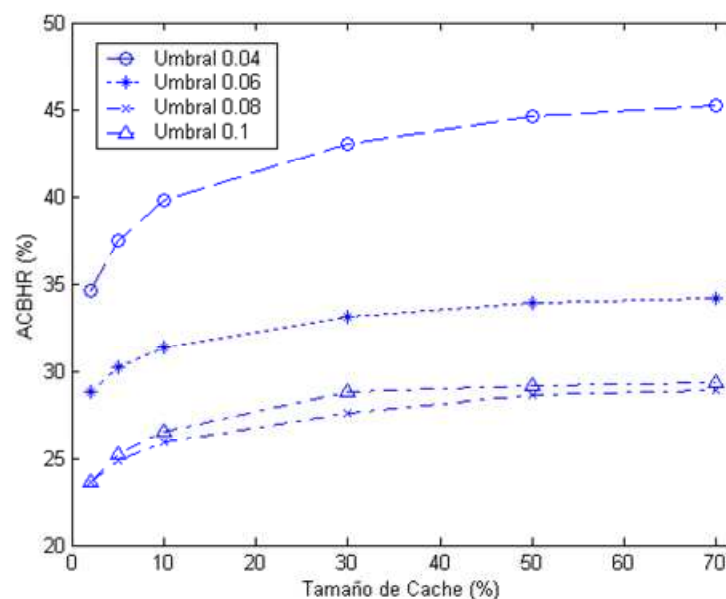


Figura 4.5. ACBHR para distintos umbrales de acceso

4.2. POLÍTICA DE ACCESO BASADA EN PERMITIR EL ACCESO DE LOS SUBTIPOS MÁS SOLICITADOS

A continuación se estudia cómo afecta al rendimiento de la *caché* permitir el almacenamiento en *caché* de los documentos cuyo subtipo pertenezca a aquellos más solicitados, rechazando el resto de documentos.

Se pretende establecer cuántos subtipos conviene aceptar para maximizar las métricas, para ello se ha estudiado el rendimiento con tres soluciones distintas: aceptar sólo el subtipo más solicitado para cada tipo, los dos, o los tres subtipos más solicitados, comparándolas con el caso de no rechazar ningún documento. En la tabla 4.3 se muestran los tres subtipos más referenciados para cada tipo y el porcentaje asociado con respecto al total de referencias del tipo. La tabla 4.4 muestra el número de peticiones rechazadas para cada política de acceso.

Tipo	Subtipo 1	%	Subtipo 2	%	Subtipo 3	%	Total %
Imagen	gif	70.38	jpg : jpeg : pjpeg	29.05	png : x-png	0.41	99.84
Texto	html	56.68	plain	22.05	css	17.43	96.16
Aplicación	Javascript : x-Javascript	73.11	octet stream : octet-stream	9.94	x-flash : x-shockwave	7.76	90.81
Audio	basic	60.32	wav : x-wav	15.45	mpeg : x-mpeg	10.03	85.8
Video	x-ms-asf	59.59	mpeg : x-mpeg	30.05	Wmv : x-ms-wmv	3.38	93.02

Tabla 4.3. Subtipos más solicitados para cada tipo de documento

Política de Acceso	Peticiones Rechazadas
3 Subtipos	179.802
2 Subtipos	217.662
1 Subtipo	1.281.176

Tabla 4.4. Peticiones rechazadas para las políticas de acceso basadas en permitir los subtipos más solicitados

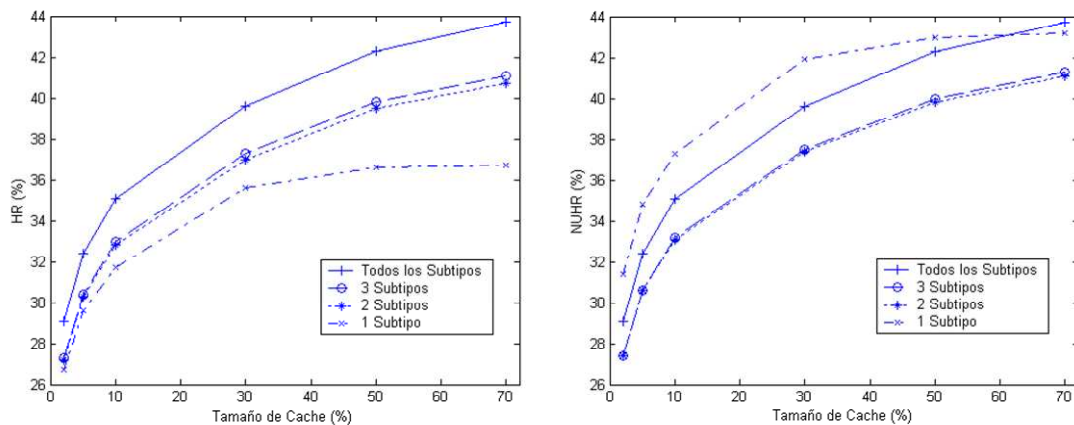


Figura 4.6. HR y NUHR para las políticas de acceso basadas en admitir los subtipos más solicitados

En la figura 4.6 se ha representado el HR y el NUHR para los casos mencionados anteriormente. En la gráfica del HR se puede observar que a medida que se rechazan más documentos el resultado que arroja la métrica va descendiendo. Siendo el caso peor el de admitir sólo el subtipo más solicitado de cada subtipo. Sin embargo en la gráfica

donde se representa el NUHR, se observa que es precisamente esta última política la que presenta un mejor resultado. Se aprecia más claramente este aspecto en la figura 4.7. Cuando sólo se admite el subtipo más solicitado, el número de documentos rechazados correctamente es mayor que en el resto de los casos. Estos documentos correctamente rechazados no producen un decremento en el número de aciertos y por tanto el rendimiento de la *caché* es mayor, quedando reflejado en la métrica definida para el estudio más adecuado de una *caché* con control de acceso, el NUHR. Sólo para el tamaño máximo de *caché* estudiado (70% de la *caché* infinita), permitir el almacenamiento de todos los documentos proporciona un NUHR mayor que almacenar sólo el subtipo más solicitado.

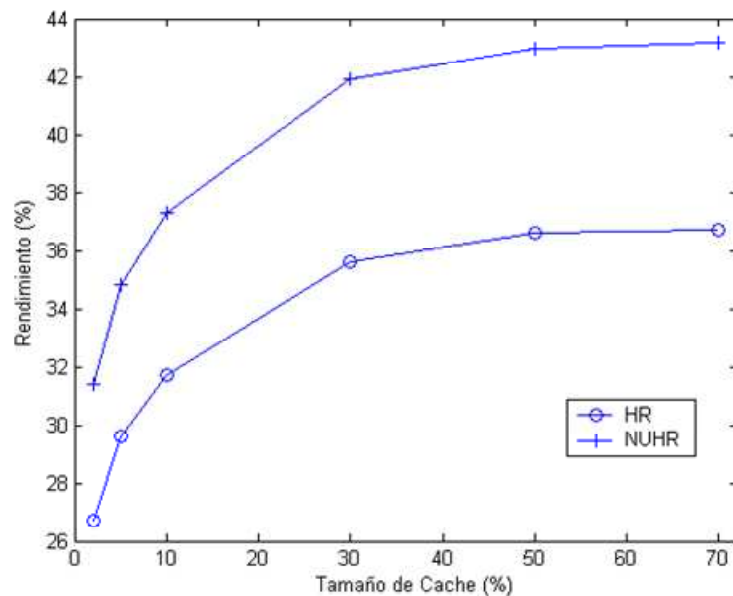


Figura 4.7. HR y NUHR para admisión del subtipo más solicitado de cada tipo de documento

En la figura 4.8 se muestra el BHR y NUBHR para las distintas políticas de acceso basadas en la admisión de los subtipos más solicitados. Se observa que tanto el BHR como el NUBHR caen significativamente cuando se aceptan sólo los documentos cuyo subtipo pertenece al más solicitado de cada tipo. Hay que tener en cuenta que se están admitiendo los subtipos más solicitados, no los subtipos con un mayor volumen de información asociado, por tanto es de esperar que las métricas que tienen en cuenta los aciertos se vean favorecidas frente a las que tienen en cuenta los Bytes acertados.

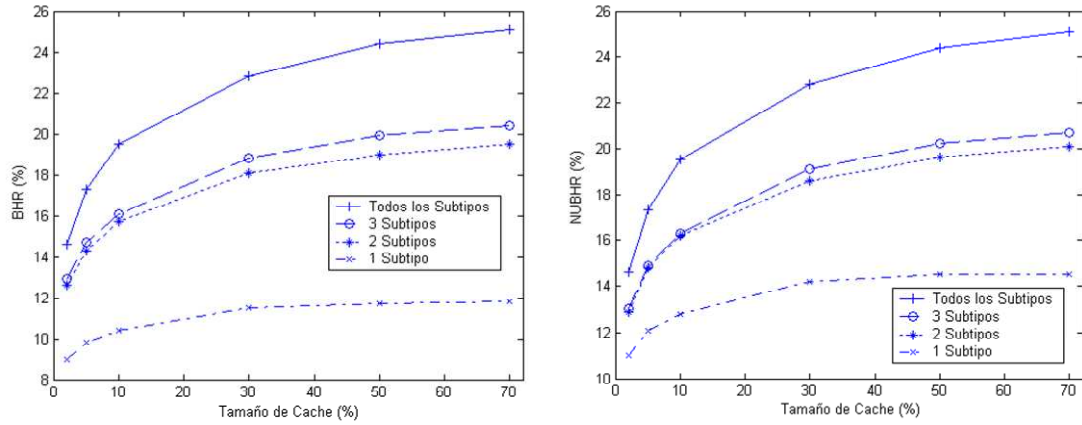


Figura 4.8. BHR y NUBHR para las políticas de acceso basadas en admitir los subtipos más solicitados

En cuanto a las métricas que evalúan el rendimiento de la política de acceso, las figuras 4.9 y 4.10 ponen de manifiesto que admitir sólo el subtipo más solicitado es la mejor opción. Es decir, si se sigue este criterio la política de acceso cumple mejor su función, maximizar la tasa de documentos correctamente rechazados por un lado, y la tasa de documentos correctamente admitidos por otro.

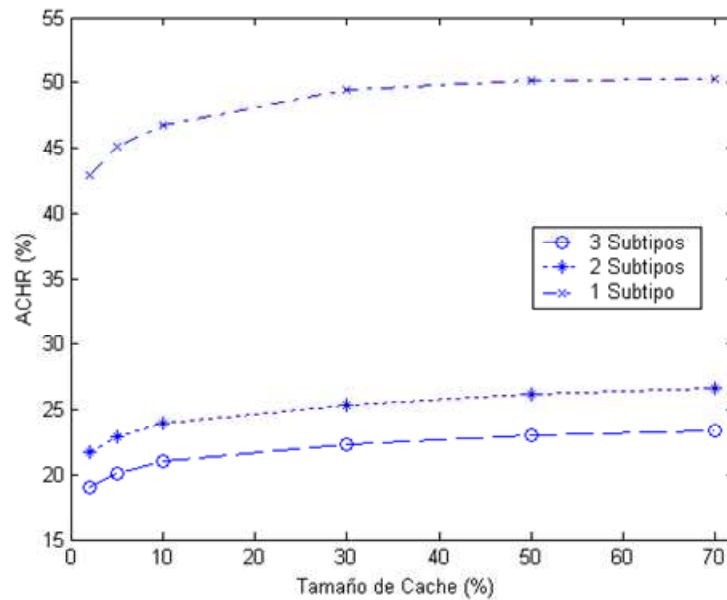


Figura 4.9. ACHR para las políticas de acceso basadas en admitir los subtipos más solicitados

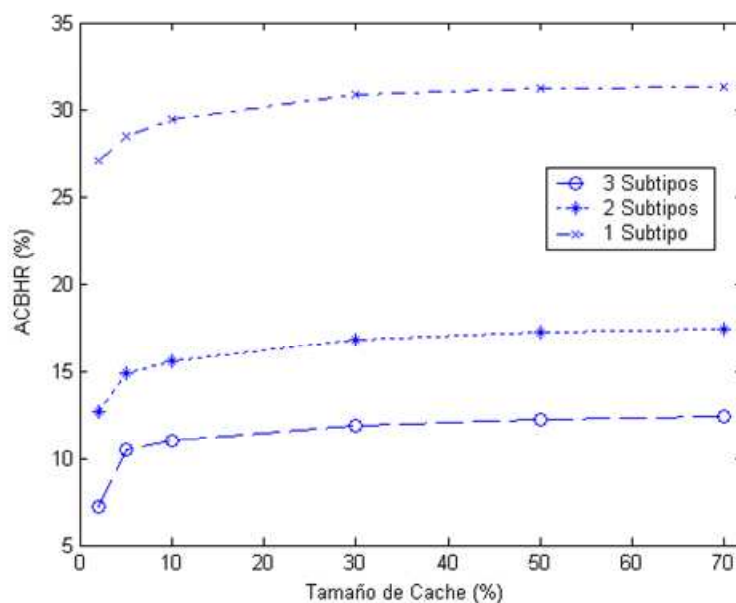


Figura 4.10. ACBHR para las políticas de acceso basadas en admitir los subtipos más solicitados

4.3. POLÍTICA DE ACCESO COMBINADA: ADMISIÓN DE DOCUMENTOS DE SUBTIPOS MÁS SOLICITADOS QUE SUPEREN UN UMBRAL

En este apartado se estudia una nueva política de acceso basada en admitir aquellos documentos que superan cierto umbral establecido y que pertenecen a los subtipos más solicitados.

Se ha considerado oportuno establecer el umbral en $U=0.06$ por ser el que proporciona un mejor comportamiento del control de acceso, en términos de ACHR, cuando se utiliza como política de acceso única, tal como se estudió en el apartado 4.1. Se compara el rendimiento de la *caché* cuando se combina este umbral con la admisión de los documentos cuyo subtipo es el primero, el segundo, o el tercero más solicitado de cada tipo. En la tabla 4.5 se señala el número de peticiones rechazadas para cada tipo de política de acceso.

Política de Acceso	Peticiones Rechazadas
3 Subtipos	186.378
2 Subtipos	222.646
1 Subtipo	1.282.978

Tabla 4.5. Peticiones rechazadas para las políticas de acceso combinadas

Los resultados son muy parecidos a los que se han obtenido en el apartado anterior, aunque se aprecia un ligero incremento en los valores de algunas de las métricas. Se puede observar en la figura 4.11 como para la política en la que se admite sólo el subtipo más solicitado se obtiene el valor más alto de NUHR salvo para la *caché* de más tamaño.

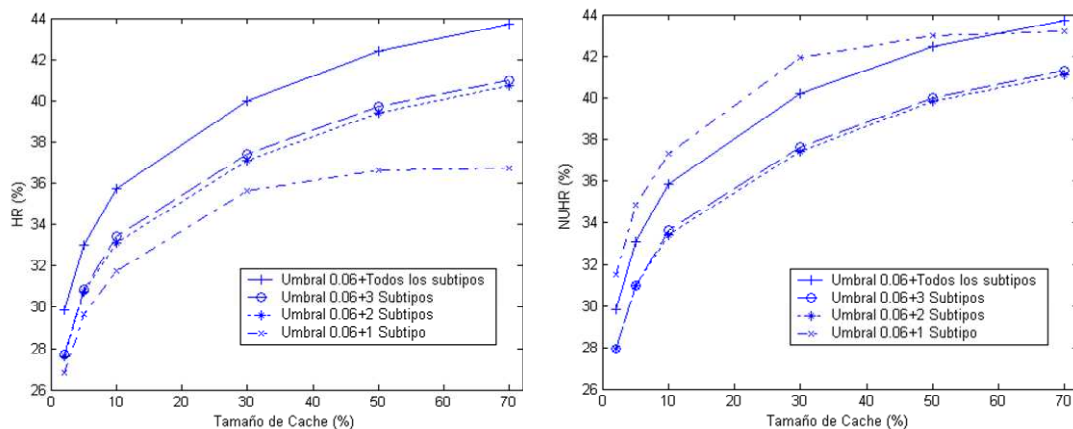


Figura 4.11. HR y NUHR para las políticas de acceso combinadas

En la figura 4.12 se comparan el BHR y el NUBHR. Como ocurría en las políticas anteriores, el NUBHR, al considerar los documentos rechazados correctamente, se incrementa con respecto al BHR.

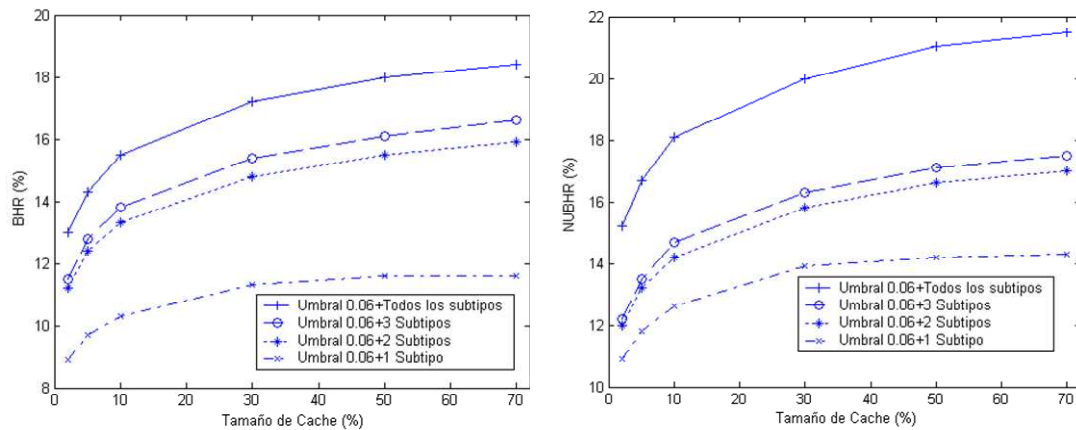


Figura 4.12. BHR y NUBHR para las políticas de acceso combinadas

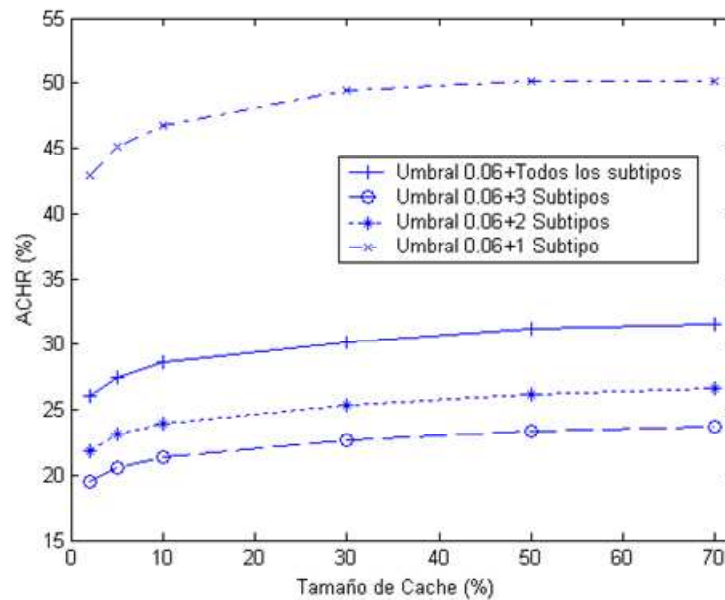


Figura 4.13. ACHR para las políticas de acceso combinadas

Las figuras 4.13 y 4.14 muestran el rendimiento de la política de acceso combinada. El mayor ACHR se da cuando se admite sólo el acceso de los documentos de subtipo correspondiente al más solicitado para cada tipo, como ocurría cuando no se consideraba ningún umbral para permitir el acceso. Sin embargo, el ACBHR mayor se obtiene cuando se usa un umbral y se permite el acceso de todos los subtipos.

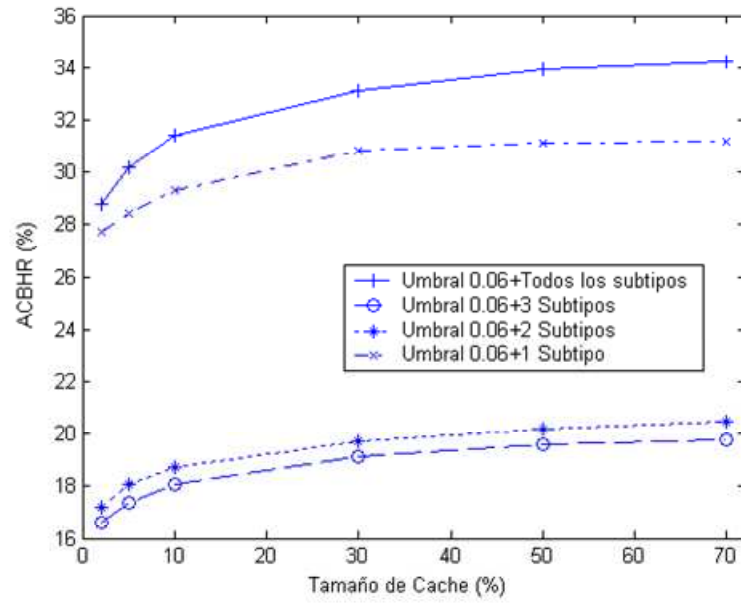


Figura 4.14. ACBHR para las políticas de acceso combinadas

CAPÍTULO 5: Conclusiones y líneas futuras

En este capítulo se presentan las conclusiones obtenidas tras la realización de este Proyecto Fin de Carrera. Para ello se comentan las implementaciones y estudios realizados para la consecución de los objetivos inicialmente planteados. Además, se proponen algunas líneas futuras de investigación.

5.1. CONCLUSIONES FINALES

El objetivo del proyecto es la evaluación del rendimiento de una *caché* Web en un servidor Proxy con control de acceso. Se proponen varias políticas de acceso y, con nuevas métricas definidas al efecto, se evalúa tanto el rendimiento de la *caché* en si misma como el de la política de acceso en particular.

El emulador de *caché* utilizado fue optimizado para poder realizar las emulaciones con un menor coste computacional y por tanto en menos tiempo. Además se le añadió distintas mejoras, entre ellas las necesarias para implementar distintas políticas de acceso, y la posibilidad de adquirir de forma automática los parámetros de entrada desde un fichero XML. Esto último es particularmente interesante a la hora de realizar sucesivas emulaciones, ya que no se requiere una supervisión directa del usuario para realizarlas.

También se añadió una nueva política de reemplazo: LFU-DA. Se comparó el rendimiento que se obtiene con esta política con el de LFU y LRU. Para tamaños de *caché* inferiores al 15% de la *caché* infinita es LFU la que muestra un mejor comportamiento tanto en el caso del HR como del BHR. A partir de ese punto, la

política de reemplazo que mostraba unos peores resultados, LRU, pasa a ser la de mejor comportamiento. LFU-DA se mantiene siempre entre las dos anteriores.

Antes de realizar el estudio de las políticas de acceso, se procedió a hacer un estudio estadístico de las características del tráfico empleado. De esta forma se obtuvieron resultados útiles para la definición de una política de acceso basada en el establecimiento de un umbral: en primer lugar el hecho de que los documentos de menor tamaño son los más solicitados, y por otro lado, que algunos tipos tienen mucho más peso que otros en el conjunto de las solicitudes efectuadas. Estos dos aspectos se utilizan para asignar un valor a cada solicitud, si este valor supera el umbral, el documento es almacenado.

Además de la política de acceso mencionada, se implementaron otras dos. Una de ellas basada en la admisión de los subtipos más solicitados de cada tipo, y la otra, basada en una combinación de las dos anteriores, es decir, se admiten los documentos de los subtipos más habituales cuya valoración supere cierto umbral.

Para la evaluación del rendimiento de la *caché* y de las políticas de acceso se usaron nuevas métricas: NUHR, NUBHR, ACHR y ACBHR, para las que se afinó el cálculo de las peticiones correspondientes a documentos rechazados correctamente.

Una vez realizadas las emulaciones con las políticas de acceso propuestas y una misma política de reemplazo, LRU, para todas ellas, las conclusiones que se extraen son las siguientes:

1. Las nuevas métricas definidas, NUHR, NUBHR, ACHR y ACBHR, evalúan de una forma más realista el rendimiento de la *caché* cuando existe una política de acceso, y de la propia política de acceso, ya que tienen en cuenta los documentos correctamente rechazados.
2. Para la política de acceso basada en el establecimiento de un umbral, de los umbrales estudiados, $U=0.06$ es el que proporciona el mejor rendimiento de la *caché* en lo que se refiere al NUHR. Para el NUBHR es el umbral $U=0.04$ el que ofrece los mejores resultados. Cuando se rechaza un mayor número de

documentos, $U=0.1$, el rendimiento de la política de acceso medido por el ACHR es el mejor. Sin embargo, conviene aplicar el umbral más bajo, $U=0.04$, si lo que se pretende es maximizar el ACBHR.

3. Para la política de acceso basada en la admisión de los subtipos más solicitados, rechazar los documentos que no pertenezcan al subtipo más solicitado de cada tipo es la mejor opción para obtener el NUHR más alto, salvo para tamaños grandes de *caché* (del orden del 70% de la *caché* infinita), en este caso conviene admitir todos los subtipos. Por el contrario, admitir sólo el subtipo más solicitado proporciona el NUBHR más bajo. En cuanto al rendimiento de la política de acceso, el mejor comportamiento se obtiene con la admisión de sólo el subtipo más solicitado, tanto en términos de ACHR como de ACBHR.
4. Cuando se usa una política de acceso combinada los resultados obtenidos son similares a los que se obtienen con la política de acceso basada en la admisión de subtipos habituales. En este caso no es determinante la aplicación del umbral en los valores de las métricas empleadas ya que los documentos que superan el umbral son pocos en comparación con los que son almacenados por pertenecer a los subtipos más habituales.

5.2. LÍNEAS FUTURAS

A continuación se describen algunas líneas de desarrollo en las que se puede seguir trabajando:

1. Inclusión en el emulador de otras políticas de reemplazo como *Lowest Latency First* y *Hybrid* [Wooster'97] encaminadas a minimizar la latencia media, para ello habría que hacer uso de nuevas muestras de tráfico con más información.

2. Se podría implementar una partición adaptativa de la *caché*, en la que el tamaño asignado a cada tipo de documento se modifique dinámicamente durante el progreso de la emulación en función de las peticiones efectuadas.
3. En cuanto a las políticas de acceso, sería interesante, sobre todo para las políticas combinadas que se han estudiado, realizar emulaciones con otros umbrales de admisión. Es posible que se obtenga mejores resultados para umbrales que rechacen un mayor número de documentos.
4. También se podrían añadir otras políticas de acceso al emulador. Se propone por ejemplo la definida por Aggarwal [Aggarwal'99], que hace uso de una pequeña *caché* auxiliar donde se almacena información sobre las peticiones realizadas.

Bibliografía

- [Abrams'95] M. Abrams, C.R. Standridge, G. Abdulla, S. Williams y E. A. Fox. "Caching Proxies: Limitations and Potentials", Julio 1995.
- [Aggarwal'99] C. Aggarwal, J. L. Wolf, y P. S. Yu, "Caching on the World Wide Web", *IEEE Transactions on knowledge and data Engineering*, Vol. 11, Núm. 1, Enero/Febrero 1999.
- [AltovaXMLSpy] <http://www.altova.com/xmlspy/>.
- [Arlitt'99] M. Arlitt, R. Friedrich, y T. Jin, "Performance Evaluation of Web Proxy Cache Replacement Policies". *En Performance Evaluation Journal*, 1999.
- [Aumaille'00] B. Aumaille. "Java 2". Colección Mega+, manuals prácticos. Editado por Ediciones ENI. Enero 2000.

- [Barford'99] P. Barford, A. Bestavros, A. Bradley y M. Crovella. "Changes in Web Client Access Patterns: Characteristics and Caching Implications", 1999.
- [Bestavros'99] S. Jin, A. Bestavros, "Temporal Locality in Web Request Streams. Sources, Characteristics, and Caching Implications", Octubre 1999.
- [BorlandJBuilder] <http://www.borland.com/de/products/jbuilder/>.
- [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'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.
- [Dilley'99] John Dilley, Ludmila Cherkasova, y Martin Arlitt. "Evaluating Content Management Techniques for Web Proxy Caches". Hewlett Packard, Abril 1999.
- [IRCache] <http://www.ircache.net/>.
- [JProfiler] <http://www.ej-technologies.com/>.
- [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 System and Networks (DSN'02), páginas 717-726, Washington, D.C. (EEUU), Junio 2002.

- [Markatos'96] E.P. Markatos, "Main Memory Caching of Web Documents". En *Proceedings of the fifth international World Wide Web conference on Computer networks and ISDN systems*. París, 1996.
- [NLANR] National Laboratory for Applied Network Research, <http://www.nlanr.net/>.
- [Wooster'97] R.P. Wooster y M.D. Abrams. "Proxy Caching that Estimates Page Load Delays". *Computer Networks*, 29(8-13): Páginas 977-986, Septiembre 1997.
- [Wessels'01] Duane Wessels. "Web Caching, reducing network traffic". Publicado por O'Reilly. Junio 2001.
- [Young'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, páginas 241-250, 1991.
- [Zukowski'03] J. Zukowski. "Programación en Java 2, J2SE 1.4". Ediciones Anaya Multimedia, Febrero 2003.

Apéndice A. Codificación de los subtipos de documentos

En este apéndice se enumera el conjunto de subtipos considerados para cada tipo de documento y la codificación 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-Javascr
- Java-vm : x-Java-vm
- mac-binhex40
- 15 metastream
- msword

20 octet_stream : octet-stream : x-octet-stream
 pdf
 pkix-crl
 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-incridimail
 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

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-Video

- 0 avi
- mpeg : x-mpeg
- quicktime
- unknown
- wmv : x-ms-wmv
- 5 x-ms-video : x-msvideo
- x-ms-wvx
- desconocido

Apéndice B. Diagramas UML

Se muestran en este apéndice los diagramas UML de las clases más importantes del emulador empleado, haciendo hincapié en las nuevas clases implementadas y en las modificadas con respecto a la versión anterior.

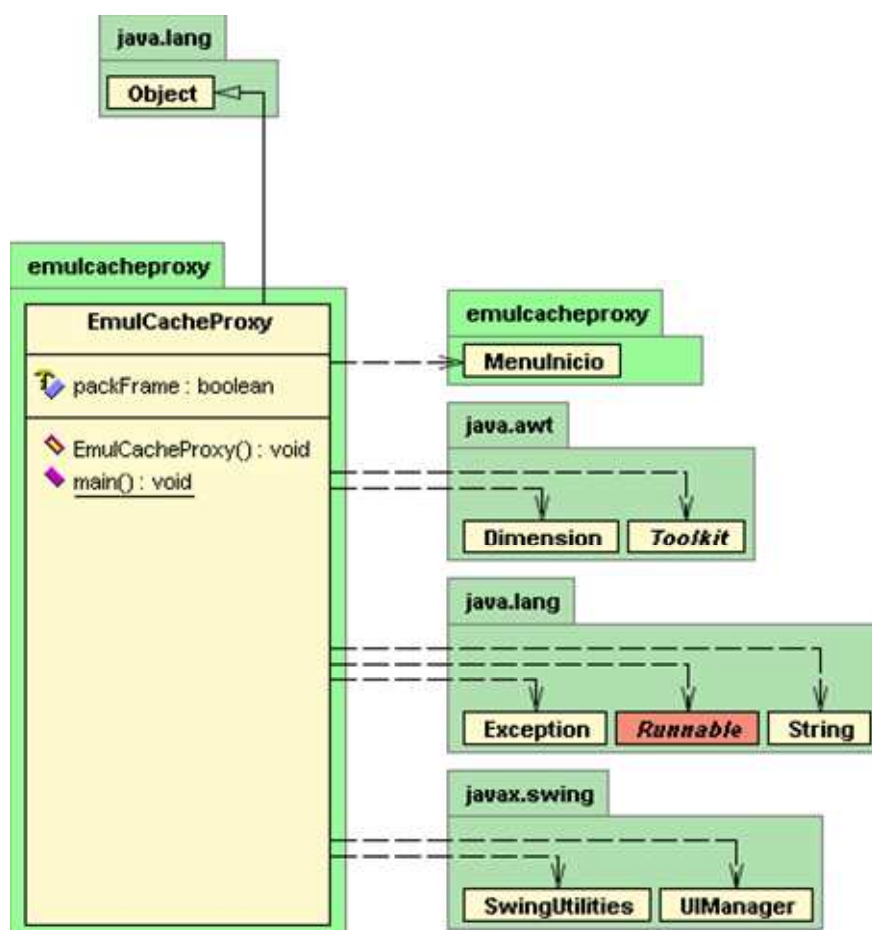


Figura A.1. Diagrama UML de *EmulCacheProxy*

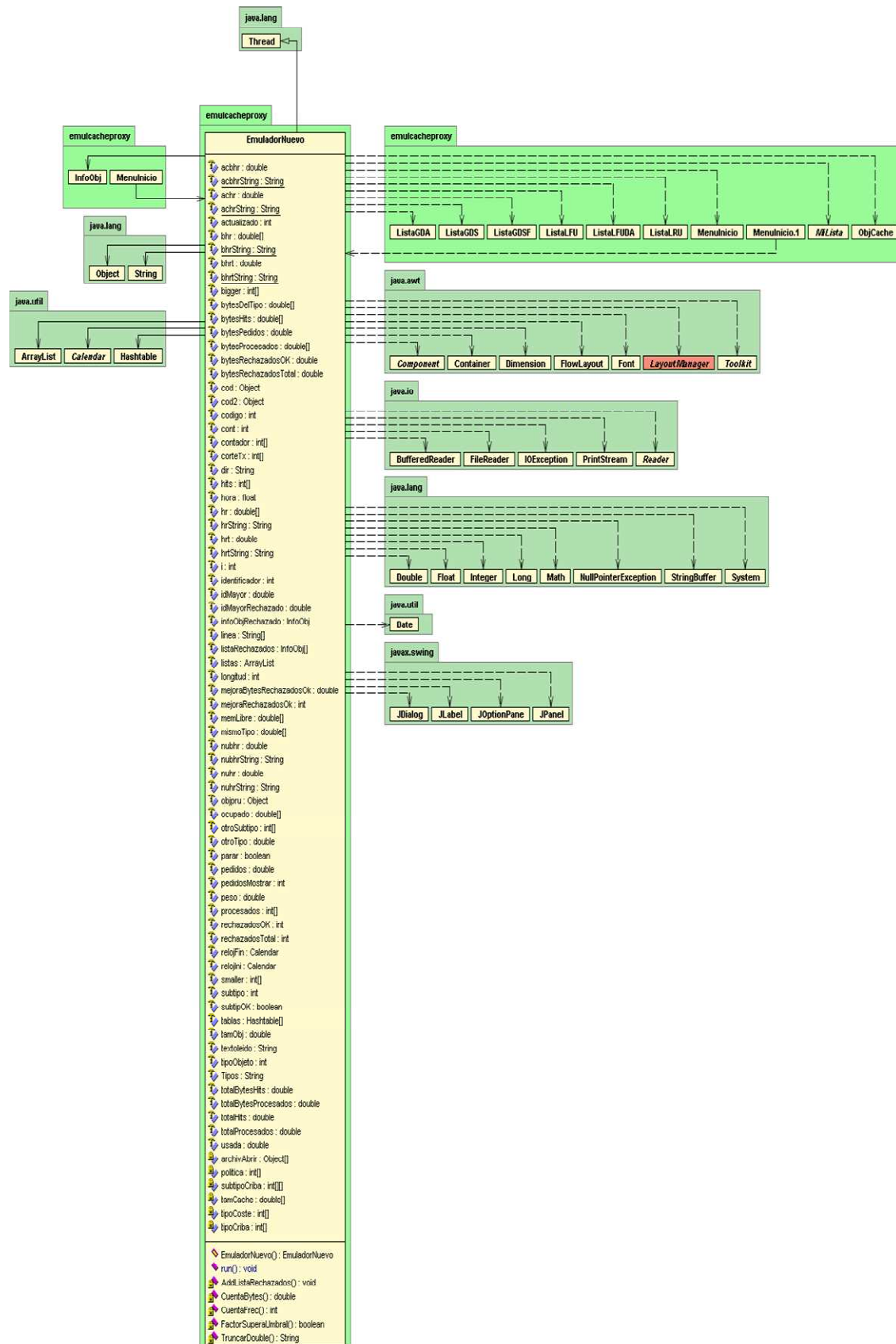


Figura A.2. Diagrama UML de *EmuladorNuevo*

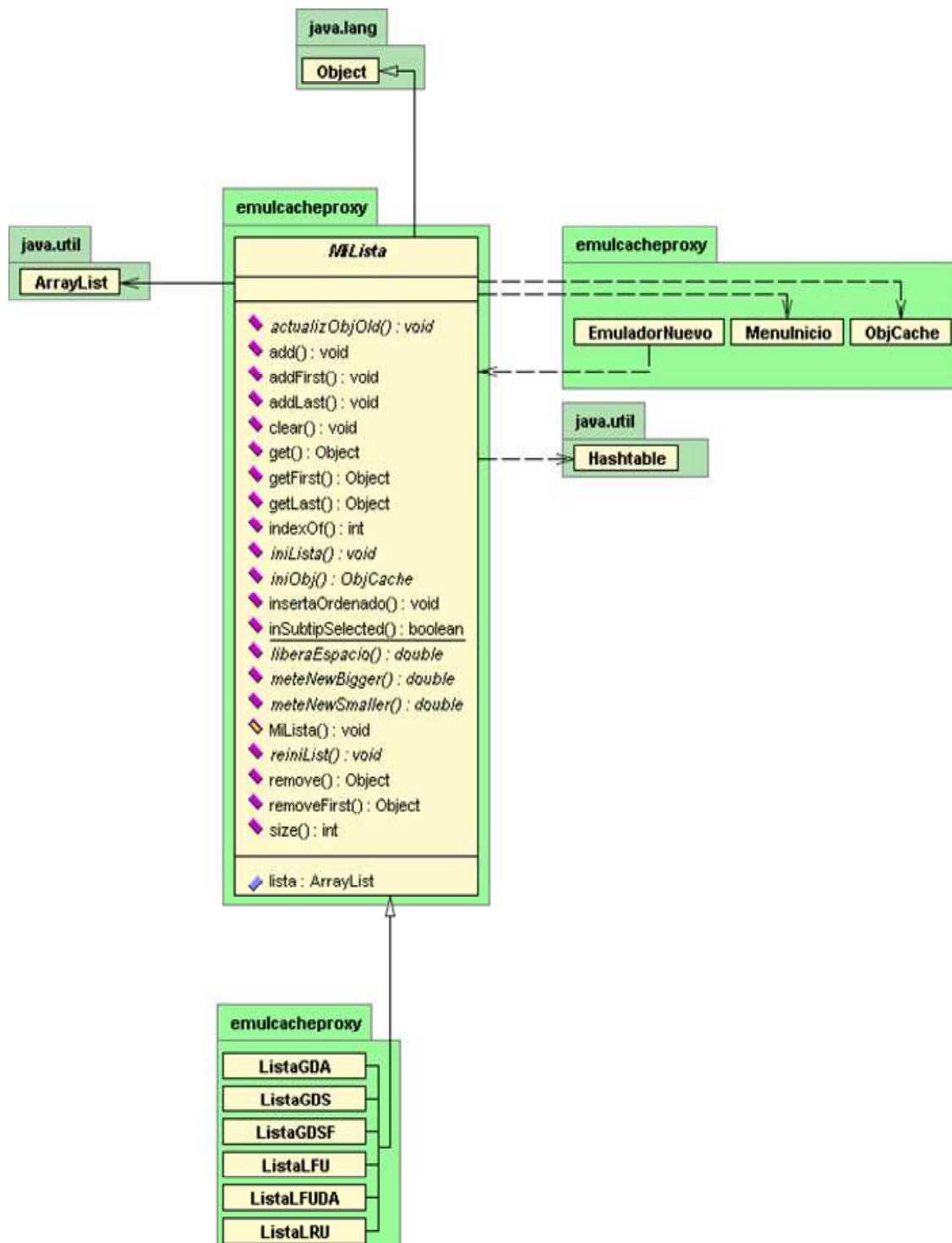


Figura A.3. Diagrama UML de *MiLista*

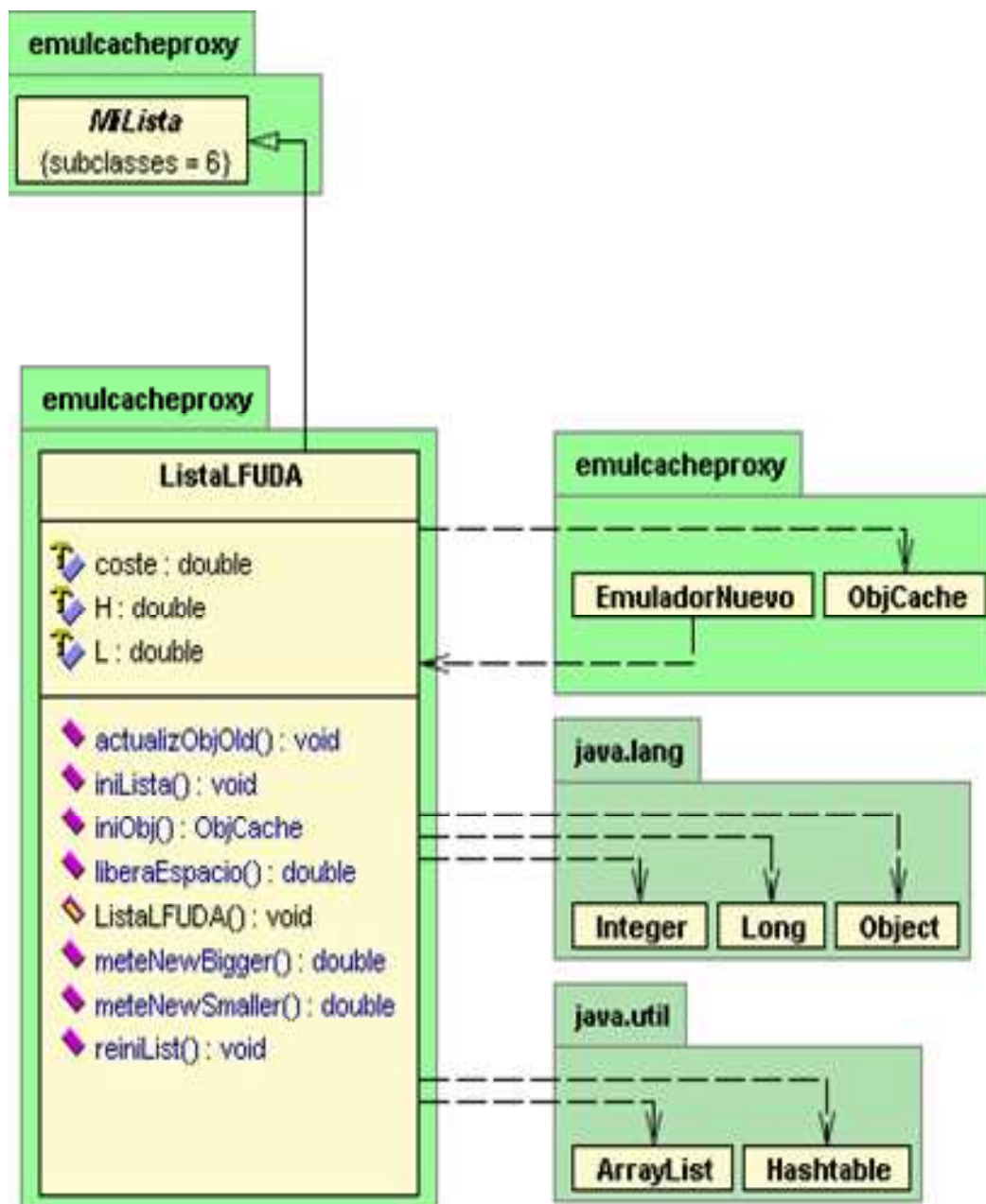


Figura A.4. Diagrama UML de *ListaLFUDA*

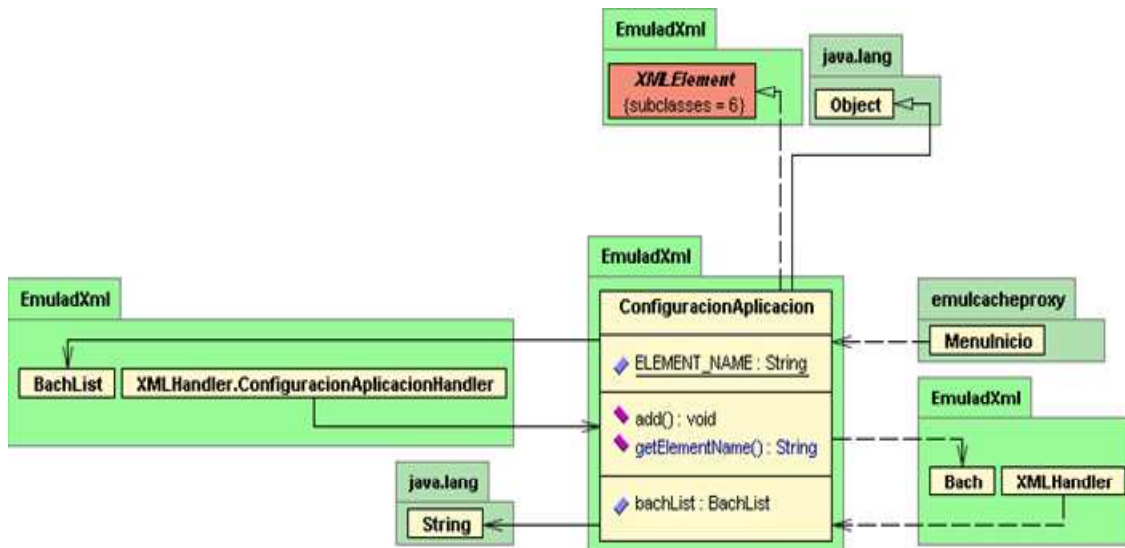


Figura A.5. Diagrama UML de *ConfiguracionAplicacion*

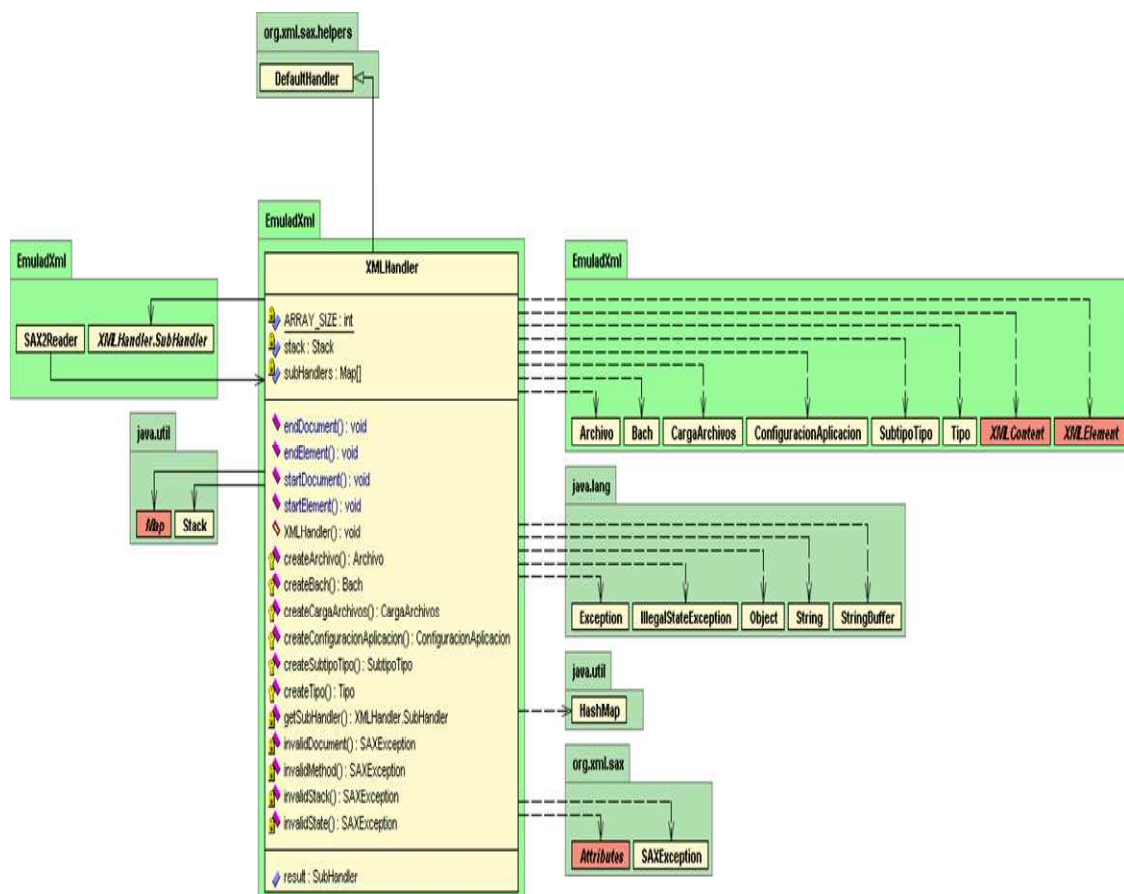


Figura A.6. Diagrama UML de *XMLHandler*

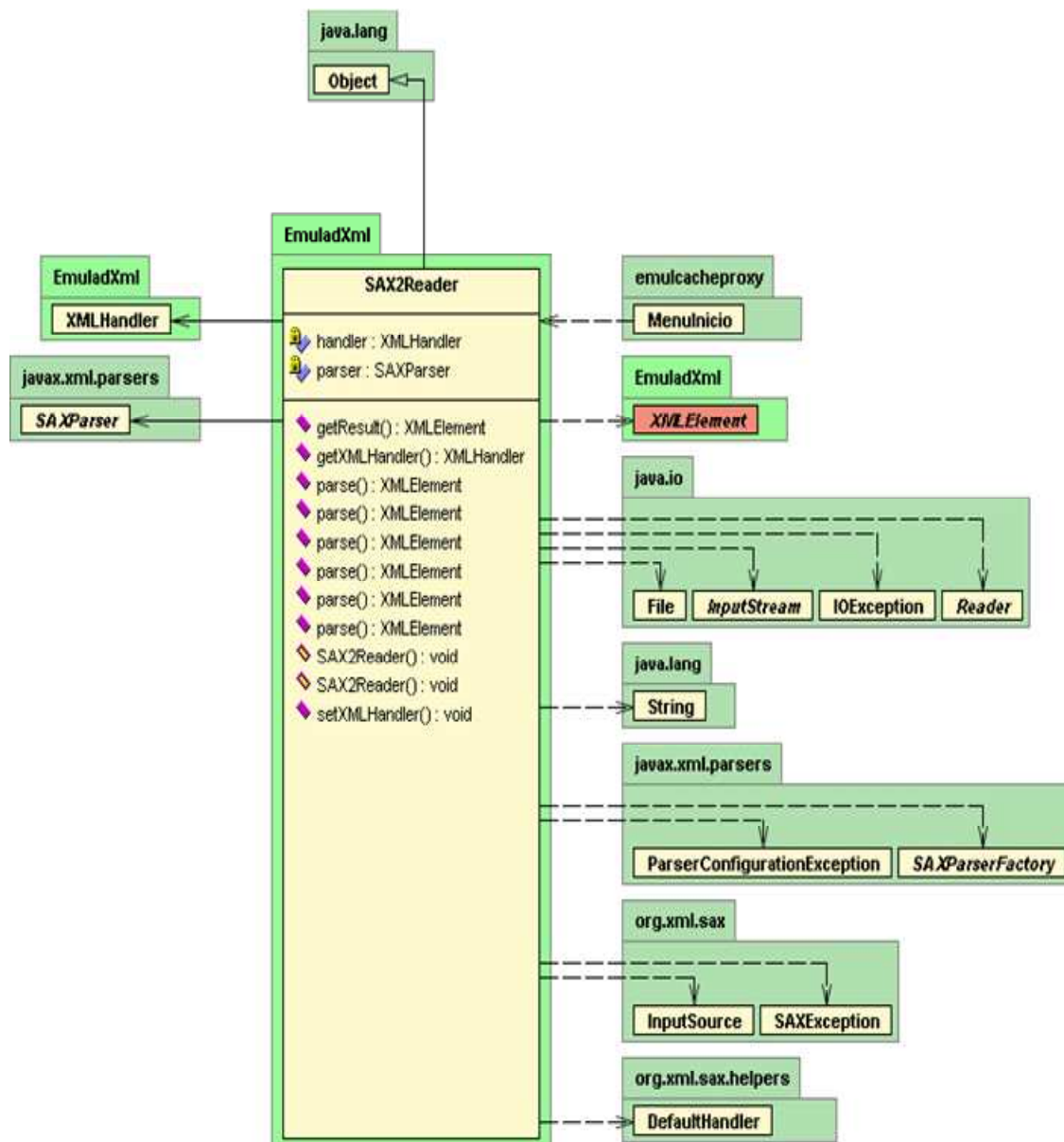


Figura A.7. Diagrama UML de *SAX2Reader*