DHTCaché - HERRAMIENTA DE APOYO A LA TOMA DE DECISIONES PARA LA CONFIGURACIÓN DE UN SISTEMA DE CACHÉ EN APLICACIONES *DHT*

CARLOS EDUARDO GÓMEZ MONTOYA

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

PROGRAMA MAGISTER EN INGENIERIA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ, D.C. 2007

DHTCaché - HERRAMIENTA DE APOYO A LA TOMA DE DECISIONES PARA LA CONFIGURACIÓN DE UN SISTEMA DE CACHÉ EN APLICACIONES *DHT*

CARLOS EDUARDO GÓMEZ MONTOYA

Tesis de grado presentada para optar al título de
Magíster en Ingeniería de Sistemas y Computación

Asesores:
Harold Enrique Castro Barrera, Ph.D.
María del Pilar Villamil Giraldo, Ph.D.

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

PROGRAMA MAGISTER EN INGENIERIA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ, D.C. 2007

NOTA DE ACEPTACIÓN

____

____

____

____

Ing. Harold Enrique Castro Barrera, Ph.D.

____

Ing. María del Pilar Villamil Giraldo, Ph.D.

____

Ing. Claudia Roncancio, Ph.D.

BOGOTÁ, D.C.

AGRADECIMIENTOS

Finalizar esta tesis no hubiera sido posible sin la colaboración de algunas personas a quienes quiero expresar mis agradecimientos.

A Harold Castro Barrera y María del Pilar Villamil Giraldo, profesores de la Universidad de Los Andes, por su asesoría durante la realización de esta tesis.

A Julián Esteban Gutiérrez Posada, compañero y amigo por sus ideas y su apoyo en la corrección de este trabajo.

A Gilberto Gómez Gómez, mi padre, por estar siempre pendiente de mí.

A la Universidad del Quindío, por brindarme la oportunidad de realizar estudios de maestría.

Y especialmente a Olga Lucía, mi esposa y a mis hijos Pablo y Martín, por su apoyo y compañía, quienes soportaron con paciencia la realización de mis estudios.

CONTENIDO

1. INTRODUCCIÓN ... 1
2. CONTEXTO ... 5
2.1 SISTEMAS *PEER-TO-PEER* ... 5
2.1.1 Aplicaciones ... 5
2.1.2 Características ... 6
2.1.3 Topologías ... 6
2.1.4 Clasificación ... 7
2.2 SISTEMAS *DHT* ... 9
2.2.1 *Distributed Lookup Service* (*DLS*) ... 10
2.2.2 *Distributed Storage Service* (*DSS*) ... 15
2.2.3 *Distributed Data Service* (*DDS*) ... 18
3. CACHÉ ... 22
3.1 CARACTERÍSTICAS ... 23
3.1.1 Gestión de caché ... 24
3.1.2 Política de reemplazo ... 24
3.1.3 Estrategia de resolución ... 25
3.2 PROTOCOLOS DE COMUNICACIÓN ENTRE CACHÉS ... 26
3.2.1 *ICP - Internet Cache Protocol* ... 26
3.2.2 *HTCP - Hyper Text Caching Protocol* ... 26
3.2.3 *CARP - Cache Array Routing Protocol* ... 26
3.2.4 *CRISP - Caching and Replication for Internet Service Performance* ... 26
3.2.5 *Cache Digest* ... 27
3.3 VENTAJAS Y DESVENTAJAS DEL CACHÉ ... 27
4. CACHÉ DISTRIBUIDO ... 29
4.1 ARQUITECTURAS DE CACHÉ DISTRIBUIDO ... 29
4.1.1 *Proxy* caché autónomo ... 29
4.1.2 *Proxy* Caché Sustituto ... 30
4.1.3 Caché jerárquico ... 30
4.1.4 Caché distribuido o en malla ... 31
4.1.5 Caché híbrido ... 31
4.1.6 Caché transparente ... 31
4.1.7 Caché cooperativo ... 32
4.1.8 Caché adaptativo ... 33
5. *DHTCaché* ... 34
5.1 ANÁLISIS DE REQUERIMIENTOS ... 34
5.1.1 Requerimientos funcionales ... 34
5.1.2 Requerimientos no funcionales ... 37
5.2 DIAGRAMA DE CASOS DE USO ... 37
5.3 ARQUITECTURA DEL SISTEMA ... 38
5.4 DIAGRAMA FUNCIONAL DEL SISTEMA ... 39
5.5 ALCANCE DE LA IMPLEMENTACIÓN ... 39

5.5.1 Otros logros......42
5.6 DIAGRAMAS DE SECUENCIA......45
6 PRUEBAS......48
6.1 PRUEBAS SOBRE EL SISTEMA *DHT*......48
6.2.1 Caso de prueba 01-01......48
6.2.2 Caso de prueba 01-02......49
6.2 PRUEBAS SOBRE EL CACHÉ *ACS*......50
6.1.1 Caso de prueba 02-01......50
6.1.2 Caso de prueba 02-02......50
6.1.3 Caso de prueba 02-03......51
6.1.4 Caso de prueba 02-04......52
6.1.5 Caso de prueba 02-05......53
6.1.6 Caso de prueba 02-06......54
6.1.7 Caso de prueba 02-07......55
6.1.8 Caso de prueba 02-08......57
6.1.9 Caso de prueba 02-09......59
6.3 PRUEBAS SOBRE EL ANALIZADOR DE ARCHIVOS *LOG*......59
6.3.1 Caso de prueba 03-01......60
6.3.2 Caso de prueba 03-02......61
6.3.3 Caso de prueba 03-03......62
7. CONCLUSIONES Y TRABAJO FUTURO......64
8. REFERENCIAS......65
ANEXO - Descripción de *ACS* - *Adaptable Cache Services*......68

1. INTRODUCCIÓN

Los sistemas *peer-to-peer* son sistemas distribuidos formados por nodos interconectados con el propósito de compartir recursos. Los nodos pueden organizarse por sí solos y adaptarse a la continua entrada y salida de otros nodos del sistema sin necesidad de la intermediación o soporte de un servidor global o una autoridad central [4].

Los sistemas *peer-to-peer* están caracterizados por la descentralización y autoorganización, porque al no depender de una autoridad central que cumpla labores de administración, el sistema debe estar en condiciones de establecer y reestablecer relaciones entre los nodos, manteniendo el sistema en operación a pesar de la alta volatilidad de los miembros del sistema; la disponibilidad y la tolerancia a fallas gracias a la redundancia en la información entre los nodos lo que hace que la falla de un nodo no bloquee el sistema; la escalabilidad, pues el desempeño del sistema no se debe afectar significativamente al incrementar el número de nodos o de objetos; la autonomía, porque cada miembro del sistema tiene la capacidad de elegir cuándo hacer parte del sistema *peer-to-peer* y cuándo realizar sus tareas individuales; y por el manejo adecuado de los recursos dada la posibilidad de compartir recursos computacionales [5].

Los sistemas *peer-to-peer* empiezan a ser conocidos a raíz de la creación de *Napster* en *1999* y su gran popularidad alcanzada en el año *2000*. En este sistema, millones de usuarios en todo el mundo encontraron una forma fácil de compartir canciones en formato *mp3* mediante un servicio de intercambio de archivos con un servidor central que mantenía el índice de los usuarios conectados y los archivos que cada uno de ellos tenía a disposición de los demás. *Napster* fue sancionado por facilitar la violación a los derechos de autor [4].

La importancia de *Napster* no finalizó con su cierre en el año *2001*. A partir de ese momento y gracias a una gran cantidad de trabajo investigativo a nivel mundial se han creado múltiples aplicaciones inspiradas en *Napster*. Algunos de los ejemplos más conocidos son: *Publius*, *Gnutella*, *Kazaa*, *FreeNet*, *MojoNation*, *OceanStore*, *FreeHaven*, *Scan*, *Emule*, *EDonkey*, *BitTorrent*, *Morpheus*, *Chord*, *Can*, *Pastry*, *Tapestry*, *Viceroy*, *Past* y *DHash*.

Los sistemas *peer-to-peer* han sido empleados en una variedad de aplicaciones como las siguientes: comunicación y colaboración; computación distribuida; servicios multicast; sistemas de bases de datos distribuidas sobre redes *overlay* con miles de nodos; y servicios de distribución de contenido, tal vez la más popular de las aplicaciones de los sistemas *peer-to-peer* [4].

Los primeros sistemas *peer-to-peer* (como *Napster*) manejaban un esquema centralizado en el cual un único nodo era el responsable de mantener un índice central para toda la red lo

que lo convierte en un punto único de falla, porque todas las consultas deben pasar por el servidor central. Por otra parte, el esquema centralizado tiene capacidades de escalabilidad muy limitadas porque llega el momento en el que el servidor central puede ser insuficiente para mantener el sistema [18].

Para contrarrestar los problemas que podían ocasionarse por un punto único de falla en el esquema centralizado, surgió una forma completamente descentralizada en la cual cada nodo mantiene un índice local de los archivos que desea compartir con los demás usuarios del sistema. En este caso, para realizar una búsqueda sobre la red, es necesario realizar una inundación hacia todos los nodos del sistema, para que cada nodo realice una búsqueda en su índice local. De este modo, se evita la centralización pero el uso de la inundación es demasiado costoso para resolver los problemas de escalabilidad. *Gnutella* es un sistema *peer-to-peer* que utiliza este mecanismo [4].

Entonces, con el fin de ofrecer una solución a los problemas de escalabilidad mencionados, surgen los sistemas *peer-to-peer* estructurados, los cuales mantienen la información en una tabla *hash* distribuida (*DHT*), donde cada nodo es responsable de sólo una parte de los objetos que se almacenan en la red, no tiene una vista global del sistema y sólo tiene conocimiento de una cantidad relativamente baja de nodos que pertenecen al sistema. Gracias a estas características, los sistemas *DHT* pueden crecer hasta miles de nodos [28]. En estos sistemas, un protocolo permite establecer cuál nodo es responsable de un objeto lo que permite que una búsqueda sea dirigida hacia el nodo responsable sin necesidad de contactar nodos innecesarios. Algunos ejemplos de sistemas estructurados son *Chord*, *Pastry*, *Tapestry* y *Can* [4].

En los últimos años se ha presentado una gran actividad investigativa y se ha desarrollado una variedad de aplicaciones construidas sobre *DHTs*. Por ejemplo, sistemas de búsqueda por palabras clave, búsquedas por múltiples atributos o búsquedas por rangos. Esta clase de búsquedas no pueden ser resueltas directamente por las *DHT* porque están diseñadas para la realización de búsquedas de coincidencia exacta por llave [35]. Otras aplicaciones *DHT* son sistemas de archivos y motores de procesamiento de consultas a gran escala [35].

A pesar de la cantidad de actividad investigativa adelantada desde que surgieron los sistemas *DHT*, aún quedan asuntos por resolver entre los cuales el desempeño es un desafío que actualmente ocupa la atención de muchos investigadores en el área [4].

El desempeño de un sistema *DHT* se ve afectado por diferentes motivos, entre ellos la ubicación de los objetos y los mecanismos de enrutamiento [4]. El uso de cachés es efectivo para alcanzar balance de carga, reducir la distancia de donde se trae un objeto y disminuir el tráfico de la red [29], lo que permite mejorar significativamente el desempeño de las aplicaciones *DHT*.

Por otra parte, los cachés pueden impactar los sistemas *DHT* especialmente en las dos capas inferiores (*DLS* y *DSS*)[1]. En la capa de búsqueda *(Distributed Lookup Service)*, el caché puede ser utilizado para resolver búsquedas de manera anticipada al contactar nodos que ya hayan sido contactados en búsquedas anteriores. En la capa de almacenamiento (*Distributed Storage Service*), el caché se utiliza para localizar una copia cercana de los objetos solicitados. Probablemente otra búsqueda por la misma llave, iniciada en nodos diferentes contacte nodos en común, y en este caso, la búsqueda termina más rápido al retornar un dato almacenado previamente en caché.

El beneficio de la utilización de cachés en sistemas *DHT* es la disminución en la latencia de acceso al reducir la distancia desde donde se traen los objetos lo que aumenta la productividad y balancea la carga de consultas. Cuando un objeto es enrutado a través de un nodo como parte de una operación de búsqueda o inserción, también es insertado en el caché local si su tamaño lo permite.

La configuración adecuada del caché es un proceso complejo porque son muchas las posibilidades que deben ser consideradas y en el proceso de desarrollo de una aplicación es necesario tomar decisiones, por lo general, sin tener en cuenta todas las opciones que se tienen a disposición. El desarrollador necesita elegir la arquitectura del caché, definir las políticas de reemplazo, el tipo y tamaño de los cachés y el protocolo de resolución de las fallas de caché. Por otra parte, determinar el tráfico de red y la cantidad de accesos a la aplicación aumenta el número de opciones que deben ser consideradas lo que incide directamente en el aumento de la complejidad en el proceso de configuración del caché y dificulta la selección de la opción más conveniente para una aplicación específica.

Teniendo en cuenta que un desarrollador de aplicaciones *DHT* puede considerar necesario usar cachés para mejorar el desempeño de una aplicación, y que la configuración adecuada de un caché es un proceso complejo, se plantea la construcción de una herramienta que permita realizar pruebas con diferentes estrategias de caché para obtener información que ayude al desarrollador de aplicaciones a establecer el esquema de caché que mejor se acomode a la aplicación.

A nivel conceptual, la aplicación está pensada para ser desarrollada sobre cualquier sistema *DHT* en forma genérica lo que quiere decir que el desarrollador plantea su aplicación en términos de las primitivas de servicio *get* y *put*, las cuales deben ser ofrecidas por la capa de almacenamiento de un sistema *DHT*. *DHTCaché* lo que hace es ponerse en medio de la aplicación construida por el desarrollador y el sistema *DHT* subyacente para crear un caché, tanto en la capa de búsqueda como en la capa de almacenamiento, de manera transparente para la aplicación, al igual que el monitoreo de una serie de eventos que se registran en archivos *log* para su análisis posterior. En otras palabras, la aplicación no tiene porqué modificar su comportamiento ni tener conocimiento si el caché está habilitado o no.

[1] Ver capítulo *2*, sección *2.2*.

Se espera que el desarrollador pueda probar diferentes configuraciones de caché y use las métricas que brinda *DHTCaché* para determinar cuál es la configuración de caché que más le favorece al desempeño de su aplicación. El desarrollador puede extender su análisis aprovechando los archivos *log* para realizar cálculos adicionales. Sin embargo, algunas aplicaciones pueden necesitar el cálculo de métricas diferentes que no se pueden obtener con la información registrada en los *logs*, por lo que el sistema no puede dar soporte a todo tipo de aplicaciones.

Para la realización del prototipo se utilizó *ACS*, un *framework* para la construcción de servicios adaptables de caché [13], y *FreePastry,* una plataforma para la prueba y desarrollo de aplicaciones *DHT* [38].

Este proyecto es uno de los trabajos de tesis conjuntos de maestría que se están desarrollando paralelamente bajo el contexto del proyecto europeo de cooperación internacional *ECOS-NORD*-Código *C06M02*.

Este documento se encuentra estructurado de la siguiente forma: el capítulo dos trata sobre los sistemas *peer-to-peer y* los sistemas *DHT* con las aplicaciones que son construidas sobre los sistemas *DHT*. En el capítulo tres se hace una presentación de los conceptos más importantes sobre el caché, y el caché distribuido se describe en el capítulo cuatro. La solución planteada se analiza en el capítulo cinco y el plan de pruebas en el capítulo seis. Al final, en el capítulo siete, se plantean las conclusiones de este trabajo y algunas consideraciones para trabajo futuro.

2. CONTEXTO

A continuación se presentan los conceptos fundamentales de los sistemas *peer-to-peer*, los sistemas *DHT*, el caché y el caché distribuido, teman en los cuales está el tema principal de este trabajo.

2.1 SISTEMAS *PEER-TO-PEER*

Los sistemas *peer-to-peer* son sistemas distribuidos formados por nodos interconectados con el propósito de compartir recursos. Los nodos pueden autoorganizarse y adaptarse a la continua entrada y salida de nodos del sistema sin necesidad de la intermediación o soporte de un servidor global o una autoridad central [4].

En sentido estricto, un sistema *peer-to-peer* es un sistema totalmente distribuido en el cual todos los nodos son equivalentes en términos de funcionalidad y tareas que desempeñan. No existe la noción de servidores o clientes, los miembros realizan operaciones tanto de servidor como de cliente, por lo que usualmente son llamados *servents* [4].

Un sistema *peer-to-peer* forma una red lógica de computadores llamada *overlay network* construida sobre una red física subyacente. La topología de la red lógica y el grado de centralización determinan el desempeño del sistema [4].

A continuación se mencionan algunos ambientes de ejecución de los sistemas *peer-to-peer*, las características, las topologías utilizadas en esta clase de sistemas, y la clasificación por el grado de centralización y estructura.

2.1.1 Aplicaciones

Los sistemas *peer-to-peer* tienen aplicación en diferentes contextos [8]. Por ejemplo, en comunicación y colaboración prestando soporte a aplicaciones de mensajería instantánea; en computación distribuida para aprovechar la capacidad de procesamiento de los *peers* enviando pequeñas unidades de trabajo y recogiendo después los resultados; y en distribución de contenido donde el objetivo es compartir archivos entre los usuarios, normalmente imágenes, videos, canciones, libros.

También se puede utilizar la tecnología *peer-to-peer* para la descarga de actualizaciones o nuevas versiones de programas o sistemas operativos antes realizadas desde un servidor central del fabricante lo que ocasionaba cuellos de botella haciendo lento el proceso de descarga [2].

Algunos de los ejemplos más conocidos son: *Napster*, *Publius*, *Gnutella*, *Kazaa*, *FreeNet*, *MojoNation*, *OceanStore*, *FreeHaven*, *Scan*, *Emule*, *EDonkey*, *BitTorrent*, *Morpheus*, *Chord*, *Can*, *Pastry*, *Tapestry*, *Viceroy*, *Past* y *DHash*.

2.1.2 Características

Las características principales de los sistemas *peer-to-peer* son:

- **Autoorganización y descentralización:** Al no depender de un sitio central que cumpla las labores de administración, el sistema debe estar en condiciones de establecer y reestablecer las relaciones entre los miembros participantes frente a la alta volatilidad de los nodos, manteniendo el sistema en operación.

- **Disponibilidad y tolerancia a fallas en la red y los nodos:** La redundancia de los objetos almacenados en diferentes nodos evita que se pierda el acceso ellos cuando se presenten fallas. Además, si se presentan fallas en nodos individuales o fallas en la red, el sistema no debe dejar de funcionar, sólo se debe afectar la parte directamente relacionada con los nodos que han presentado la falla.

- **Escalabilidad:** El desempeño del sistema depende del número de nodos u objetos en la red. Un incremento significativo en el número de nodos o en el número de objetos no debe afectar el desempeño o la disponibilidad del sistema. Sin embargo, la escalabilidad tiene límites, bien sea porque puede llegar el momento en que es mayor el costo tratando de aumentar la capacidad que el beneficio en desempeño, o simplemente el sistema no es capaz de mantenerse en operación.

- **Autonomía:** Cada nodo miembro del sistema tiene la capacidad de trabajar en el sistema *peer-to-peer* al mismo tiempo que realizar sus tareas individuales.

2.1.3 Topologías

La topología de una red *peer-to-peer* es la forma como están dispuestos los nodos en el sistema. Las redes *peer-to-peer* organizan los nodos participantes en una de las cuatro topologías básicas: estrella, en anillo, jerárquica y descentralizada [31].

- **Topología en estrella:** El concepto de topología en estrella está basado en el modelo tradicional cliente-servidor. Existe un servidor central, usado para manejar índices sobre los archivos y bases de datos de múltiples *peers* conectados. El cliente contacta al servidor para informarle su dirección *IP* y el nombre de todos los archivos que desea compartir. La información que el servidor recoge de los *peers* es usada por el servidor para crear un índice centralizado, que mapea los nombres de los archivos a un conjunto de direcciones *IP*. Todas las búsquedas son enviadas al servidor, quien ejecutará una búsqueda en su índice local. Si hay una coincidencia, se crea un enlace directo entre el *peer* que está compartiendo el archivo y el *peer*

que lo solicita. Luego, la transferencia es ejecutada. *Napster* es un ejemplo de una aplicación que usa esta topología.

- **Topología en anillo:** El inconveniente principal de una topología en estrella es que el servidor central se convierte en un cuello de botella y punto único de falla. Estos son algunos de los principales factores que contribuyeron a la aparición de la topología en anillo. Esta topología está construida sobre un grupo de máquinas que están dispuestas en forma circular para actuar como un servidor distribuido. Este grupo de máquinas trabaja unido con el fin de proporcionar mejor balance de carga y alta disponibilidad. La topología en anillo es generalmente usada cuando todas las máquinas están relativamente cerca una de la otra en la red, lo que significa que es más probable que sean propiedad de la misma organización donde el anonimato no es problema. Un ejemplo de sistema *peer-to-peer* que dispone sus nodos en una topología de anillo es *Chord*.

- **Topología jerárquica:** Esta topología es muy apropiada para sistemas donde hay *peers* que pueden dividirse en niveles jerárquicos o que involucran alguna forma de delegación de autoridad.

- **Topología descentralizada:** En una arquitectura *peer-to-peer* pura, un servidor centralizado no existe. Todos los *peers* son iguales, creando una topología de red plana no-estructurada. Para unirse a la red, un *peer* primero debe conectarse con un nodo que permanece en línea (nodo *bootstrap*), el cual da al *peer* que se está conectando la dirección *IP* de uno o más *peers* existentes, para que pueda conectarse al sistema. Cada *peer*, sin embargo, solo tendrá información acerca de sus vecinos, los cuales son *peers* que tienen un enlace directo con ellos en la red. Un ejemplo de una aplicación que utilice este modelo es *Gnutella*.

2.1.4 Clasificación

Existen dos formas de clasificar los sistemas *peer-to-peer*, bien sea por su grado de centralización o por su estructura [4].

Clasificación por su grado de centralización

Aunque en su forma más pura las redes *peer-to-peer* son totalmente descentralizadas, en realidad hay diferentes grados de centralización en sistemas aceptados como *peer-to-peer* [4].

- **Completamente descentralizados:** Todos los nodos desempeñan exactamente las mismas tareas, actúan como servidores y clientes a la vez y no hay ninguna clase de coordinación central en sus actividades.

- **Parcialmente descentralizados:** La base es la misma que en el caso anterior. Sin embargo, algunos nodos asumen un papel más importante actuando como índices de archivos compartidos para *peers* cercanos. Estos nodos especiales toman el nombre de supernodos o *superpeers*. Los supernodos no constituyen puntos únicos de falla, ya que son asignados dinámicamente y cuando un supernodo falla, el sistema lo reemplaza por otro.

- **Descentralizados híbridos:** En estos sistemas se utiliza un servidor que facilita la interacción entre los *peers*, mediante el mantenimiento de directorios de metadatos que describen los archivos compartidos por cada *peer*. Lo anterior con el fin de agilizar la búsqueda e identificación de los nodos que tienen los archivos deseados por otros *peers*. Naturalmente, el servidor central se convierte en punto único de falla lo que afecta sus propiedades de escalabilidad.

Clasificación por su estructura

Un sistema *peer-to-peer* se dice que es estructurado si la ubicación de los objetos guarda relación con la topología de la red *overlay* y existe un método para determinar si un objeto está almacenado en el sistema o no. Si por el contrario, la ubicación de objetos en los nodos se hace en forma no-determinista, se dice que el sistema es no-estructurado [4].

- **Sistemas *peer-to-peer* no-estructurados:** En estos sistemas, la ubicación de los objetos no tiene ninguna relación con la topología de la red *overlay*. Cuando es necesario buscar un objeto en una red no-estructurada, se utilizan métodos como la inundación de la red propagando la consulta hasta encontrar el objeto deseado. Estos mecanismos de búsqueda tienen implicaciones importantes en la escalabilidad y disponibilidad del sistema [4]. Los sistemas no-estructurados generalmente son apropiados para acomodarse a poblaciones de nodos altamente transitorias. Algunos ejemplos muy conocidos de sistemas no-estructurados son *Napster*, *Gnutella* y *Kazaa*.

- **Sistemas *peer-to-peer* estructurados:** Surgieron con el fin de resolver los problemas de escalabilidad que sufren los sistemas *peer-to-peer* no-estructurados. En los sistemas estructurados la topología de la red *overlay* está completamente controlada y los objetos son ubicados en lugares específicos. Estos sistemas proporcionan un mapeo entre los objetos y los nodos en los cuales se alojan, formando una tabla *hash* distribuida (*DHT*), de manera que las consultas pueden ser enrutadas eficientemente al nodo que almacena el objeto deseado. Los sistemas estructurados ofrecen una solución escalable para consultas de coincidencia exacta, es decir, consultas donde el identificador exacto del objeto solicitado es conocido [4]. Algunos ejemplos de sistemas estructurados son *Chord*, *Pastry*, *Tapestry* y *Can*.

Como fue mencionado en la introducción, el interés mayor de este trabajo se centra en los sistemas *peer-to-peer* estructurados, comúnmente llamados *DHT*, los cuales serán comentados en la siguiente sección.

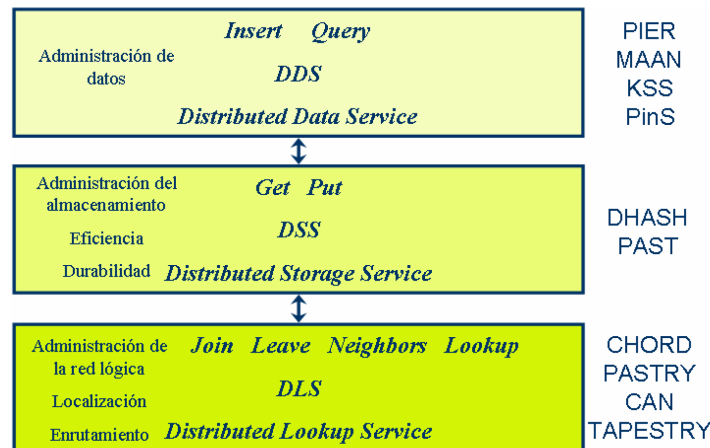
2.2 SISTEMAS *DHT*

Un sistema *DHT* hace parte de la segunda generación de sistemas *peer-to-peer* y es presentada como una solución al problema de escalabilidad que afecta a los sistemas no-estructurados [35]. Los sistemas *DHT* han venido ganando reconocimiento recientemente porque permiten la creación de sistemas distribuidos robustos y de gran escala, en los cuales se pueden localizar objetos mediante el intercambio de mensajes en un orden logarítmico [12].

Los sistemas *DHT* pueden ser usados como soporte para la creación de nuevos servicios distribuidos de alta escalabilidad y tolerantes a fallas, tales como almacenamiento cooperativo de archivos, distribución de contenido, y aplicaciones de comunicación [12].

Las aplicaciones que se crean sobre un sistema *DHT* generalmente comparten características similares y con el fin de estudiarlas, se puede hacer una división en tres capas funcionales [35]:

Figura 1. Arquitectura de los sistemas DHT



A continuación se presenta una descripción de las capas mencionadas explicando las operaciones y los servicios que ofrece cada una de ellas.

2.2.1 *Distributed Lookup Service* (*DLS*)

Figura 2. Servicio distribuido de búsqueda (DLS)

La capa *DLS* desempeña las actividades relacionadas con la administración de la red lógica (*overlay*), la localización y el enrutamiento.

En la administración de la red lógica se maneja la topología lógica del sistema *peer-to-peer*. Las topologías más usadas por los diferentes algoritmos que implementan la capa *DLS* son: anillo, árbol, espacio coordenado y mariposa.

La localización consiste en determinar cuál es el nodo responsable de almacenar una llave. Generalmente se calcula a través de una función de *hashing* (*SHA-1*[2]), para obtener los identificadores de nodos e identificadores de llaves que permiten mapear una llave en el nodo que será el responsable de almacenarla.

El enrutamiento es el proceso de enviar una llave al nodo que la va a almacenar, a través de nodos intermedios. También se refiere al envío de una consulta por una llave al nodo que la tiene almacenada, igualmente a través de nodos intermedios.

En esta capa, se ofrecen cuatro servicios:

- *Join (key)*
- *Leave (key)*
- *Neighbors (key)*
- *Lookup (key)*

En el momento en que un nodo se va a unir al sistema, se calcula un identificador para el nodo utilizando una función de *hashing* sobre la dirección *IP* [6]. Esta función retorna la llave (*key*) con la cual se puede establecer la posición que el nodo va a tener dentro del sistema *peer-to-peer* estructurado, de acuerdo con la topología de red *overlay* que maneje el algoritmo que ofrece los servicios de esta capa.

Join (key) es ejecutado cuando un nodo se va a unir al sistema. Luego toma su posición y recibe de los nodos vecinos las llaves que le corresponde almacenar de acuerdo con esa posición. De forma similar, *Leave (key)* se ejecuta si un nodo desea salir del sistema. Antes

---

[2] *SHA-1*: *Secure Hash Algorithm*, Algoritmo de *Hash* Seguro. Es un método que calcula un resumen (*digest*) de longitud fija (*160* bits) a partir de un objeto cualquiera. Si el objeto es modificado, así sea en un solo bit, el resultado del *SHA-1* cambia, por lo que es utilizado para validar la integridad del objeto [33].

de abandonar el sistema, entrega al nodo que le corresponda, las llaves que tiene almacenadas para que el sistema continúe en operación. La forma como se eligen las llaves de las que es responsable cada nodo, depende del algoritmo que implementa la capa *DLS*.

Neighbors (key) es la operación que se utiliza para determinar los vecinos de un nodo con la llave especificada. Los vecinos de un nodo son utilizados para almacenar réplicas de los objetos con el fin de dar al sistema una mayor resistencia a fallas.

Lookup (key) retorna la dirección *IP* del nodo que guarda la llave *key* especificada. La consulta debe ser enviada al nodo responsable a través de nodos intermedios y la forma de enrutar la respuesta depende de cada algoritmo en particular.

Existe una variedad de productos que implementan la capa *DLS*. Todos ofrecen un servicio de *lookup* escalable en un ambiente distribuido con frecuente entrada y salida de nodos; establecen un espacio lógico; y manejan información parcial de enrutamiento. Sin embargo, cada uno aplica un algoritmo diferente. A continuación una breve descripción de los productos más representativos: *Chord*, *Pastry*, *Tapestry* y *Can* y un breve análisis comparativo de sus características más importantes.

Chord

Chord es un poderoso sistema de búsqueda distribuido que soporta una operación muy sencilla: dada una llave, la mapea en un nodo [30].

Cada nodo en *Chord* tiene asignado un identificador único de *m* bits, el cual es calculado utilizando una función de *hashing* (*SHA-1*) sobre la dirección *IP* del nodo. De forma similar, las llaves de los objetos también son proyectadas en el anillo *Chord* haciendo *hashing* sobre ellas. Tanto los identificadores de los nodos como los identificadores de las llaves son dispuestos en un espacio circular ordenado denominado anillo *Chord* cuyo tamaño es 2^m, donde los nodos son numerados entre 0 y $2^m -1$ [30].

Chord mapea cada identificador *id* en el nodo que tenga el menor identificador entre los nodos con identificador igual o mayor que *id,* es decir, el siguiente nodo en el anillo *Chord* en el sentido de las manecillas del reloj. El nodo que lo almacena se conoce con el nombre de sucesor de *id* [10].

Cuando un nodo nuevo llega al sistema, solo algunas de las llaves de su sucesor son trasladadas al nuevo nodo y al abandonar el sistema, todas sus llaves son transferidas a su nodo sucesor [30].

Chord usa dos estructuras de datos para realizar las operaciones de búsqueda: una lista con los sucesores inmediatos y una lista especial llamada *Tabla Finger* con apuntadores en la cual los nodos están espaciados exponencialmente alrededor del anillo *Chord* [6]. La *tabla Finger* contiene mayor cantidad de nodos cercanos que nodos lejanos y contiene *m* entradas [30].

Chord es eficiente, porque para determinar el nodo sucesor de un identificador requiere solo el intercambio de *O(Log n)* mensajes para un sistema de *n* nodos [10].

Pastry

Pastry es un esquema de localización y enrutamiento basado en redes *overlay* autoorganizado, completamente descentralizado, tolerante a fallas, escalable y confiable. Ha sido pensado como soporte para la construcción de aplicaciones *peer-to-peer* [28].

A cada nodo en una red *Pastry* se le asigna un identificador único (*nodeId*) de *128* bits. El *nodeId* es usado para establecer la posición del nodo en un espacio circular de identificadores, el cual va desde *0* hasta $2^{128}-1$, similar a *Chord* [28].

Cada nodo *Pastry* mantiene una tabla de enrutamiento, una tabla con los vecinos (*neighborhood set*) más cercanos de acuerdo a una métrica de proximidad definida (por ejemplo la cantidad de saltos [8] o el *RTT*[3] y una tabla (*leaf set*) con los nodos con identificador más cercanos [28].

Un nodo *Pastry* enruta eficientemente el mensaje al nodo con el *nodeId* más cercano a la llave entre todos demás nodos del sistema. En cada salto de enrutamiento, un nodo reenvía el mensaje a otro nodo cuyo *nodeId* comparta con la llave un prefijo (parte inicial del *nodeId*) de al menos un dígito más que el prefijo que comparte con el nodo actual. *Pastry* es eficiente en el enrutamiento porque el número esperado de saltos es *O(Log n)* donde *n* es el número de nodos en la red [28].

Pastry toma en cuenta la red física, es decir, busca la distancia mínima para cada salto en la red *IP*. Cada nodo mantiene la lista de los vecinos inmediatos y notifica a las aplicaciones la llegada o salida de nodos del sistema [28].

Tapestry

Tapestry es una infraestructura de enrutamiento distribuida eficiente, escalable y de alto desempeño para redes *peer-to-peer* estructuradas [41].

Los nodos en una red Tapestry tienen asignado un identificador (*nodeId*) dentro de un gran espacio identificador de *160* bits que usa una función de *hashing* como *SHA-1*. *Tapestry* mapea cada identificador en un nodo [41].

Cada nodo mantiene un mapa de vecinos, el cual tiene múltiples niveles, donde cada nivel *l* contiene apuntadores a nodos cuyos identificadores coinciden con el identificador del nodo en *l* dígitos [4].

---

[3] *RTT: Round Trip Time*. Tiempo que tarda el envío de un paquete pequeño de un computador a otro y recibir su respuesta [19].

Cuando se necesita enrutar un mensaje hacia un nodo, los mensajes son reenviados a través de nodos vecinos cuyos identificadores son progresivamente más cercanos al nodo en el espacio identificador, es decir, cada vez tienen mayor coincidencia en el identificador del nodo [41]. *Tapestry* alcanza su objetivo en *O(Log n)* saltos donde *n* es el número de nodos en la red [6].

Al igual que *Pastry*, *Tapestry* tiene en cuenta la distancia en la red física en el momento de construir las tablas de enrutamiento de los nodos [41]. Además utilizan enrutamiento basado en prefijo/sufijo de direcciones basado en el mismo principio, el algoritmo de *Plaxton, et.al.* [40]. También tienen algunas diferencias, especialmente en la técnica para localizar el nodo numéricamente más cercano en el espacio de identificadores de nodos y en el manejo de la replicación. Mientras *Pastry* replica los objetos en sitios aleatorios en la red, *Tapestry* ayuda a ubicar la copia más cercana del objeto deseado [18].

Can

Can es un sistema que implementa un esquema de enrutamiento y localización distribuida, escalable, tolerante a fallas, robusto y completamente autoorganizado [26]. *Can* usa un espacio cartesiano de *d*-dimensiones, con identificadores de nodos de 128 bits que definen un punto en el espacio, donde cada nodo es responsable de una zona del espacio de coordenadas [4].

Cada nodo de la red *Can* almacena una parte (llamada zona) del espacio coordenado, así como información acerca de un pequeño número de zonas adyacentes en la tabla. Las solicitudes para insertar, buscar o eliminar una llave particular son enrutadas a través de las zonas intermedias al nodo que mantiene la zona que contiene la llave [4].

Cualquier llave es mapeada en forma determinista en un punto *P* del espacio de coordenadas usando una función de *hashing*. Así la pareja (llave, objeto) es almacenada en el nodo responsable de la zona que corresponda al punto *P* [26] y [4].

Cuando un nuevo nodo llega a una red *Can*, el nuevo nodo elige un punto *P* al azar dentro del espacio coordenado y envía un mensaje especial (*JOIN*) al nodo responsable de la zona correspondiente al punto P el cual le asigna al nuevo nodo la mitad la zona controlada por el nodo que recibe la solicitud *JOIN* [26]. Además, al nuevo nodo le son transferidas las llaves correspondientes a la zona que está manejando. Si un nodo sale del sistema, las zonas que maneja y sus correspondientes llaves son entregadas a uno de sus vecinos [4].

Un nodo *Can* mantiene una tabla de enrutamiento que guarda la dirección *IP* y la zona de coordenadas de cada uno de sus inmediatos vecinos en el espacio de coordenadas. En un espacio *d* dimensional, dos nodos son vecinos si la extensión de sus coordenadas se traslapan a lo largo de *d-1* dimensiones y comparten el lindero en una dimensión [26].

Can es eficiente porque para un espacio con *d* dimensiones partido en *n* zonas (una por cada nodo), la longitud promedio de un camino de enrutamiento es $O((d/4)(n^{1/d}))$ saltos y los nodos individuales mantienen *2d* vecinos [26].

Es de anotar que existen diferentes caminos entre dos nodos, lo que le da a *Can* la capacidad de enrutar mensajes aún en la presencia de fallas en los nodos [26].

Análisis

Los cuatro algoritmos de búsqueda mencionados tienen semejanzas y diferencias entre sí. Con respecto al espacio lógico *Chord*, *Pastry* y *Tapestry* usan un espacio circular (anillo) ordenado de una dimensión para organizar los nodos y cada llave es mapeada al nodo responsable de esa llave, mientras que *Can* usa un espacio cartesiano *d*-dimensional, y utiliza un punto en ese espacio lógico para el mapeo de llaves.

El nodo responsable de una llave en *Chord* es el nodo sucesor de la llave. En *Pastry*, es el nodo numéricamente más cercano. *Tapestry* utiliza un nodo raíz que es elegido mediante la aplicación de un algoritmo determinista. En el caso de *Can* el nodo responsable de una llave es el que maneja la zona en la cual cae el punto *P*.

Chord maneja identificadores de *160* bits, al igual que *Tapestry*; los identificadores en *Pastry* y en *Can* son de *128* bits. En los cuatro algoritmos analizados, se utiliza una función de *hashing* como *SHA-1* para determinar los identificadores de los nodos y los objetos.

Para el enrutamiento de mensajes, cada nodo mantiene una tabla de enrutamiento en la cual las entradas son parejas (identificador del nodo, dirección *IP*) para nodos con los cuales mantiene enlaces sobre la red *overlay*. Los mensajes son reenviados a través de enlaces en la red *overlay* cuyos identificadores son progresivamente más cercanos a la llave en el espacio identificador [12].

Cuando un nuevo nodo se une al sistema, sin importar cuál sea el algoritmo, el nuevo nodo recibe las llaves que le corresponde manejar. En *Chord* algunas de las llaves almacenadas por su sucesor son enviadas al nuevo nodo. En *Pastry* y *Tapestry*, sucede igual pero con los nodos más cercanos en el espacio identificador. Si un nuevo nodo llega a una red *Can*, se calcula un punto en el espacio de coordenadas y la zona en la cual cae el punto es dividida en dos partes iguales y toma el control de una de ellas, además recibe las llaves que hacen parte de la zona que va a controlar el nuevo nodo. Un proceso inverso ocurre cuando un nodo sale del sistema [12].

2.2.2 *Distributed Storage Service* (*DSS*)

Figura 3. Servicio distribuido de almacenamiento (DSS)

En esta capa se desempeñan las actividades relacionadas con la administración del almacenamiento, la durabilidad y la eficiencia para ofrecer almacenamiento persistente, confiable y escalable construido sobre un servicio de *lookup* en el cual hay frecuente entrada y salida de nodos [35].

La administración del almacenamiento determina el *peer* responsable de una llave apoyado en los servicios de ofrece la capa de *lookup*. De esta manera, envía los objetos al *peer* que los va a almacenar y los recupera cuando son requeridos por la capa de datos (la capa superior). Además está pendiente de la entrada y salida de nodos para mover los objetos a los nodos siempre que sea necesario para garantizar la consistencia del sistema.

La eficiencia es la capacidad que tiene esta capa de administrar el almacenamiento de objetos en los nodos de manera escalable, en términos de *O(Log n)*, donde *n* es el número de *peers* del sistema.

La durabilidad es la propiedad que se encarga de garantizar la persistencia los objetos almacenados en el sistema frente a fallas en los nodos. Cuando la capa *DSS* da la orden de guardar un objeto en un *peer*, también monitorea la entrada y salida de nodos para que los nuevos responsables de una llave tengan almacenados los objetos que le correspondan. De este modo, la capa superior asume que este servicio funciona correctamente.

En esta capa, se ofrecen dos servicios:

- *Put (key, object)*
- *Get (key)*

Put (key, object) envía el objeto especificado al nodo responsable de la llave *key* para que sea almacenado en ese *peer*. Esta operación se apoya en los servicios ofrecidos por la capa de *lookup*.

Get (key) permite obtener el objeto asociado a la llave *key*. La consulta se enruta al *peer* responsable del objeto utilizando los servicios de la capa *DLS*.

Los dos productos más importantes que implementan la capa de almacenamiento son *DHash* y *Past*. Ambos ofrecen almacenamiento persistente, confiable y escalable

construido sobre un servicio de *lookup* en el cual hay frecuente entrada y salida de nodos. A continuación una breve descripción de ellos y un análisis comparativo.

DHash

DHash implementa una tabla *hash* distribuida sobre *Chord*. *DHash* proporciona un *API* que ofrece los servicios *put* y *get* y permite a las aplicaciones *peer-to-peer* almacenar objetos en los nodos del sistema y obtenerlos, si se conoce la llave de los objetos [12].

DHash trabaja asociando la llave de un objeto con un nodo, haciendo que los llamados a *get* y *put* usen la llave como identificador [12]. Una llave en *DHash* es el resultado del cálculo del *hashing* del objeto y tiene una longitud de *160* bits [9].

DHash usa *Chord* para localizar el nodo responsable de un objeto [11]. *Chord* proporciona el servicio de localización, mientras que *DHash* es responsable del almacenamiento de datos y de la creación y mantenimiento de réplicas de los objetos almacenados [9].

DHash hereda las propiedades de *Chord* porque requiere el intercambio de *O(Log n)* mensajes entre los diferentes nodos de la red al realizar las operaciones *put* y *get* [11].

Mediante la replicación de objetos en los *n* sucesores más cercanos, *DHash* ofrece disponibilidad frente a los problemas que se pueden presentar por la constante entrada y salida de nodos en el sistema y frente a la posibilidad de fallas en los nodos, de ese modo, si un nodo falla, una réplica de sus objetos probablemente estará en el sucesor del nodo que ha fallado [12] y [11]. El valor del parámetro *n* es configurable y permite establecer la capacidad deseada de resistencia frente a fallas en el sistema [11].

DHash mantiene un caché de objetos para evitar la sobrecarga ocasionada por la recuperación de objetos populares y de ese modo puede aumentar el desempeño y distribuir mejor la carga entre los nodos del sistema. Los objetos pueden ser almacenados en caché a lo largo del camino que se ha utilizado para contactar el sucesor de una llave, previendo que un nuevo acceso al mismo nodo pueda ser resuelto más rápidamente que el anterior, previamente guardado en caché [11].

Aunque los conceptos de replicación y caché son similares, *DHash* hace una diferencia notoria, porque utiliza la replicación en sitios predecibles y se asegura que las réplicas siempre existan, mientras que el caché no siempre va a existir y cuando exista, no siempre se va a saber con anticipación dónde estará ubicado [11].

Past

Past es una utilidad de almacenamiento global *peer-to-peer* persistente, escalable, seguro y de alta disponibilidad. Está basado en la autoorganización de una red *overlay* sobre Internet con nodos que pueden iniciar y enrutar solicitudes para insertar y recuperar archivos, y de forma opcional, los nodos también pueden ofrecer almacenamiento en disco. Los archivos

que hacen parte del sistema son replicados para aumentar la disponibilidad, además, *Past* usa caché de archivos populares para mejorar el tiempo de respuesta [29].

Past aprovecha la diversidad de los nodos conectados a Internet para alcanzar persistencia y alta disponibilidad, lo que permite tener archivos de datos y copias de seguridad en diferentes lugares y facilita el uso de ancho de banda compartido por un grupo que al trabajar cooperativamente puede almacenar o publicar contenido que podría de algún modo exceder la capacidad de los nodos individuales [29].

Past está construido sobre *Pastry*, quien se encarga de enrutar eficientemente las solicitudes de los clientes (con alta probabilidad) a un nodo cercano al cliente que hizo la solicitud. El número de nodos contactados y el número de mensajes intercambiados en el enrutamiento de una solicitud tiene una complejidad logarítmica con respecto al número de nodos en el sistema [29].

Past maneja réplicas de archivos en *k* nodos, donde el parámetro *k* es especificado por el usuario para alcanzar la disponibilidad deseada, sin embargo, puede ser un valor que tenga en cuenta las fallas esperadas de nodos individuales en el sistema. Los archivos tienen un identificador (*fileId*) de *160* bits que se calcula usando la función de *hashing SHA-1*. De manera similar, los nodos tienen un identificador (*nodeId*) de *128* bits utilizado para determinar su posición en el espacio circular de identificadores de *Pastry*. Durante el proceso de inserción de un archivo, *Past* almacena réplicas en los *k* nodos más cercanos numéricamente tomando los 128 bits más significativos del identificador del archivo [29].

A pesar de las *k* réplicas que *Past* mantiene en el sistema por motivos de disponibilidad, un archivo muy popular puede afectar su desempeño en términos de latencia y tráfico de red. Por lo tanto, para mejorar estas métricas de desempeño, *Past* utiliza el caché de archivos populares en zonas libres de disco [29].

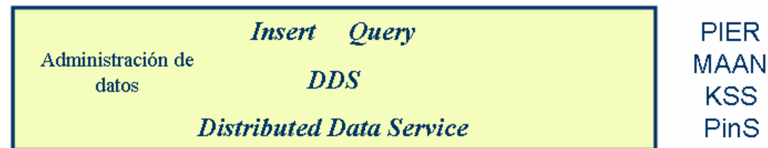
Análisis

Tanto *DHash* como *Past* ofrecen fundamentalmente el mismo servicio, los dos suponen que en la capa inferior encuentran un servicio de *lookup* confiable. La principal diferencia es que *DHash* se construye sobre *Chord* mientras que *Past* está implementado sobre *Pastry*.

Los dos sistemas ofrecen una alta escalabilidad al ser servicios con una complejidad logarítmica en el manejo de sus operaciones. Ambos usan replicación y caché para tolerancia a fallas y aumento de la disponibilidad.

2.2.3 *Distributed Data Service* (*DDS*)

Figura 4. Servicio distribuido de datos (DDS)



Los servicios *DHT* en las dos capas inferiores (*DLS* y *DSS)* están en capacidad de realizar búsquedas de objetos cuyo identificador coincida exactamente con una llave dada.

La capa de datos tiene la misión de administrar los datos que las aplicaciones necesitan almacenar en los diferentes nodos y se encarga de la insertar los objetos y realizar las consultas, aprovechando los servicios que le ofrece la capa *DSS*. Igualmente se encarga de las tareas relacionadas con el manejo de consultas de más alto nivel que las que se pueden hacer en la capa de almacenamiento, porque por lo general, los actuales sistemas *DHT* asumen que sus aplicaciones ya conocen la llave del objeto deseado. Esta funcionalidad no es suficiente porque existen numerosos recursos que pueden tener más de un atributo o se necesita ofrecer el servicio de búsqueda por palabras clave. También se presenta la posibilidad de realizar consultas por rangos en lugar de valores exactos, especialmente cuando se trata de objetos cuyo valor corresponde a un número. Es decir, en esta capa se presta el servicio de traducción de consultas por palabras clave, por rangos o por múltiples atributos a consultas por llave que finalmente sean entendidas por la capa de almacenamiento [37].

Los servicios que se ofrecen en esta capa son:

- *Insert*
- *Query*

Insert, solicita a un *peer* la inserción de un objeto, apoyado en *put* ofrecido por la capa *DSS*, a quien le corresponde la responsabilidad de gestionar el almacenamiento del objeto en el *peer* adecuado. En esta capa, se desconoce completamente la dirección del nodo que almacena el objeto [37].

Query solicita a un *peer* uno o más objetos. Si es del caso, solicita una parte de un objeto. Dependiendo de la aplicación, puede realizar consultas compuestas con conjunciones o disyunciones. En este caso, es necesario consolidar la respuesta [37].

Las consultas en esta capa pueden incluir información semántica, generalmente usando metadatos, logrando de esta manera ofrecer consultas más elaboradas que las que se pueden hacer en la capa de almacenamiento donde es necesario conocer la llave del objeto que se desea buscar [36].

Existe una gran variedad de productos en esta capa, entre los que se destacan los sistemas de búsqueda *Pier*, *PinS*, *Kss* y *Maan*, los cuales se describen a continuación, con su respectivo análisis comparativo.

Pier

Pier - Peer-to-peer Information Exchange and Retrieval, es un motor de consultas que puede escalar hasta miles de nodos participantes, pensado para hacer consultas sobre Internet. Está construido sobre un sistema *DHT*, para aprovechar sus características de escalabilidad, confiabilidad y tolerancia a fallas [16].

Los objetos que se almacenan en este sistema, por lo regular son filas de una tabla relacional y son almacenados en grupos de nodos. *Pier* permite el acceso a los datos almacenados en nodos mismo grupo mediante comunicación *multicast* [35].
La arquitectura de *Pier* se compone de tres niveles [16]:

- Capa de aplicación
- Capa *Pier*
- Capa *DHT*

En la capa *DHT*, *Pier* utiliza *Can* para enrutamiento de mensajes y un manejador de almacenamiento temporal de los datos, bien sea en memoria principal o en un sistema de archivos tradicional [16].

El esquema de nombramiento es muy importante en *Pier* y es utilizado para traducir identificadores de alto nivel a llaves que puedan ser entendidas por el *DHT*. Cada objeto tiene como atributos de identificación tres campos llamados *namespace*, *resourceId* e *instanceId*, utilizados para calcular la llave *DHT* mediante una función de *hashing* [16].

El *namespace* identifica la aplicación o grupo al que pertenece el objeto; el *resourceId* generalmente almacena la llave primaria del objeto que se desea almacenar en el *DHT*. El tercer campo, *instanceId*, se utiliza para evitar las colisiones que se puedan presentar cuando existan objetos con el mismo *namespace* y el mismo *resourceId* [16] y [35].

Pier utiliza un campo con el tiempo de vida cuando un objeto se va a guardar en el sistema. Ese es el mecanismo utilizado por *Pier* para lograr confiabilidad frente a las constantes fallas y desconexiones en los nodos. El tiempo de vida permite que los datos en el sistema sean temporales, sin embargo, el usuario puede renovar la vigencia de los datos cada vez que lo desee [16].

PinS

PinS - Peer to Peer Interrogation and Indexation system based on DHTs es un *middleware* para la realización de consultas e indexación en sistemas *DHT*. *PinS* extiende las

capacidades de consulta de los sistemas *DHT* permitiendo ejecutar consultas de comparación de igualdad y por rangos. *PinS* permite insertar cualquier tipo de objetos en el sistema (video, audio, texto, fotografía, etc.) y utiliza metadatos en forma de parejas (*nombreAtributo*, *valor*) para describir objetos. Los metadatos también son insertados en el sistema para hacer posible la realización de consultas de localización [36].

PinS propone varias estrategias de ejecución para ofrecer optimización de consultas cada una de ellas con diferentes características de desempeño. Estas estrategias proporcionan una optimización de consultas con base en las características del sistema y en la calidad del servicio requerida. Los principales objetivos de *PinS* son ofrecer la capacidad de realizar consultas de localización de objetos a través de metadatos en un sistema *DHT*, adicionar semántica a las consultas y proporcionar un eficiente desempeño en la evaluación de consultas [37].

Las actuales soluciones involucran un número grande de *peers* en la evaluación de consultas por rangos, imponen restricciones en la función *hash* o modifican el comportamiento del sistema *DHT peer-to-peer* subyacente [38]. *PinS* tiene una arquitectura *peer-to-peer* pura ya que no cuenta con catálogos centralizados ni *peers* dotados con características especiales; además, no requiere funciones de *hashing* con propiedades particulares y puede ser usada con cualquier sistema *DHT* [37].

Kss

Kss - A Keyword-set Search System for peer-to-peer networks es un sistema de búsqueda *peer-to-peer* que usa un índice invertido distribuido. Un índice invertido es una estructura de datos que mapea palabras a documentos y ayuda a encontrar rápidamente el documento en el cual aparece la palabra dada. Un índice invertido es una lista en la que se asocian las palabras más importantes de cada documento con apuntadores a los documentos donde aparecen. En un índice invertido distribuido, en lugar de almacenar todo el índice en un único servidor, se distribuye el índice en múltiples nodos en la red [14].

El esquema más natural es la partición del índice por palabras clave, pero esto implica la transmisión de una gran cantidad de datos por la red. *Kss* construye el índice invertido por conjuntos de palabras clave en el cual la cantidad de información que es necesario enviar por la red es mucho menor, especialmente cuando se trata de consultas de múltiples palabras [14].

Kss puede ser implementado en cualquier sistema *DHT*, tal como *Chord*, *Pastry*, *Can* y *Tapestry*. El trabajo presentado en [14] describe una aplicación de búsqueda de archivos musicales en una red de archivos compartidos usando metadatos.

Kss trabaja de la siguiente forma: cuando un usuario comparte un archivo, *Kss* genera la entrada al índice por cada conjunto de palabras en el archivo, calcula en *hashing* al conjunto de palabras clave para formar la llave de entrada al índice y mapea la llave en el nodo usando el sistema *DHT* subyacente.

Para encontrar la lista de documentos que coinciden con una consulta, el sistema encuentra los nodos que almacenan las entradas al índice para cada conjunto de palabras clave mediante el *hashing* de las palabras clave de la consulta, trae cada lista, y calcula el resultado de la intersección de las listas para encontrar una lista de documentos que contiene todas las palabras clave en la consulta.

Maan

Maan – A Multiattribute addressable network for Grid Information Services es un sistema de búsqueda por múltiples atributos basada en consultas por rangos sobre un sistema *peer-to-peer* estructurado [6].

Maan permite la realización de consultas por rangos y por múltiples atributos. *Maan* trata las consultas por rangos mediante el mapeo de valores de atributos al espacio identificador de *Chord* a través de *hashing* que preserve la localidad uniforme. *Maan* usa un algoritmo de enrutamiento iterativo o dominado por un único atributo para resolver consultas basadas en múltiples atributos [6].

Maan usa *SHA-1*, (el cual genera una distribución uniforme de identificadores, pero destruye la localidad de las llaves), para asignar identificadores a cada nodo y a los valores de atributos cuando el valor permite la coincidencia exacta (por ejemplo tipo *String*). En el caso de atributos con valores numéricos, *Maan* usa funciones de *hashing* que preserven la localidad para asignar a cada valor de atributo un identificador en el espacio de identificadores [6].

Análisis

Hay diferentes trabajos realizados para extender la capacidad de los sistemas *DHT*. Los cuatro productos mencionados tratan las consultas de localización a través de la introducción de metadatos en el sistema para adicionar semántica a los objetos almacenados.

Algunas diferencias entre *Pier*, *PinS*, *Kss* y *Maan* se pueden identificar. Por ejemplo, *Maan* necesita utilizar una función de *hashing* que no afecte la localidad de los objetos cuando se trata de almacenar valores numéricos, mientras que los otros sistemas no necesitan una función de *hashing* especial. *Pier* es un motor de consultas que almacena generalmente tablas de bases de datos relacionales, *PinS* puede realizar consultas de comparación de igualdad y por rangos, *Kss* está orientado al manejo de búsqueda por conjuntos de palabras clave y *Maan* permite la búsqueda por múltiples atributos.

3 CACHÉ

El caché es una forma de almacenamiento que permite mejorar el tiempo de respuesta de un sistema. Puede ser una parte reservada de la memoria principal, o de un dispositivo de almacenamiento secundario [22]. El caché se encuentra en diferentes contextos. Entre los más destacados se encuentran: la memoria caché (en los microprocesadores), el caché de disco, el caché de Web y el caché en los servidores *DNS*. A continuación se da una breve descripción del caché en cada uno de los contextos mencionados.

En el caché de memoria, el objetivo es disminuir el tiempo de envío de datos desde la memoria principal al microprocesador. El caché de los microprocesadores modernos es una memoria, que comparada con la memoria principal es más pequeña, más costosa y con un tiempo de acceso mucho menor. Esta memoria tiene un tamaño fijo construido dentro del microprocesador y se llama caché de nivel 1 (*L1*). La mayoría de los computadores personales también traen una memoria caché externa llamada caché de segundo nivel (*L2*). Las memorias caché de nivel 1y 2 son más costosas que la memoria principal. Sin embargo, con respecto a *L1*, la memoria caché *L2* es menos costosa y de mayor capacidad [22].

El caché de disco trabaja bajo el mismo principio de la memoria caché de los microprocesadores. El caché está ubicado en la memoria principal (*RAM*) y el objetivo es aumentar la velocidad de acceso a los datos que están en el disco. Los datos accedidos más recientemente del disco (y los sectores adyacentes) son almacenados en el caché. Cuando un programa necesita acceder a los datos del disco, primero revisa si los datos están en el caché. El caché de disco puede mejorar significativamente el desempeño de aplicaciones, porque acceder datos de la *RAM* es muchísimo más rápido que hacerlo del disco duro [22].

El caché de Web es también llamado servidor *Proxy*. Un servidor *proxy* es un programa intermediario que resuelve las solicitudes *http*. Para las solicitudes que no puede atender, realiza una nueva petición a otro servidor para poder dar respuesta a la solicitud inicial [7]. Un servidor *proxy* por lo general está ubicado dentro de la red local, sin embargo, puede ubicarse también cerca de un servidor Web. Un servidor *proxy* además, mantiene copias de las páginas Web que han sido visitadas recientemente por sus clientes. Cuando se realiza una solicitud *http*, primero se busca en el caché para verificar si allí hay una copia del sitio Web solicitado. Si se encuentra en el caché, no es necesario ir hasta el sitio especificado en Internet para dar respuesta a la solicitud. Por el contrario, si no hay una copia del sitio Web en el caché, es el *proxy* el que se encarga de hacer la solicitud al servidor Web de origen. Cuando recibe los objetos solicitados, el *proxy* los almacena y los envía al cliente que realizó la solicitud inicialmente. El caché de Web se usa cada vez más, porque aprovecha las conexiones rápidas entre los clientes y el caché, y disminuye significativamente el tráfico en organizaciones con una gran cantidad de usuarios, por ejemplo una universidad [19].

Los servidores de nombres de dominio en Internet usan ampliamente el caché para mejorar el rendimiento reduciendo el número de mensajes que viajan por la red. Cuando un servidor *DNS* recibe una respuesta que contiene la pareja (nombre de *host*, dirección *IP*), puede guardarla en su memoria local. De este modo, cuando llegue otra consulta para el mismo *host*, el servidor *DNS* puede resolver esa consulta aún si no es el servidor *DNS* autoritativo para el *host* solicitado [19].

Por otra parte, en el caché se almacenan habitualmente datos que han sido usados recientemente o datos que se presume pueden ser utilizados en un futuro cercano, con base en el principio de localidad ya sea espacial o temporal. El principio de localidad temporal dice que si un elemento es accedido, existe una alta probabilidad de ser accedido de nuevo en el futuro [3]. En el caso de la localidad espacial, si un elemento es accedido, existe una alta probabilidad de acceder en el futuro los elementos cercanos [3]. El aprovechamiento del principio de localidad hace que el caché aumente el desempeño del sistema.

Extendiendo el concepto de caché de disco, los sistemas distribuidos mejoran notablemente su rendimiento si se puede guardar una copia local de datos que se requieren con frecuencia y cuya fuente original es remota. El beneficio real del caché depende de la calidad de los enlaces entre los diferentes sitios del sistema distribuido.

Para medir el rendimiento de un caché, se contabilizan los aciertos y fallos en la búsqueda de los objetos que tenga almacenados. Un acierto (*hit*) se cuenta cada vez que un objeto solicitado está en el caché. Por el contrario, un fallo (*miss*) se acumula si el objeto solicitado no se encuentra en el caché. La efectividad del caché es juzgada por su *hit rate*, es decir, el número de aciertos (*hits*) dividido entre el número de intentos (aciertos+fallos) [24]. Cuando se trata de objetos que tienen tamaño homogéneo, el *hit rate* es una medida adecuada, pero si el tamaño de los objetos es variable, el *byte hit rate* es una mejor medida de desempeño. El *byte hit rate* se define como el número de bytes recuperados del caché, con respecto al total de bytes de los objetos solicitados [34].

Buscando mejorar aún más el desempeño de muchos sistemas de caché, se utilizan técnicas de caché inteligente, donde se trata de predecir los datos que serán accedidos en el futuro para almacenarlos en el caché. Las estrategias para determinar cuáles datos deben estar en el caché constituyen un problema muy interesante en las ciencias de la computación [22].

3.1 CARACTERÍSTICAS

Un caché se puede distinguir por sus componentes los cuales determinan el comportamiento e influyen directamente en su rendimiento [13]. Estos componentes corresponde a la gestión de caché, la política de reemplazo y la estrategia de resolución.

La gestión de caché se encarga del direccionamiento de los elementos que se almacenan en el caché. También maneja la búsqueda cuando se necesita extraer algún elemento del caché. La política de reemplazo elige los elementos que serán desalojados del caché cuando no hay suficiente espacio para insertar un nuevo elemento. La estrategia de resolución

determina la acción a seguir cuando hay errores de caché producido por objetos que han sido desalojados [13].

3.1.1 Gestión de caché

La gestión de caché se encarga de las políticas de ubicación o direccionamiento de los elementos que se almacenan en el caché, es decir, especifica la forma de acceder a los datos. Además, establece las políticas de escritura y de búsqueda [13].

Las políticas de escritura determinan la forma como se maneja la escritura de objetos que actualmente están almacenados en caché con respecto a la fuente original. Existen dos políticas principales:

Write through o escritura sincrónica. El objetivo es mantener la coherencia entre el caché y la fuente original escribiendo en el caché y luego, en un corto período de tiempo en la fuente original. La principal desventaja es el bajo desempeño con respecto a la escritura *Write back* [22].

Write Back o escritura asincrónica. El objetivo es aplazar la escritura en la fuente original lo más que se pueda para no degradar el desempeño. Si el objeto que se encuentra en el caché ha sido actualizado, se marca para que cuando sea desalojado del caché se actualice la fuente original [22].

La búsqueda es el procedimiento que se realiza cuando se necesita extraer algún elemento del caché. Puede ser secuencial, binaria, basada en árboles ordenados o con técnicas de *hashing*. Establecer el método de búsqueda adecuado depende del tamaño del caché y del tiempo de ejecución que se desee manejar [13].

El método de búsqueda debe estar de acuerdo con el tamaño del caché. Así, la búsqueda secuencial puede ser efectiva cuando el caché es pequeño [13].

Para cachés de mayor tamaño, se recomienda el uso de estructuras de datos más sofisticadas que puedan ser más eficientes, como es el caso de listas enlazadas, árboles ordenados o técnicas de *hashing* [13].

3.1.2 Política de reemplazo

La política de reemplazo elige los objetos que serán desalojados del caché cuando no hay espacio suficiente para insertar un objeto nuevo. Se pueden usar técnicas basadas en tiempo, en frecuencia o en el tamaño de los objetos que están almacenados en el caché o que se quieren guardar allí. Las basadas en tiempo eliminan el ítem más reciente o más antiguo en el caché para dar paso al nuevo elemento que se va a insertar en el caché. Con base en la frecuencia de acceso, se puede eliminar el elemento que tenga menor cantidad de uso. Si se tiene en cuenta el tamaño de los objetos, se puede desalojar del caché el elemento

más grande o el más pequeño. Otra técnica es la aleatoria donde se elimina cualquier elemento al azar [3] y [13].

Las políticas de reemplazo más usadas son [3]:

- ***Aleatoria*:** Se elimina cualquier elemento.
- ***LRU*:** *Least Recently Used*. El elemento que se desaloja del caché es el menos recientemente usado.
- ***LFU*:** *Least Frecuently Used*. Con esta política de reemplazo se elimina del caché el elemento menos frecuentemente usado.
- ***LFRU*:** Combina LRU y LFU.
- ***Más grande*:** Se desaloja del caché el objeto de mayor tamaño.
- ***Más pequeño*:** Se desaloja del caché el objeto de menor tamaño.
- ***FIFO*:** *First In First Out*. El elemento que se elimina del caché es el primer elemento que ha ingresado, es decir, el que lleva más tiempo en el caché.
- ***LIFO*:** *Last In First Out*. El elemento que se elimina del caché es el último elemento que ha ingresado, es decir, el que lleva menos tiempo en el caché.
- ***GreedyDual-Size* (*GD-S*):** Se mantiene un peso para cada archivo en el caché. Después de una inserción o uso (*cache hit*), el peso H_d asociado con el archivo d es calculado como $c(d)/s(d)$, donde $c(d)$ representa un costo asociado con d, por ejemplo el tiempo que toma llevar al objeto al caché y $s(d)$ es el tamaño del archivo d. Cuando se necesita reemplazar un archivo, el archivo v es desalojado cuyo H_v es el menor entre todos los H_v del caché [29].

3.1.3 Estrategia de resolución

La gestión de resolución determina la acción a seguir cuando hay fallos de caché producidos por objetos que han sido desalojados. Un *cache miss* se resuelve buscando el objeto, bien sea en un caché de nivel superior (si se trata de un caché jerárquico) o en la fuente original [13].

Algunas políticas de resolución son las siguientes:

- ***Storage*:** Si el objeto solicitado no se encuentra en el caché, se busca directamente en el repositorio original de datos.
- ***Parent*:** Si el objeto solicitado no se encuentra en el caché, se busca en el caché de nivel superior.
- ***Sibling*:** Si el objeto solicitado no se encuentra en el caché, se busca en el caché de un nodo que se encuentre al mismo nivel.

3.2 PROTOCOLOS DE COMUNICACIÓN ENTRE CACHÉS

Existen diferentes protocolos utilizados para la comunicación entre servidores *proxy*. Los más destacados son *ICP*, *HTCP*, *CARP*, *CRISP*, y *Cache Digest*. A continuación se describen brevemente.

3.2.1 *ICP - Internet Cache Protocol*

Fue creado dentro del proyecto *Harvest*, un proyecto conjunto de la *USC* - *University of Southern California* (*www.usc.edu*) y *University of Colorado* (*www.colorado.edu*) a mediados de los años *90*. *ICP* describe el formato de los mensajes que son utilizados para el intercambio de información entre cachés con el fin de localizar objetos en los cachés vecinos. Cuando se requiere localizar un objeto, un caché envía una solicitud *ICP* a sus vecinos quienes contestan indicando si el objeto está o no en el caché [23]. Una vez obtenida una respuesta para la solicitud, *ICP* ignora las demás [13]. *ICP* es liviano y viaja sobre *UDP* [23]. Con base en el tiempo que tarda una solicitud en ser satisfecha y la cantidad de paquetes perdidos, se puede determinar cuáles de los enlaces con los vecinos están congestionados lo que permite una forma de balance de carga [23].

3.2.2 *HTCP - Hyper Text Caching Protocol*

Es una variación del protocolo *ICP* donde sólo se envían las actualizaciones, esto permite reducir la cantidad de información que debe ser enviada por la red [13].

3.2.3 *CARP - Cache Array Routing Protocol*

Divide el espacio de direcciones (*URL*) en un arreglo de servidores *proxy*, con base en una función de *hashing* para proporcionar de forma determinista un camino para resolver una solicitud. La función de *hashing* se aplica sobre la identidad de los miembros del arreglo de *proxy* y los *URL*s, lo que significa que para un *URL* dado, el navegador sabrá exactamente dónde se encuentra almacenado, ya sea en caché debido a una solicitud anterior, o haciendo primero la consulta en la fuente original para posteriormente guardarla en caché. Todos los servidores *proxy* son mantenidos en una lista con los miembros del arreglo, la cual se actualiza de forma automática a través de una función del tiempo de vida que regularmente verifica los servidores *proxy* activos.

Gracias al uso de las funciones de *hashing*, no es necesario guardar demasiados datos para localizar la información almacenada en caché. El navegador solo debe ejecutar la misma función al objeto para determinar donde encontrarlo. Los objetos son distribuidos estadísticamente lo que permite cierto balance de carga [22].

3.2.4 *CRISP - Caching and Replication for Internet Service Performance*

Es un *proxy* caché escalable desarrollado bajo el liderazgo de *AT&T Labs Research* (*www.research.att.com*). Los cachés *CRISP* están estructurados como una colección de

servidores *proxy* autónomos que comparten su contenido de caché a través de un servicio de mapeo en un directorio centralizado. *CRISP* permite que un grupo de cachés compartan un directorio común que lista los documentos almacenados en caché en todo el grupo. *CRISP* administra el directorio de forma centralizada. En caso de fallo al buscar un objeto, *CRISP* revisa el directorio común. Si un compañero lo tiene, le traspasan la solicitud, sino se consulta al servidor original [23].

Si el objeto solicitado está almacenado en el caché del cliente, el objeto es entregado sin acceder su fuente original. Para probar el caché distribuido, el *proxy* reenvía el *URL* solicitado a un servidor de mapeo. El servicio de mapeo mantiene un directorio de todo el caché. Los *proxies* notifican al servicio de mapeo en cualquier tiempo que ellos adicionan o remueven un objeto del caché. Las actualizaciones y las pruebas son hechas intercambiando mensajes *unicast*. Si el mapeo en el servidor central indica que un objeto solicitado está en un *peer*, el *proxy* solicitante recupera el objeto directamente del *peer* y lo retorna al cliente [25].

3.2.5 *Cache Digest*

Utiliza una técnica basada en catálogo. Cada caché maneja un catálogo con el contenido de los cachés hermanos. Los catálogos son sincronizados periódicamente. La sincronización de los catálogos implica el envío de mensajes en la red y el procesamiento de esos mensajes. El proceso de sincronización es costoso, por lo tanto en ciertos casos es preferible tener algunas inconsistencias en el catálogo y no actualizar. Cuando ocurre un error, el caché busca en su catálogo para saber los cachés hermanos donde está posiblemente el objeto y enviarles un mensaje de consulta. La primera respuesta obtenida se procesa y el resto se ignoran. Si de acuerdo con el catálogo no hay ningún caché que tenga los datos o si la respuesta de los hermanos es *null*, los padres o las fuentes originales son contactadas. El *Cache Digest* es relevante solo si las modificaciones se hacen raramente [13].

Cada caché proporciona catálogo del contenido de su caché a los *proxies* participantes en su grupo. El *digest* es utilizado para identificar algún caché cercano que probablemente pueda tener un objeto solicitado. *Cache Digest* emplea dos protocolos, uno para construir e interpretar un *digest* y otro para intercambiar *digests* entre los diferentes cachés.

3.3 VENTAJAS Y DESVENTAJAS DEL CACHÉ

Utilizar un caché tiene ventajas y desventajas que deben ser tenidas en cuenta antes de ponerlo en operación.

Las principales ventajas son: la reducción de la latencia experimentada por los usuarios a la espera de un objeto; la reducción del tráfico de la red y por lo tanto el consumo de ancho de banda; se aumenta la robustez del sistema gracias a la posibilidad de tener datos replicados; es un mecanismo que puede ayudar en la escalabilidad de un sistema distribuido y ayuda a

reducir cuellos de botella en servidores que son accedidos con mucha frecuencia al evitar puntos únicos de fallo [25].

En el caché no sólo existen aspectos positivos. Entre los aspectos negativos o desventajas significativas se tienen: la falta de actualización de algunos objetos almacenados en caché lo que puede conducir a la propagación de copias obsoletas; el aumento de la latencia cuando se presenta un fallo de caché (*cache miss*), por el trabajo adicional que se debe realizar; y la falta de precisión en las estadísticas de acceso de los servidores de origen porque la distribución de los objetos almacenados en caché no son contabilizados.

4 CACHÉ DISTRIBUIDO

El trabajo cooperativo entre cachés es una importante mejora en el desempeño con respecto a los cachés que trabajan de forma aislada, porque la cantidad de clientes que se conectan a un único caché es pequeña comparada con la creciente cantidad de información disponible en las aplicaciones distribuidas. El *hit rate* puede ser incrementado significativamente, a través de información compartida por usuarios que tengan intereses similares [27].

En la siguiente sección se presenta una descripción de las arquitecturas de caché distribuido más utilizadas.

4.1 ARQUITECTURAS DE CACHÉ DISTRIBUIDO

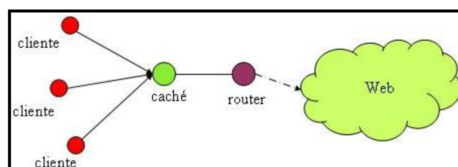
Los dos enfoques básicos utilizados en la implementación de un caché distribuido de gran escala son el esquema jerárquico y el esquema distribuido. Sin embargo, existen otras propuestas, de las cuales algunas se mencionan a continuación.

4.1.1 *Proxy* caché autónomo

Un servidor *proxy* recibe solicitudes *http* de los clientes por un objeto y si lo tiene almacenado en caché, lo retorna al cliente que lo solicitó, sin afectar el tráfico hacia el servidor de destino en la red. En caso contrario, el *proxy* intenta traerlo directamente del servidor de origen. Las ventajas del *proxy* caché son la disminución en el ancho de banda consumido y en la latencia experimentada por el cliente. Como se puede apreciar en la figura *5*, un *proxy* caché generalmente se ubica en la puerta de salida hacia Internet [34].

El *proxy* caché se convierte en un único punto de fallo, esto corresponde a una de sus principales inconvenientes. Además, los clientes deben ser configurados manualmente, lo que significa que si un *proxy* caché llega a fallar, los navegadores de los usuarios deben ser reconfigurados manualmente y normalmente en forma individual [34].

Figura 5. Proxy caché autónomo



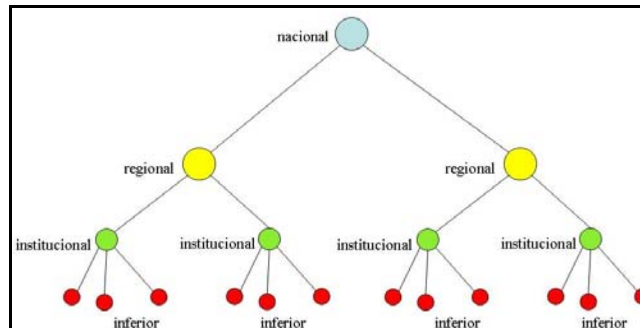
4.1.2 *Proxy* Caché Sustituto

También llamado *surrogate*. Consiste en ubicar el *proxy* cerca de los servidores, en lugar de ubicarlos cerca de los clientes [34]. Tiene delegada la autoridad en nombre del servidor al cual representa. Se utiliza para acelerar servidores Web lentos. Un caché sustituto también se puede utilizar para descifrar la información que viaja cifrada sobre conexiones *http/tls* porque es preferible ocupar al *surrogate* en estas actividades y no al servidor origen. Un *surrogate* usualmente recibe solicitudes para un número muy pequeño de servidores por lo que el *hit rate* es alto, en muchos casos, cercanos al 90% [25].

4.1.3 Caché jerárquico

Consiste en conformar una jerarquía ubicando los cachés en los diferentes niveles. En el nivel más bajo de la jerarquía se encuentra el caché de los clientes. Si una solicitud no se puede resolver en el caché del cliente, la solicitud se reenvía al caché del siguiente nivel. Si al terminar la jerarquía no se ha podido resolver, la solicitud entonces es enviada al servidor original [20]. En [20] se hace referencia a un sistema de caché jerárquico con cuatro niveles, basado en el proyecto *Harvest*. Como se puede ver en la figura *6*, la jerarquía propuesta se compone de un nivel inferior donde se ubica el caché del cliente, un nivel institucional, un nivel regional y un nivel nacional.

Figura 6. Arquitectura jerárquica

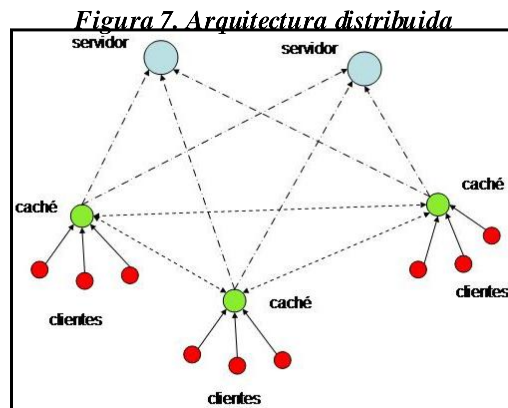


Cuando se solicita un objeto, si no puede ser satisfecha por el caché del cliente en el nivel inferior, la solicitud es reenviada al caché de nivel institucional. Si el objeto no está en el caché de nivel institucional, la solicitud se envía al caché del nivel regional y si el objeto no se encuentra en el nivel regional, la petición va hasta el caché de nivel nacional. Si en el caché de nivel nacional no se puede resolver la solicitud, se contacta al servidor de origen. En el momento en que se encuentre el objeto solicitado, la respuesta inicia el camino en orden inverso por la jerarquía, dejando una copia en el caché de cada nivel.

4.1.4 Caché distribuido o en malla

En el caché distribuido, todos los cachés son configurados para que estén al mismo nivel, sin necesidad de tener cachés de nivel intermedio, tal como aparece en la figura *7*. Para determinar cuál es el caché de donde será obtenido un documento, los cachés comparten información de metadatos acerca del contenido de los otros cachés.

Figura 7. Arquitectura distribuida



Con el caché distribuido la mayor parte del tráfico fluye en los niveles bajos de la red, los cuales son menos congestionados y no requieren espacio de disco adicional en los niveles intermedios de la red. Además, el caché distribuido permite un mejor balance de carga y mejor tolerancia a fallos. Sin embargo, un sistema de caché distribuido a gran escala puede encontrar varios problemas tales como alto tiempo de conexión, alto uso de ancho de banda y problemas de administración [20].

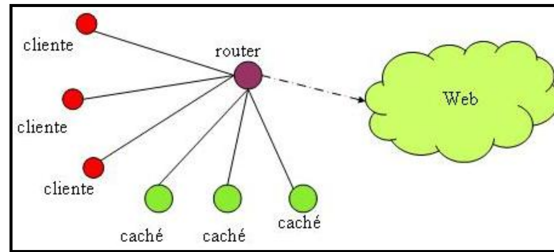
4.1.5 Caché híbrido

En un esquema híbrido, los cachés pueden cooperar en el mismo nivel o en un nivel más alto usando caché distribuido. *ICP* es un ejemplo típico. El documento es traído de un caché padre/vecino que tiene un bajo *RTT*. En algunos casos, puede ser mejor contactar al servidor original que esté más cerca que un caché vecino [20]. Esto mejora la eficiencia y el rendimiento.

4.1.6 Caché transparente

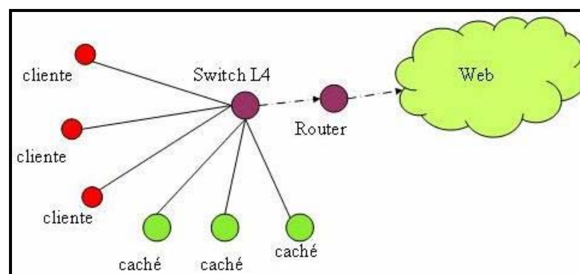
Con el fin de superar el inconveniente generado por la configuración manual de los navegadores de los clientes, la arquitectura de caché transparente trabaja interceptando las solicitudes *http* de los clientes y las redirige a un cluster de servidores *proxy* caché. Existen dos técnicas, una usando un *router* y otra usando un *switch* de nivel *4*, tal como se ve en la figuras *8* y *9*. El caché transparente usa enrutamiento basado en políticas para hacer las solicitudes al caché apropiado [25].

Figura 8. Configuración caché router transparente



En el caché transparente basado en *switching* de capa *4*, el *proxy* actúa como un balanceador de carga dedicado. En esta técnica se reduce la sobrecarga (*overhead*) que normalmente ocurren en el caché transparente basado en *router* [25].

Figura 9. Configuración caché switch transparente



4.1.7 Caché cooperativo

En algunos servidores Web el cuello de botella no se encuentra en la red sino en el tiempo de utilización del procesador. Esto ocurre especialmente en sitios con mucha utilización de solicitudes de contenido dinámico. Una solución para este problema, presentado en [15] consiste en crear un servidor Web distribuido llamado *Swala*, en el cual se utiliza un caché cooperativo para almacenar los resultados de solicitudes CGI. [15].

Swala es un servidor Web *multithread* que corre sobre un *cluster* de estaciones de trabajo, las cuales comparten información de los datos almacenados en caché e intercambian información sobre el contenido del caché de cada nodo. El servidor guarda los resultados de la ejecución de programas *CGI* e información (metadatos) acerca de los datos almacenados en el directorio de caché. Cada nodo tiene dos componentes, un módulo *http* y un módulo de caché [25].

El módulo de caché mantiene el directorio local de caché, el cual contiene información acerca de las entradas de los cachés de cada nodo. Tiene tres *threads*, uno pendiente de actualizar el directorio local, un segundo *thread* escuchando solicitudes de los otros nodos, y un tercer *thread* encargado de eliminar las entradas que están vencidas [15].

4.1.8 Caché adaptativo

El caché adaptativo está formado por grupos dinámicos de cachés. La arquitectura general se compone de múltiples servidores de caché que se auto-organizan formando mallas en grupos *multicast* traslapados que se adaptan automáticamente a medida que las condiciones van cambiando. Estas mallas forman una jerarquía implícita que puede ser usada para difundir contenido popular de acuerdo a la demanda [25].

El caché adaptativo usa dos protocolos *Cache Group Management Protocol* (*CGMP*) que especifica la formación de mallas y el *Content Routing Protocol* (*CRP*) utilizado para localizar contenido almacenado en caché dentro de las mallas existentes [25].

5. *DHTCaché*

Como se mencionó antes, un desarrollador de aplicaciones *DHT* puede usar cachés para mejorar el desempeño de una aplicación, y como la configuración adecuada de un caché es un proceso complejo, se plantea la construcción de un sistema *DHT* que permita realizar pruebas con diferentes estrategias de caché para obtener información que ayude al desarrollador de aplicaciones a establecer el esquema de caché que mejor se acomode a la aplicación.

A nivel conceptual, el sistema está pensado para ser desarrollado con cualquier sistema *DHT* en forma genérica lo que quiere decir que el desarrollador plantea su aplicación en términos de las primitivas de servicio *get* y *put*, las cuales deben ser ofrecidas por la capa de almacenamiento de un sistema *DHT*. El sistema lo que hace es ponerse en medio de la aplicación construida por el desarrollador y el sistema *DHT* subyacente para crear un caché, tanto en la capa de búsqueda como en la capa de almacenamiento, de manera transparente para la aplicación, al igual que el monitoreo de una serie de eventos que se registran en archivos *log* para su análisis posterior. En otras palabras, la aplicación no tiene porqué modificar su comportamiento ni tener conocimiento si el caché está habilitado o no.

Se espera que el desarrollador pueda probar diferentes configuraciones de caché y use las métricas que brinda el sistema para determinar cuál es la configuración de caché que más le favorece al desempeño de su aplicación. El desarrollador puede extender su análisis aprovechando los archivos *log* para realizar cálculos adicionales. Sin embargo, algunas aplicaciones pueden necesitar el cálculo de métricas diferentes que no se pueden obtener con la información registrada en los *logs*, por lo que el sistema no puede dar soporte a todo tipo de aplicaciones.

5.1. ANÁLISIS DE REQUERIMIENTOS

El sistema ofrece fundamentalmente dos servicios a las aplicaciones *DHT* (*get* y *put*), los cuales son usados por el desarrollador para la creación de sus aplicaciones. Además, con el fin de realizar pruebas de diferentes configuraciones de caché, el desarrollador puede configurar los cachés en las dos capas inferiores del sistema *DHT* en cada nodo y después de realizar las pruebas, analizar los registros que son almacenados en archivos *log* para sacar sus propias conclusiones. Los requerimientos funcionales y no funcionales del sistema.

5.1.1 Requerimientos funcionales

Existen tres componentes que conforman el sistema: el caché, el sistema *DHT* y el analizador de archivos *log*. El caché es el componente que permite crear, configurar y

utilizar cachés en los diferentes nodos que conforman la red *overlay* subyacente al sistema *DHT*; el sistema *DHT* es el componente que ofrece los servicios a las aplicaciones *DHT* y permite almacenar y recuperar objetos; y el analizador de archivos *log* es el componente que examina línea por línea los archivos *log* para consolidar la información relacionada y realizar los cálculos de las métricas. A continuación, se presentan los requerimientos funcionales.

Identificador del caso de uso: CU-01.

Nombre del caso de uso: Crear caché.

Actor: Sistema *DHT*.

Descripción: Este caso de uso permite al sistema *DHT* crear un caché. El caché puede ser creado tanto en su capa búsqueda como en la capa almacenamiento.

Precondiciones: Ninguna.

Poscondiciones: El caché queda disponible para ser configurado.

Curso básico de eventos: El sistema *DHT* crea el caché.

Identificador del caso de uso: CU-02.

Nombre del caso de uso: Configurar caché.

Actor: Sistema *DHT*.

Descripción: Este caso de uso permite al sistema *DHT* configurar un caché especificando sus propiedades.

Precondiciones: Debe tener el archivo donde se especifican las propiedades, almacenado en el directorio raíz del proyecto. En caso que no exista ese archivo, el caché será configurado por defecto.

Poscondiciones: El caché queda disponible para insertar elementos en él o para recuperarlos.

Curso básico de eventos: El sistema *DHT* crea el caché. El constructor del caché busca el archivo y obtiene las propiedades tamaño del caché, tamaño del parámetro de limpieza automática, umbral de limpieza automática, política de reemplazo y nombre del archivo de propiedades del *logger* que se va a utilizar para registrar los eventos del caché. El caché queda listo para ser utilizado.

Identificador del caso de uso: CU-03.

Nombre del caso de uso: *get* - Obtener un objeto del sistema *DHT*.

Actor: Aplicación *DHT*.

Descripción: Este caso de uso permite a las aplicaciones *DHT* obtener del sistema un objeto especificando su identificador.

Precondiciones: La aplicación tiene el identificador del objeto que desea solicitar.

Poscondiciones: Si el objeto está almacenado en el sistema *DHT*, es entregado a la aplicación.

Curso básico de eventos: La aplicación *DHT* da el identificador del objeto que busca en el sistema *DHT*. El sistema *DHT* entrega el objeto solicitado.

Curso alterno: Si el objeto solicitado está almacenado en el caché es entregado. Si no está, el objeto es buscado en el sistema *DHT* en forma tradicional, sin cambios.

Identificador del caso de uso: CU-04.

Nombre del caso de uso: *put* - Insertar un objeto en el sistema *DHT*

Actor: Aplicación *DHT*.

Descripción: Este caso de uso permite a las aplicaciones *DHT* insertar un objeto en el sistema *DHT*.

Precondiciones: La aplicación tiene listo el objeto que desea insertar y el identificador del objeto.

Poscondiciones: El objeto queda almacenado en el sistema *DHT*.

Curso básico de eventos: La aplicación *DHT* da el objeto y el identificador del objeto al sistema *DHT*. El sistema *DHT* inserta en la capa *DSS* el objeto en el nodo correspondiente.

Identificador del caso de uso: CU-05.

Nombre del caso de uso: Obtener métricas.

Actor: Desarrollador de aplicaciones *DHT*.

Descripción: Este caso de uso permite al desarrollador consolidar en una tabla todos los eventos registrados sobre el uso del caché y el uso del sistema *DHT*. El analizador de archivos *log* produce una tabla a partir de la cual el desarrollador puede obtener las métricas que son de su interés.

Precondiciones: Deben existir dos archivos *log* en el directorio raíz del proyecto, el del caché y el del sistema *DHT*.

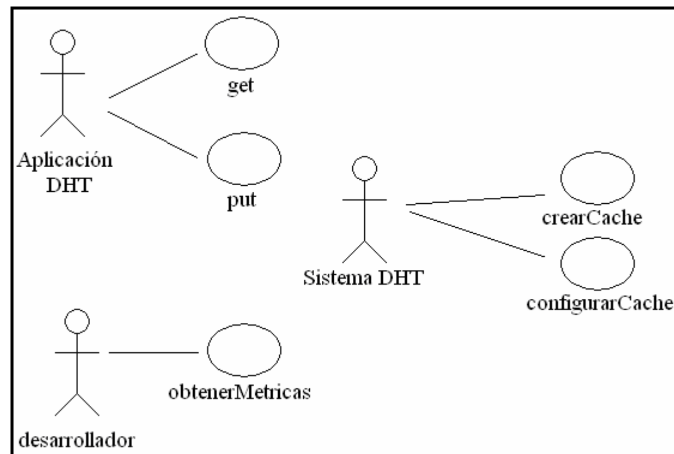
Poscondiciones: Un archivo de resultados queda almacenado en el sistema *DHT*.

5.1.2 Requerimientos no funcionales

El único requerimiento no funcional considerado es la transparencia para la normal operación de aplicaciones *DHT*. El desarrollador puede poner en funcionamiento una aplicación *DHT* escrita en términos de las primitivas de servicio *get* y *put*.

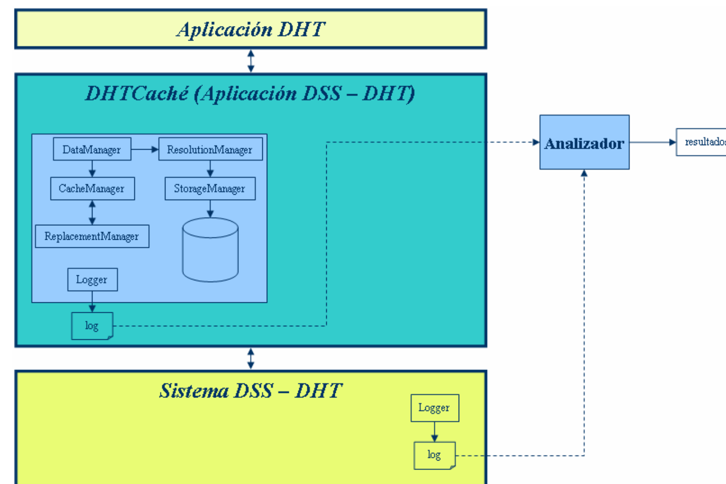
5.2. DIAGRAMA DE CASOS DE USO

Figura 10. Diagrama de Casos de Uso



5.3. ARQUITECTURA DEL SISTEMA

Figura 11. Arquitectura del sistema



DHTCaché es una aplicación *DSS - DHT* que proyecta los servicios de la capa *DSS* del sistema *DHT* subyacente para permitir la ejecución de aplicaciones *DHT* en forma transparente. La aplicación *DHT* es externa al sistema y se espera que funcione sobre *DHTCaché* sin ninguna modificación.

DHTCaché intercepta las operaciones *get* y *put* de que la aplicación *DHT* hace sobre la capa *DSS* del sistema *DHT*. Cuando se hace uso de la operación *put*, *DHTCaché* atrapa el objeto que se desea insertar en la capa *DSS*, la inserta en el caché del nodo de acceso y luego continúa el proceso habitual de la capa *DSS*. Si la operación que la aplicación *DHT* desea usar es *get*, *DHTCaché* atrapa la solicitud y busca en el caché. Si el objeto solicitado se encuentra en el caché, el objeto es retornado. Si el objeto no está en el caché, es solicitado a la capa *DSS* como si *DHTCaché* no existiera.

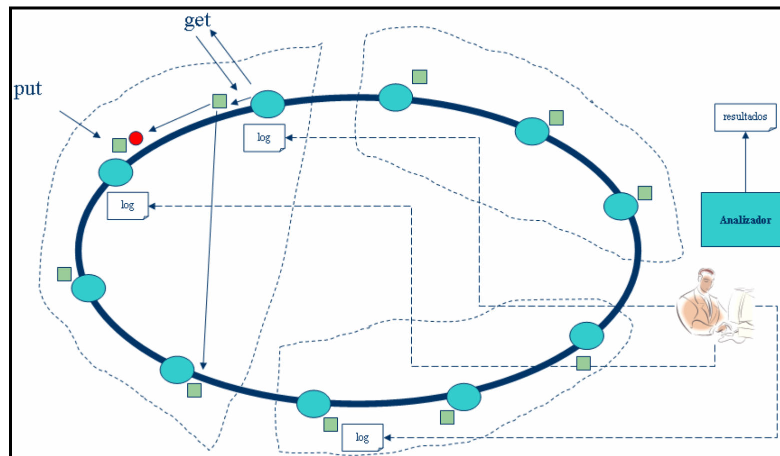
El caché es el componente que presta el servicio de caché dentro de *DHTCaché*. El caché del sistema *DHTCaché* permite el uso de cachés distribuidos que pueden trabajar en forma cooperativa para ayudar a resolver fallas de caché.

Todas las solicitudes de los servicios *put* y *get* que hace la aplicación *DHT* son registrados en archivos *log*, al igual que las utilizaciones del caché. Además se construyó un componente que analiza los registros almacenados en los archivos *log* examinando línea por línea para consolidar la información y obtener resultados que pueden ser usados por el desarrollador de aplicaciones *DHT* para apoyar la toma de decisiones sobre una configuración de un caché para su aplicación.

El *Sistema DSS – DHT* corresponde a la capa *DSS* del sistema *DHT* subyacente que permite la ejecución de aplicaciones *DHT* y ofrece servicios *get* y *put*.

5.4 DIAGRAMA FUNCIONAL DEL SISTEMA

Figura 12. Diagrama funcional de DHTCaché



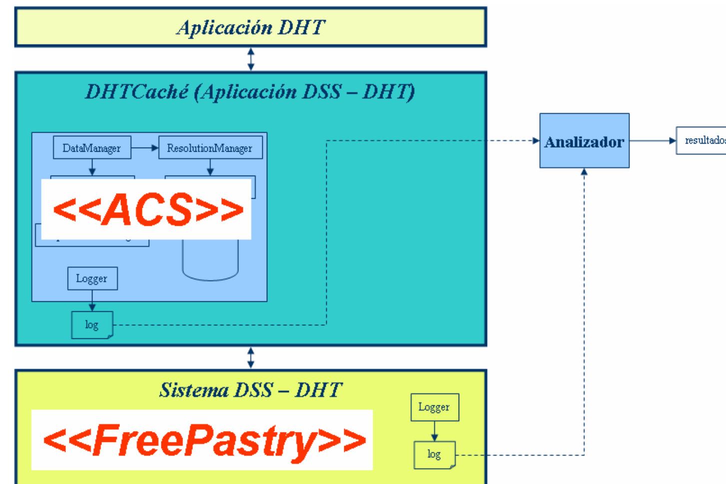
En la figura *12* se pueden observar los nodos y el caché en cada uno de ellos. Por otra parte, se pueden identificar grupos de cachés distribuidos que pueden resolver fallas de caché entre ellos. La operación *put* que hace una aplicación *DHT* sobre un nodo en *DHTCaché* hace que el objeto sea almacenado en el caché de ese nodo y luego entregado al sistema *DHT* subyacente para que lo almacene en el nodo o los nodos que le corresponda de acuerdo con el algoritmo utilizado en la capa *DSS*. En el caso de la operación *get*, un objeto solicitado primero se busca en el caché del nodo que es accedido. Si el objeto no se encuentra en el caché, entra en funcionamiento el protocolo de una resolución de fallas de caché entre los cachés que conforman un grupo. Si el objeto no es encontrado en el caché, es buscado en la capa *DSS* del sistema *DHT*.

Al terminar la ejecución de la aplicación *DHT*, se crear unos archivos *log* que almacenan los registros de los eventos ocurridos en el sistema *DHT* y en el caché de *DHTCaché*. Estos archivos *log* quedan dispersos por los diferentes nodos que conforman la red *overlay* y es el desarrollador de aplicaciones quien debe recogerlos y pasarlos por el componente analizador para obtener los resultados.

5.5. ALCANCE DE LA IMPLEMENTACIÓN

Para la realización del prototipo se utilizó *ACS*, un *framework* para la construcción de servicios adaptables de caché [13], y *FreePastry,* una plataforma para la prueba y desarrollo de aplicaciones *DHT* [38], tal como se puede apreciar en la figura *13*.

Figura 13. Implementación de un prototipo



Con respecto a los requerimientos funcionales, el alcance de la implementación de la solución es el siguiente:

- ***get* y *put***

Estas dos operaciones fueron realizadas satisfactoriamente mediante modificaciones a *FreePastry* en los métodos *lookup* e *insert* en la clase *PastImpl* del paquete *rice.p2p.past*. En el método *insert* el cambio consiste en adicionar al caché *ACS* que se ha creado en *Past*, el objeto que se desea insertar en el sistema *DHT*. De igual manera, el método *lookup* busca el objeto solicitado en el caché *ACS* de *Past*. Si lo encuentra lo retorna, si no está, lo busca en *Past* en la forma habitual. Es de anotar que el caché se ubica en cada nodo, en forma opcional. Una propiedad cache = true determina si un nodo en *FreePastry* tiene un caché *ACS* o no.

- **Crear caché**

Se construyó la clase *ObjectCache* en el paquete *cache* para permitir la creación de un caché *ACS* para ser usado dentro de *Past*. Esta es una clase nueva que se adicionó al *framework ACS*.

El tipo de caché que se ofrece en el sistema es el caché de objetos, porque es el caché que viene incluido en *ACS*. Es importante anotar que *ACS* hace una aproximación a un caché de memoria, pero no está documentado su uso completamente.

Las políticas de reemplazo fueron limitadas a *FIFO* y *LRU*, porque son las dos políticas de reemplazo suministradas por *ACS*.

- **Configurar Caché**

La configuración de un caché permite establecer las características que el desarrollador desea probar. Las propiedades que se han considerado para configurar un caché son el tamaño del caché, el parámetro de limpieza automática, el umbral de limpieza automática, la política de reemplazo y el nombre del archivo que especifica las propiedades del *logger* del caché. Se utiliza un archivo de propiedades llamado *cache.properties* para facilitar su utilización. Si se ejecuta el sistema en un ambiente de red, el archivo de propiedades debe estar en cada nodo y de esta manera, los parámetros de los cachés de los diferentes nodos pueden ser diferentes.

A continuación, un ejemplo del contenido del archivo de propiedades para el caché.

```
maxObjects = 20
autoCleanSize = 2
autoCleanThreshold = 20
replacement = lru
cache = true
propertiesLogFileName = log.properties
```

La propiedad *maxObjects* es el tamaño del caché en número de objetos; *autoCleanSize* es el número de elementos que se pueden borrar automáticamente del caché (en *background*) cuando es alcanzado el umbral; *autoCleanThreshold* es la propiedad que establece el umbral para activar la limpieza automática del caché. Estas tres propiedades son propias de *Perseus* el *framework* en el que se basó *ACS* y son utilizadas en este trabajo sin modificación alguna. La propiedad *replacement* establece la política de reemplazo que determina el objeto que debe ser desalojado del caché cuando no hay espacio suficiente para insertar un nuevo objeto y *propertiesLogFileName* establece el nombre del archivo de configuración del *logger* que se utiliza para registrar los eventos ocurridos en el caché.

En caso que el archivo de propiedades no esté almacenado en el directorio raíz del proyecto, el caché será configurado con propiedades por omisión, entre las que se destacan el tamaño del caché (10 objetos) y la política de reemplazo *FIFO*.

- **Obtener métricas**

Para el análisis de la información contenida en los archivos *log*, se cuenta con un componente analizador llamado *stats*. Este análisis es realizado después de terminar la ejecución de la aplicación de pruebas, pero el procedimiento no es automático. La información arrojada por el sistema es un archivo de texto con los registros de los eventos ocurridos en el caché y otro archivo con los eventos ocurridos en el sistema *DSS - DHT*. Estos dos archivos son por cada nodo y sirven de entrada al sistema de análisis de los datos.

Este componente contabiliza los eventos asociados a cada uno de los identificadores de los objetos que son insertados en el sistema *DSS - DHT* durante la ejecución de las pruebas, en particular lo que tiene relación con el uso del caché.

En primer lugar, se recorre el archivo *log* del caché para formar una tabla *hash* con el identificador como llave y el objeto almacenado en la tabla está conformado por un vector con información sobre el tiempo en el que ocurre cada evento y su duración. Luego se analiza el archivo *log* que ha sido creado por *FreePastry* para almacenar los eventos en la misma tabla *hash*.

llave	objeto		
id	evento	tiempo	duración
	evento	tiempo	duración
	evento	tiempo	duración

Después de tener la tabla *hash* con todos los eventos guardados, se hace un recorrido por toda la tabla para formar una nueva tabla *hash* donde la llave de nuevo es el identificador del objeto y el objeto almacenado en la tabla es otra tabla *hash* con el evento como llave y unas estadísticas formadas por el tiempo promedio y la frecuencia como objeto.

llave	objeto		
	llave	objeto	
id	evento	tiempoPromedio	frecuencia

La salida del componente analizador de los archivos *logs* es una tabla que resume el tiempo promedio y la frecuencia por cada evento ocurrido en toda la prueba. Sin embargo, es posible obtener más información de la tabla *hash* que tiene contenidos todos eventos ocurridos durante la prueba.

El componente analizador de archivos *log* entrega un archivo de resultados que resume el desempeño de una aplicación *DHT* y le sirve al desarrollador para establecer las comparaciones entre las diferentes configuraciones de caché para una aplicación particular.

5.5.1 Otros logros

- **Resolución de fallas de caché**

El método *load* es utilizado en *ACS* para traer un objeto bien sea del caché o de alguno de los sitios que están configurados en un caché para resolver las fallas de caché. Estos sitios pueden ser un almacén de datos local u otro caché.

Se realizaron algunas modificaciones con el fin de aumentar la funcionalidad especialmente en lo que tiene que ver con las arquitecturas de caché distribuido, donde *ACS* solo daba soporte parcial puesto que permite un solo manejador de fallas de caché, el cual podía ser *StorageResolutionManager* o *ParentResolutionManager*. Si se desea construir una arquitectura diferente, es necesario construir una clase nueva con la arquitectura fija y no permite aumentar el número de opciones para resolver las fallas de caché.

Para hacer posible que un caché tenga asociados uno o varios cachés que le apoyen en la resolución de fallas de caché, un caché tiene un vector de cachés a los cuales le solicita un objeto produjo una falla de caché y que tampoco fue encontrado en el almacén de datos local.

Es importante resaltar que pueden presentarse ciclos en el proceso de búsqueda en el caché distribuido. Esto es controlado mediante *flags* que impiden volver a buscar un mismo objeto dos veces en un mismo caché durante el mismo procedimiento de búsqueda.

Las arquitecturas de caché distribuido que fueron probadas se encuentran las siguientes:

- Caché local
- Caché distribuido de dos cachés con enlace unidireccional
- Caché distribuido de dos cachés con enlace bidireccional
- Caché jerárquico en dos niveles
- Caché distribuido plano en anillo

Las modificaciones realizadas a *ACS* permiten generar arquitecturas de cualquier topología a gusto del desarrollador. Sin embargo, en la implementación de *DHTCaché* se incluyó la arquitectura de caché local. Las otras arquitecturas de caché implementadas en *ACS* necesitan de la información de enrutamiento almacenada en la capa *DLS* del sistema *DHT* subyacente, a partir de la cual se puedan conformar los grupos de cachés que pueden colaborar en la resolución de fallas de caché.

- **Registro de la ocurrencia de eventos relacionados con el caché y el sistema *DHT* subyacente en archivos *log***

En este sistema se utilizan los *loggers* para permitir que una misma prueba pueda ser aplicada a diferentes configuraciones de caché y luego se puedan realizar comparaciones a partir de la información que se registra en los archivos *log*.

Un *logger* es un componente que le permite al desarrollador establecer categorías de mensajes que facilitan la depuración y control de las aplicaciones. Un *logger* controla la salida de mensajes, ya sea hacia la consola o hacia un archivo de texto; maneja diferente nivel de detalle dependiendo de la necesidad del desarrollador y tiene la ventaja que se pueden activar o desactivar a través de un archivo de configuración lo que facilita ponerse las aplicaciones en producción sin necesidad de modificar el código [21].

El *logger* de *ACS* depende del proyecto *ObjectWeb* [39] del que hace parte *Perseus*, el *framework* de caché predecesor de *ACS*. La configuración del *logger* de *ACS* se carga a partir de un archivo de propiedades. A continuación, un ejemplo de un archivo de configuración del *logger*.

```
log.config.classname org.objectweb.util.monolog.wrapper.log4j.MonologLoggerFactory

handler.fileHandler.type File
handler.fileHandler.appendMode false
handler.fileHandler.output cache.log
handler.fileHandler.pattern %d %m%n

logger.root.level INFO
logger.root.handler.0 fileHandler
```

En este caso, la salida se hará a un archivo llamado *cache.log* y un patrón con una estampilla de tiempo con el tiempo que ha transcurrido desde el inicio de la ejecución de la aplicación (`%r`), un mensaje (`%m`) y un salto de línea (`%n`). En el archivo *cache.log* saldrán mensajes hasta la categoría *INFO*.

La configuración del *logger* canaliza la salida hacia un archivo de texto usando el formato ya explicado. Los mensajes tienen un prefijo predeterminado (en este caso las iniciales del autor *"CEG")* los que permite diferenciar mensajes de la categoría *INFO* propios de ACS que no tengan relación con el trabajo realizado. Un fragmento de un archivo *log* es la siguiente:

```
2031 CEG_Bind E6B976CEE70D4720B4D2E2597259ACBA0340AFF1 52.520 ns
17500 CEG_Lookup_Hit A3A375DC45B077A1E03F6C8934AE861D8D38FD24 96.661 ns
18031 CEG_Lookup A3A375DC45B077A1E03F6C8934AE861D8D38FD24 27.378 ns
```

El primer campo corresponde a la estampilla de tiempo; el segundo el código del evento ocurrido; el tercero un identificador del objeto almacenado en el caché; y el campo final es el tiempo que tardó en completarse el evento, medido en *nanosegundos*. Se utilizó la unidad de medida en *nanosegundos* porque es la que mayor precisión otorgó y especialmente porque al realizar pruebas en una sola máquina, le tiempo de acceso a los datos es muy cercano a cero.

Los eventos que están siendo monitoreados son: *Bind*, cuando se inserta un objeto en el caché, *Lookup* cuando se busca algo en el caché y *LookupHit* cuando algo que se busca en el caché es encontrado.

Al igual que *ACS*, *FreePastry* también utiliza *logger* para manejar la salida de los mensajes, bien sea hacia la consola o hacia un archivo de texto. En el *log* de *FreePastry* se registran los eventos *insert* y *get* en el caché *ACS*. Cada nodo activo en el sistema genera un archivo *log*, con el siguiente formato:

```
20:25:43,064:CEG_Insert 64ED03406972573E6CAEBBBE157ECC7AE9517E4E 320.432 ns
20:26:13,064:CEG_Get A3A375DC45B077A1E03F6C8934AE861D8D38FD24 518.502 ns
20:26:13,064:CEG_Get_Hit A3A375DC45B077A1E03F6C8934AE861D8D38FD24 664.889 ns
```

En la primera columna se puede apreciar una estampilla de tiempo junto con el mensaje asociado al evento, luego el identificador del objeto y el tiempo en nanosegundos.

Los eventos que están siendo monitoreados son: *Insert*, cuando se inserta un objeto *Past*, *Get* cuando se solicita un objeto a *Past* y *GetHit* cuando una solicitud realizada es satisfecha.

5.6 DIAGRAMAS DE SECUENCIA

Un diagrama de secuencia permite ilustrar la interacción entre los diferentes objetos del sistema y muestra algunos detalles de la implementación. A continuación se presentan tres diagramas de secuencia, de los métodos *lookup* e *insert* en la clase *PastImpl* del paquete *rice.p2p.past* de *FreePastry* y *load* en la clase *BasicDataManager* del paquete *org.objectweb.perseus.data.lib* en el *framework ACS*.

En los métodos *insert* y *lookup*, los objetos *PastImpl* que implementan la capa de almacenamiento en *FreePastry* interactúan con el caché y con el *logger*.

Figura 14. Diagrama de secuencia de la operación insert

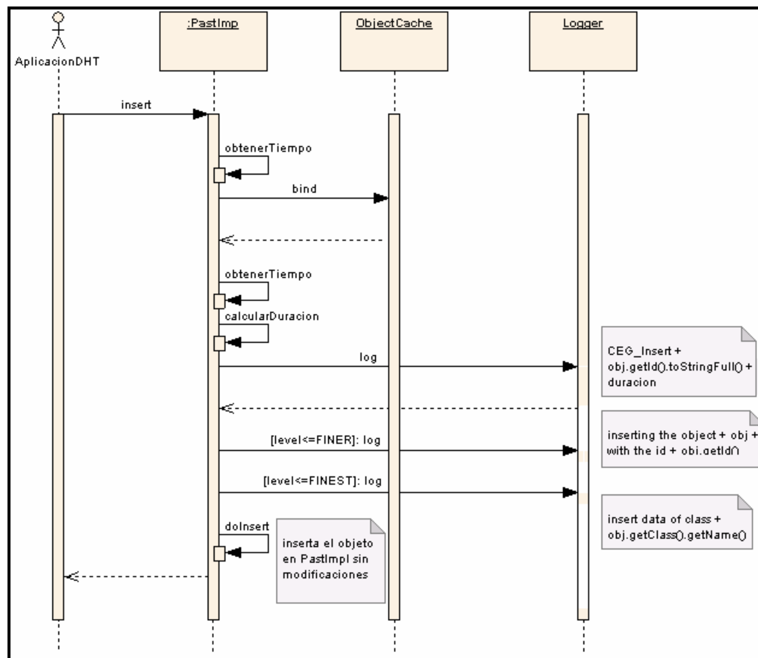


Figura 15. Diagrama de secuencia de la operación lookup

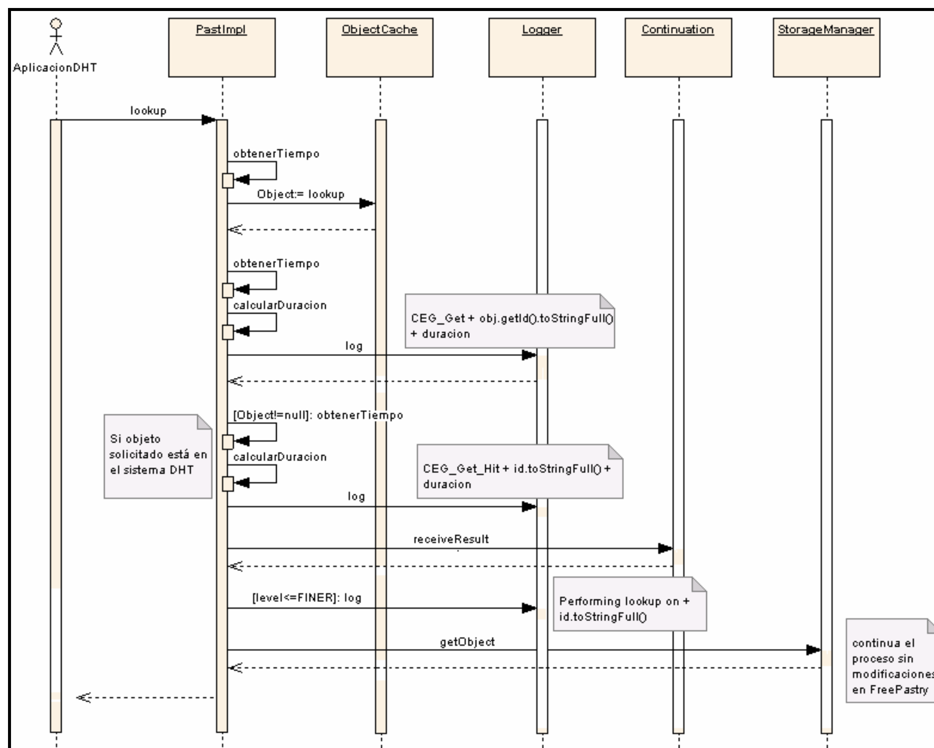
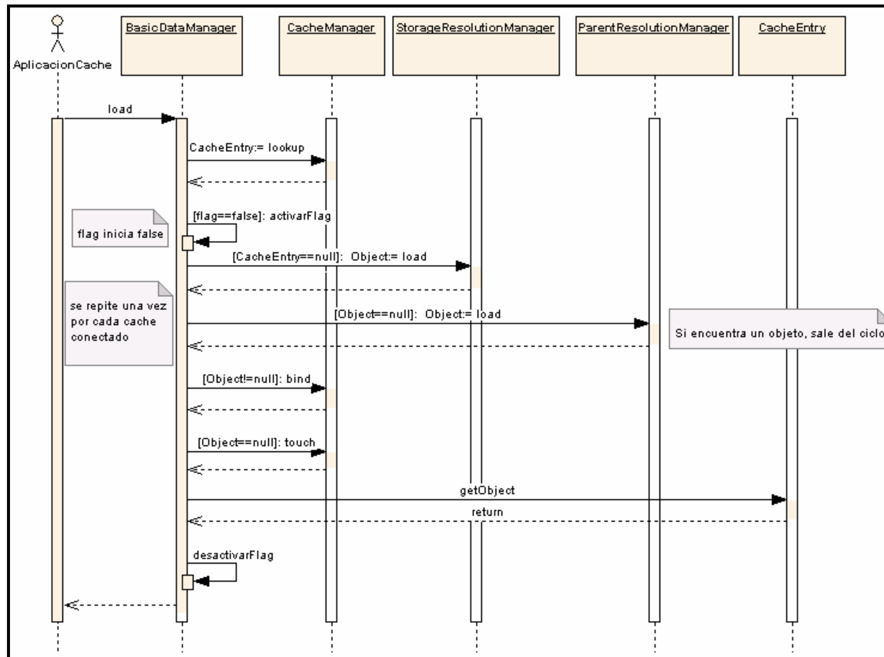


Figura 16. Diagrama de secuencia de la operación load



6. PRUEBAS

Al finalizar la implementación de *DHTCaché* se realizaron pruebas para verificar el cumplimiento del objetivo que es ofrecer una herramienta que apoye al desarrollador de aplicaciones *DHT* en la selección de una configuración de caché que pueda beneficiar una aplicación particular.

Se ejecutaron tres aplicaciones DHT diferentes desarrolladas por otras personas, las cuales utilizaron *DHTCaché* en lugar de *FreePastry* y configuraron el caché mediante diferentes archivos de configuración. Como producto de la ejecución de estas aplicaciones se produjeron archivos *log* que fueron analizados y como resultado se obtuvieron archivos de resultados con valores diferentes que sugieren que el desarrollador de aplicaciones los puede usar para sacar conclusiones sobre el desempeño del caché.

Adicionalmente, se diseñó un plan de pruebas que permitió verificar el correcto funcionamiento de la aplicación desarrollada, con base en los requerimientos. El diseño de las pruebas es conformado por la especificación de los casos de prueba y los procedimientos de prueba. De ser posible, también incluye los componentes de prueba que automaticen la realización de los procedimientos de prueba que se puedan automatizar. Cada caso de prueba incluye una descripción de la prueba, las entradas, los resultados esperados y algunas condiciones relevantes que deben satisfacerse para poder realizar la prueba. Un procedimiento de prueba es la forma como se realiza cada caso de prueba [17].

Los casos de prueba han sido clasificados en tres grupos: pruebas sobre la capa *DSS – DHT de DHTCaché*, las pruebas sobre el funcionamiento del caché y las pruebas sobre el componente analizador de archivos *log*.

6.1 PRUEBAS SOBRE EL SISTEMA *DHT*

El objetivo de este grupo de pruebas es verificar el funcionamiento de las clases modificadas a *FreePastry*, verificar la transparencia en la ejecución de aplicaciones *DHT* sobre la versión modificada de *FreePastry* y el registro de eventos en los archivos *log*.

6.1.1 Caso de prueba 01-01

Ejecución de una aplicación *DHT* en forma transparente sobre la versión modificada de *FreePastry*.

Descripción de la prueba: De un conjunto de tres aplicaciones *DHT* construidas por otras personas, se ejecutará la *aplicación1*, tanto en *FreePastry* sin modificaciones como en *FreePastry* modificado.

Entrada: Ninguna.

Resultado esperado: Ejecución normal sin ningún cambio.

Condiciones: Ninguna.

Clase ejecutable: *aplicación1.SourceMain.java.*

Procedimiento: Se ejecuta la *aplicación1* en los dos ambientes y se compara el resultado en la consola. Pueden variar los tiempos de ejecución puesto que la capa DSS no es la misma. Esta aplicación lee los argumentos de la línea de comandos. En este caso, utilice –*nodes 20 type 2*. Es de anotar que de igual manera se pueden ejecutar *aplicación2* con los siguientes argumentos de la línea de comandos: *-nodes 20* y *aplicación3*, que no necesita parámetros. Las clases ejecutables de aplicació2 y aplicación3 son *aplicación2.test.PastorTest.java* y *aplicación3.gui.dds_p2pg5.java*, respectivamente.

Resultado de la prueba: *Ok*.

6.1.2 Caso de prueba 01-02

Utilizar una aplicación *DHT* para probar el funcionamiento de los métodos *insert* y *lookup* de la clase *PastImpl* de *FreePastry*. De igual manera, se puede verificar el registro de acciones en los archivos *log*.

Descripción de la prueba: Se ejecuta una aplicación cuyo único objetivo es crear cuatro nodos e insertar un objeto, para posteriormente solicitarlo y de esta manera comprobar el funcionamiento de las operaciones *insert* y *lookup*. La clase ejecutable viene incluida dentro del código fuente de *FreePastry* y fue modificada para esta prueba.

Entrada: Un objeto generado en forma aleatoria.

Resultado esperado: Salida por la consola:

```
Simple Route Request
  Initial Lookup............................................[SUCCESS]
  File Insertion............................................[SUCCESS]
  Remote File Lookup........................................[SUCCESS]
  Local File Lookup.........................................[SUCCESS]
```

Creación de un archivo *cache.log* y cuatro archivos *log* (uno por cada nodo). Los nombres de los archivos inician con el prefijo *ceg* y tienen la extensión *.log*. Dentro de los archivos debe ser posible evidenciar que se hace una búsqueda que no encuentra el objeto buscado, luego una inserción y luego una búsqueda exitosa.

Condiciones: Ninguna.

Clase ejecutable: *rice.p2p.past.testing.PastRegrTest .java.*

Procedimiento: Se ejecuta la clase ejecutable con los argumentos *–nodes 4* y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok*.

6.2 PRUEBAS SOBRE EL CACHÉ *ACS*

El objetivo de este grupo de pruebas es verificar el funcionamiento de las clases modificadas al *framework ACS*, de la clase *ObjectCache* que se adicionó al *framework* y del registro de eventos en los archivos *log*.

6.2.1 Caso de prueba 02-01

Carga de propiedades de configuración del caché.

Descripción de la prueba: El caché puede ser configurado cargando las propiedades especificadas en el archivo *cache.properties*.

Entrada: Un archivo con las siguientes propiedades:

```
maxObjects = 10
autoCleanSize = 2
autoCleanThreshold = 10
replacement = lru
propertiesLogFileName = log.properties
```

Resultado esperado: Corresponde a la salida del método *toString()* de las propiedades que han sido efectivamente cargadas del archivo de propiedades. Es decir:

```
{propertiesLogFileName=log.properties, maxObjects=10, replacement=lru,
 autoCleanThreshold=10, autoCleanSize=2}
```

Condiciones: El archivo *cache.properties* con las propiedades mencionadas en la entrada debe existir en el directorio raíz del proyecto.

Clase ejecutable: *test.TestCargaArchivoConfiguracion.java.*

Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok*.

6.2.2 Caso de prueba 02-02

Carga de propiedades de configuración del caché por omisión

Descripción de la prueba: El caché puede ser configurado por defecto si no existe el archivo de configuración *cache.properties*.

Entrada: Ninguna.

Resultado esperado: Corresponde a la salida del un mensaje de error por no existir el archivo de propiedades y la salida asociada al método *toString()* de las propiedades que han sido especificadas como propiedades por defecto.

```
Error al abrir el archivo de propiedades
Cache inicializado con propiedades por defecto
{propertiesLogFileName=log.properties, maxObjects=10, replacement=fifo,
autoCleanThreshold=10, autoCleanSize=2}
```

Condiciones: El archivo *cache.properties* no debe existir en el directorio raíz del proyecto.

Clase ejecutable: t*est.TestCargaArchivoConfiguracion.java*.

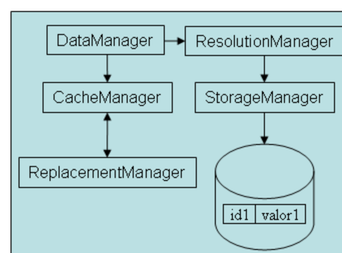
Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado. Debe asegurarse que el archivo *cache.properties* no exista para garantizar el éxito de la prueba.

Resultado de la prueba: *Ok*.

6.2.3 Caso de prueba 02-03

Creación de un caché con almacén de datos local y verificación de las operaciones *bind*, *lookup* y *load* en el caché.

Figura 17. Arquitectura de caché con almacén de datos local



Descripción de la prueba: El caché se crea con las propiedades especificadas en el archivo de configuración. Un objeto *String* será almacenado en el almacén de datos local. Luego el mismo objeto será buscado en el caché y no será encontrado. A continuación se intenta solucionar la falla de caché a través del *StorageManager*, usando el método *load* para traer el objeto del almacén de datos local. Finalmente se vuelve a buscar el objeto en el caché.

Entrada: Un objeto

Identificador	Objeto
id1	valor1

Resultado esperado: Salida en la consola:

```
lookup null
load valor1
lookup valor1
```

Condiciones: Si el archivo *cache.properties* no existe en el directorio raíz del proyecto, la salida indica que se han cargado las propiedades por defecto y luego la salida debe ser la esperada.

Clase ejecutable: *test.TestCrearCache01.java.*

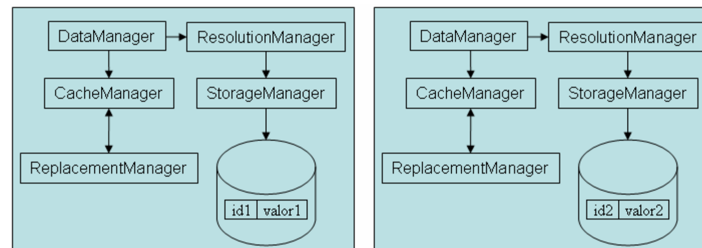
Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok*.

6.2.4 Caso de prueba 02-04

Creación de dos cachés cada uno con almacén de datos local y ninguna conexión entre ellos.

Figura 18. Dos cachés sin conexión entre ellos



Descripción de la prueba: Se crean dos cachés con las propiedades especificadas en el archivo de configuración. Un primer objeto *String* será almacenado en el almacén de datos local del primer caché y un segundo objeto *String* será almacenado en el segundo caché. Luego serán buscados los dos objetos en el primer caché y no serán encontrados. A continuación se intenta solucionar las fallas de caché a través del *StorageManager*, usando el método *load* para traer el objeto bien sea del almacén de datos local o del otro caché. Finalmente se vuelven a buscar los objetos en el caché.

Entrada: Dos objetos

Identificador	Objeto
id1	valor1
id2	valor2

Resultado esperado: Salida en la consola:

```
lookup null
lookup null
load valor1
load null
lookup valor1
lookup null
```

Condiciones: Si el archivo *cache.properties* no existe en el directorio raíz del proyecto, la salida indica que se han cargado las propiedades por defecto y luego la salida debe ser la esperada.

Clase ejecutable: *test.TestCrearCache02.java.*

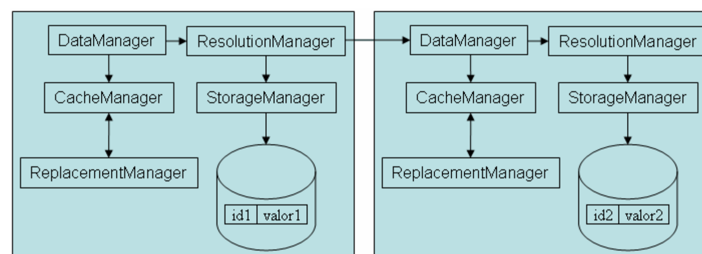
Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok.*

6.2.5 Caso de prueba 02-05

Creación de una arquitectura de caché distribuido de dos cachés donde las fallas del primer caché son resueltas en el almacén de datos local o en el otro caché.

Figura 19. Arquitectura de caché distribuido con dos cachés con enlace unidireccional



Descripción de la prueba: Se crean dos cachés con las propiedades especificadas en el archivo de configuración. Se establece una conexión unidireccional entre el primero y el segundo caché que se utiliza para resolver las fallas de caché. Un primer objeto *String* será almacenado en el almacén de datos local del primer caché y un segundo objeto *String* será almacenado en el segundo caché. Luego serán buscados los dos objetos en el primer caché y no serán encontrados. A continuación se intenta solucionar las fallas de caché a través del *StorageManager* y el *ParentResolutionManager*, usando el método *load* para traer el objeto

bien sea del almacén de datos local o del otro caché. Finalmente se vuelven a buscar los objetos en el caché.

Entrada: Dos objetos

Identificador	Objeto
id1	valor1
id2	valor2

Resultado esperado: Salida en la consola:

```
lookup null
lookup null
load valor1
load valor2
lookup valor1
lookup valor2
```

Condiciones: Si el archivo *cache.properties* no existe en el directorio raíz del proyecto, la salida indica que se han cargado las propiedades por defecto y luego la salida debe ser la esperada.

Clase ejecutable: *test.TestCrearCache03.java.*

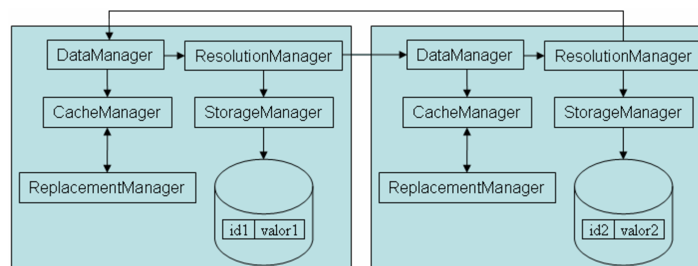
Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok*.

6.2.6 Caso de prueba 02-06

Creación de una arquitectura de caché distribuido de dos cachés donde las fallas de caché son resueltas o en el almacén de datos local o en el otro caché.

Figura 20. Arquitectura de caché distribuido con dos cachés con enlace bidireccional



Descripción de la prueba: Se crean dos cachés con las propiedades especificadas en el archivo de configuración. Se establece una conexión bidireccional entre los cachés se utiliza para resolver las fallas de caché. Un primer objeto *String* será almacenado en el

almacén de datos local del primer caché y un segundo objeto *String* será almacenado en el segundo caché. Luego serán buscados los dos objetos en el tanto en el primer caché como en el segundo y no serán encontrados. A continuación se intenta solucionar las fallas de caché a través del *StorageManager* y el *ParentResolutionManager*, usando el método *load* para traer el objeto bien sea del almacén de datos local o del otro caché. Finalmente se vuelven a buscar los objetos en ambos cachés.

Entrada: Dos objetos

Identificador	Objeto
id1	valor1
id2	valor2

Resultado esperado: Salida en la consola:

```
lookup null
lookup null
lookup null
lookup null
load valor1
load valor2
load valor1
load valor2
lookup valor1
lookup valor2
lookup valor1
lookup valor2
```

Condiciones: Si el archivo *cache.properties* no existe en el directorio raíz del proyecto, la salida indica que se han cargado las propiedades por defecto y luego la salida debe ser la esperada.

Clase ejecutable: *test.TestCrearCache04.java.*

Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok*.

6.2.7 Caso de prueba 02-07

Creación de una arquitectura de caché distribuido jerárquico de dos niveles y tres cachés donde las fallas de caché son resueltas o en el almacén de datos local o en otro caché.

Figura 21. Arquitectura de caché jerárquico en dos niveles

Descripción de la prueba: Se crean tres cachés con las propiedades especificadas en el archivo de configuración. Se forma una arquitectura jerárquica de dos niveles con un caché padre y dos cachés hijos. Se establece una conexión bidireccional entre los cachés hijos y el caché padre. Estas conexiones servirán para resolver las fallas de caché. Un primer objeto *String* será almacenado en el almacén de datos local del caché padre, dos objetos *String* más serán almacenados, uno en cada almacén de datos local de cada caché hijo. Luego serán buscados los tres objetos en cada caché y no serán encontrados. A continuación se intenta solucionar las fallas de caché a través del *StorageManager* y el *ParentResolutionManager*, usando el método *load* para traer el objeto bien sea del almacén de datos local o de otro caché. Finalmente se vuelven a buscar los objetos en ambos cachés.

Entrada: Tres objetos

Identificador	Objeto
id1	valor1
id2	valor2
id3	valor3

Resultado esperado: Salida en la consola:

```
lookup null
lookup null
lookup null
lookup null
lookup null
lookup null
lookup null
lookup null
lookup null
load valor1
```

```
load valor2
load valor3
load valor1
load valor2
load valor3
load valor1
load valor2
load valor3
lookup valor1
lookup valor2
lookup valor3
lookup valor1
lookup valor2
lookup valor3
lookup valor1
lookup valor2
lookup valor3
```

Condiciones: Si el archivo *cache.properties* no existe en el directorio raíz del proyecto, la salida indica que se han cargado las propiedades por defecto y luego la salida debe ser la esperada.

Clase ejecutable: *test.TestCrearCache05.java.*

Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

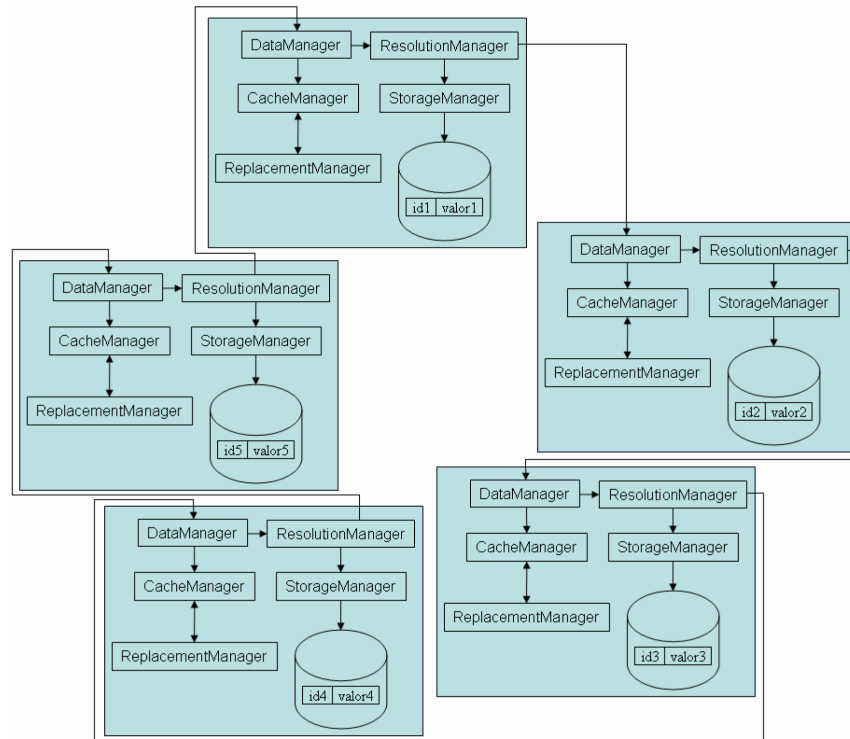
Resultado de la prueba: *Ok*.

6.2.8 Caso de prueba 02-08

Creación de una arquitectura de caché distribuido plano en anillo con cinco cachés donde las fallas de caché son resueltas o en el almacén de datos local o en otro caché.

Descripción de la prueba: Se crean cinco cachés con las propiedades especificadas en el archivo de configuración. Se forma una arquitectura distribuida plana. Se establece una conexión unidireccional entre los cachés de la siguiente forma: el primer caché con el segundo, el segundo con el tercero, el tercero con el cuarto, el cuarto con el quinto y el quinto con el primero. Estas conexiones servirán para resolver las fallas de caché. Cinco objetos *String* serán almacenados, uno en cada almacén de datos local del cada caché. Luego serán buscados los objetos de la siguiente forma: el cache1 buscará los cinco objetos y no serán encontrados. A continuación se intenta solucionar las fallas de caché a través del *StorageManager* y el *ParentResolutionManager*, usando el método *load* para traer los objetos bien sea del almacén de datos local o de otro caché. Después el caché2 buscará traer los objetos 4 y 1 del almacén de datos local o de algún caché y el caché5 hará lo mismo con el objeto 2. Finalmente otros objetos serán buscados en el caché: el caché1 el objeto1, el caché2 los objetos 1 y 4, el caché5 el objeto2, el caché2 el objeto3 y el caché4 el objeto3.

Figura 22. Arquitectura de caché distribuido en anillo



Entrada: Cinco objetos

Identificador	Objeto
id1	valor1
id2	valor2
id3	valor3
id4	valor4
id5	valor5

Resultado esperado: Salida en la consola:

```
lookup null
lookup null
lookup null
lookup null
lookup null
load valor1
load valor2
load valor3
load valor4
load valor5
load valor4
load valor1
load valor2
lookup valor1
lookup valor1
lookup valor4
lookup valor2
```

```
lookup valor3
lookup null
```

Condiciones: Si el archivo *cache.properties* no existe en el directorio raíz del proyecto, la salida indica que se han cargado las propiedades por defecto y luego la salida debe ser la esperada.

Clase ejecutable: *test.TestCrearCache06.java.*

Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok*.

6.2.9 Caso de prueba 02-09

Registro de eventos en el archivo *cache.log*.

Descripción de la prueba: Creación de un caché y verificación del registro de eventos en el archivo *log*.

Entrada: Un objeto

Identificador	Objeto
id1	valor1

Resultado esperado: Creación del archivo log con el registro de los eventos lookup, bind y lookup hit con los siguientes campos en cada línea: *tiempo*, *evento*, *identificador* y *duración*.

Condiciones: Ninguna.

Clase ejecutable: *test.TestCrearCache01.java.*

Procedimiento: Se ejecuta la clase ejecutable sin ninguna clase de argumentos y se compara el resultado obtenido con el resultado esperado.

Resultado de la prueba: *Ok*.

6.3 PRUEBAS SOBRE EL ANALIZADOR DE ARCHIVOS *LOG*

El objetivo de este grupo de pruebas es verificar el funcionamiento del componente analizador de archivos *log*, para obtener calcular las métricas que fueron implementadas.

6.3.1 Caso de prueba 03-01

Comprobar la tabla que almacena todos los eventos registrados por identificador.

Descripción de la prueba: Utilizar dos archivos *log* ficticios, creados manualmente, para comprobar si son cargados correctamente en la tabla *hash tablaId*.

Entrada: Dos archivos *pruebaCache.log* y *pruebaDht.log*, con el siguiente contenido:

pruebaCache.log

```
0 CEG_Bind 10101010 100 ns
20 CEG_Bind 10101010 200 ns
40 CEG_Bind 10101010 300 ns
60 CEG_Bind 10101010 400 ns
80 CEG_Bind 20202020 1000 ns
100 CEG_Bind 20202020 2000 ns
120 CEG_Bind 20202020 3000 ns
140 CEG_Bind 20202020 4000 ns
160 CEG_Lookup 10101010 100 ns
180 CEG_Lookup 10101010 100 ns
200 CEG_Lookup 10101010 300 ns
220 CEG_Lookup 10101010 300 ns
240 CEG_Lookup 20202020 1000 ns
260 CEG_Lookup 20202020 1000 ns
280 CEG_Lookup 20202020 3000 ns
300 CEG_Lookup 20202020 3000 ns
320 CEG_Lookup_Hit 10101010 100 ns
340 CEG_Lookup_Hit 10101010 100 ns
360 CEG_Lookup_Hit 10101010 100 ns
380 CEG_Lookup_Hit 10101010 100 ns
400 CEG_Lookup_Hit 20202020 1000 ns
420 CEG_Lookup_Hit 20202020 1000 ns
440 CEG_Lookup_Hit 20202020 1000 ns
460 CEG_Lookup_Hit 20202020 1000 ns
```

pruebaDht.log

```
fecha 20:55:34,044:CEG_Insert 10101010 1000 ns
fecha 20:55:34,054:CEG_Insert 10101010 2000 ns
fecha 20:55:34,064:CEG_Insert 10101010 3000 ns
fecha 20:55:34,074:CEG_Insert 10101010 2000 ns
fecha 20:55:34,044:CEG_Get 10101010 1000 ns
fecha 20:55:34,054:CEG_Get 10101010 2000 ns
fecha 20:55:34,064:CEG_Get 10101010 2000 ns
fecha 20:55:34,074:CEG_Get 10101010 1000 ns
fecha 20:55:34,044:CEG_Get_Hit 10101010 100 ns
fecha 20:55:34,054:CEG_Get_Hit 10101010 200 ns
fecha 20:55:34,064:CEG_Get_Hit 10101010 200 ns
fecha 20:55:34,074:CEG_Get_Hit 10101010 100 ns
fecha 20:55:34,044:CEG_Insert 20202020 1000 ns
fecha 20:55:34,054:CEG_Insert 20202020 2000 ns
fecha 20:55:34,064:CEG_Insert 20202020 3000 ns
fecha 20:55:34,074:CEG_Insert 20202020 4000 ns
fecha 20:55:34,044:CEG_Get 20202020 1000 ns
fecha 20:55:34,054:CEG_Get 20202020 1000 ns
fecha 20:55:34,064:CEG_Get 20202020 2000 ns
fecha 20:55:34,074:CEG_Get 20202020 4000 ns
fecha 20:55:34,044:CEG_Get_Hit 20202020 100 ns
fecha 20:55:34,054:CEG_Get_Hit 20202020 100 ns
fecha 20:55:34,064:CEG_Get_Hit 20202020 200 ns
fecha 20:55:34,074:CEG_Get_Hit 20202020 100 ns
```

Resultado esperado: Un archivo de texto llamado *resultados.txt* con el siguiente contenido:

```
10101010,
[[0 - CEG_Bind - 100.0], [20 - CEG_Bind - 200.0],
[40 - CEG_Bind - 300.0], [60 - CEG_Bind - 400.0],
[160 - CEG_Lookup - 100.0], [180 - CEG_Lookup - 100.0],
[200 - CEG_Lookup - 300.0], [220 - CEG_Lookup - 300.0],
[320 - CEG_Lookup_Hit - 100.0], [340 - CEG_Lookup_Hit - 100.0],
[360 - CEG_Lookup_Hit - 100.0], [380 - CEG_Lookup_Hit - 100.0],
[34,044 - CEG_Insert - 1000.0], [34,054 - CEG_Insert - 2000.0],
[34,064 - CEG_Insert - 3000.0], [34,074 - CEG_Insert - 2000.0],
[34,044 - CEG_Get - 1000.0], [34,054 - CEG_Get - 2000.0],
[34,064 - CEG_Get - 2000.0], [34,074 - CEG_Get - 1000.0],
[34,044 - CEG_Get_Hit - 100.0], [34,054 - CEG_Get_Hit - 200.0],
[34,064 - CEG_Get_Hit - 200.0], [34,074 - CEG_Get_Hit - 100.0]]

20202020,[[80 - CEG_Bind - 1000.0], [100 - CEG_Bind - 2000.0],
[120 - CEG_Bind - 3000.0], [140 - CEG_Bind - 4000.0],
[240 - CEG_Lookup - 1000.0], [260 - CEG_Lookup - 1000.0],
[280 - CEG_Lookup - 3000.0], [300 - CEG_Lookup - 3000.0],
[400 - CEG_Lookup_Hit - 1000.0], [420 - CEG_Lookup_Hit - 1000.0],
[440 - CEG_Lookup_Hit - 1000.0], [460 - CEG_Lookup_Hit - 1000.0],
[34,044 - CEG_Insert - 1000.0], [34,054 - CEG_Insert - 2000.0],
[34,064 - CEG_Insert - 3000.0], [34,074 - CEG_Insert - 4000.0],
[34,044 - CEG_Get - 1000.0], [34,054 - CEG_Get - 1000.0],
[34,064 - CEG_Get - 2000.0], [34,074 - CEG_Get - 4000.0],
[34,044 - CEG_Get_Hit - 100.0], [34,054 - CEG_Get_Hit - 100.0],
[34,064 - CEG_Get_Hit - 200.0], [34,074 - CEG_Get_Hit - 100.0]]
```

Condiciones: Los dos archivos de entrada mencionados deben existir en el directorio raíz del proyecto.

Clase ejecutable: *stats.StatsCacheMain01.java*

Procedimiento: Se ejecuta la clase ejecutable sin argumentos de la línea de comandos.

Resultado de la prueba: *Ok*.

6.3.2 Caso de prueba 03-02

Comprobar la tabla que consolida todos los eventos registrados por identificador y por evento.

Descripción de la prueba: Utilizar dos archivos *log* ficticios, creados manualmente, para comprobar si en la tabla *hash tablaIdEvento* es acumulado correctamente el tiempo promedio y la frecuencia de cada evento, separados por identificador y evento.

Entrada: Dos archivos *pruebaCache.log* y *pruebaDht.log*, con el mismo contenido de la prueba anterior.

Resultado esperado: Un archivo de texto llamado *resultados.txt* con el siguiente contenido:

```
10101010
CEG_Insert,(2000.0 - 4)
CEG_Lookup_Hit,(100.0 - 4)
CEG_Lookup,(200.0 - 4)
CEG_Bind,(250.0 - 4)
CEG_Get_Hit,(150.0 - 4)
CEG_Get,(1500.0 - 4)

20202020
CEG_Insert,(2500.0 - 4)
CEG_Lookup_Hit,(1000.0 - 4)
CEG_Lookup,(2000.0 - 4)
CEG_Bind,(2500.0 - 4)
CEG_Get_Hit,(125.0 - 4)
CEG_Get,(2000.0 - 4)
```

Condiciones: Los dos archivos de entrada mencionados deben existir en el directorio raíz del proyecto.

Clase ejecutable: *stats.StatsCacheMain02.java*

Procedimiento: Se ejecuta la clase ejecutable sin argumentos de la línea de comandos.

Resultado de la prueba: *Ok*.

6.3.3 Caso de prueba 03-03

Comprobar el correcto almacenamiento de datos en la tabla llamada *tabla* la cual resume el tiempo promedio y la frecuencia, agrupados por evento.

Descripción de la prueba: Utilizar dos archivos *log* ficticios, creados manualmente, para comprobar si en la tabla *hash tabla* es acumulado correctamente el tiempo promedio y la frecuencia de cada evento, separados por evento. En esta tabla se obtiene el tiempo promedio que el sistema ha tardado en completar todos los eventos.

Entrada: Dos archivos *pruebaCache.log* y *pruebaDht.log*, con el mismo contenido de la prueba anterior.

Resultado esperado: Un archivo de texto llamado *resultados.txt* con el siguiente contenido:

```
CEG_Insert,(2250.0 - 8)
CEG_Lookup,(1100.0 - 8)
CEG_Lookup_Hit,(550.0 - 8)
CEG_Bind,(1375.0 - 8)
CEG_Get_Hit,(137.5 - 8)
CEG_Get,(1750.0 - 8)
```

Condiciones: Los dos archivos de entrada mencionados deben existir en el directorio raíz del proyecto.

Clase ejecutable: *stats.StatsCacheMain03.java*

Procedimiento: Se ejecuta la clase ejecutable sin argumentos de la línea de comandos.

Resultado de la prueba: *Ok*.

7. CONCLUSIONES Y TRABAJO FUTURO

En este trabajo de tesis se ha presentado un sistema para la obtención de métricas de diferentes configuraciones de caché en aplicaciones *DHT*, el cual busca dar información al desarrollador de aplicaciones *DHT* que le pueda servir para decidir la configuración de caché que mejor se acomode a las aplicaciones *DHT*.

Para dar inicio al desarrollo del software, se creó un prototipo en *Java* en el cual se utilizó el *framework ACS* [13] y la versión *1.4.4* de *FreePastry* [38]. Se hicieron mejoras importantes a *ACS* para que soportara cualquier tipo de topología pensada por un desarrollador para ubicar los cachés distribuidos y enlazarlos de la forma que deseara, controlando los ciclos infinitos que se pueden formar en esta clase de redes. En *FreePastry* el avance fue menor, porque se incluyó un cambio en el almacenamiento y recuperación de objetos en la implementación de *Past*.

El software desarrollado utiliza *loggers* para registrar eventos en archivos *log* y a partir de la información almacenada en ellos calcula métricas que pueden servir para la toma de decisiones en la configuración de cachés en aplicaciones *DHT*. Las métricas calculadas pueden ser usadas para comparar el comportamiento de diferentes configuraciones de caché en aplicaciones *DHT*. Esta información es importante para que el desarrollador pueda tomar una mejor decisión con respecto a la configuración del caché en sus aplicaciones *DHT*.

Las pruebas realizadas dan evidencia del software desarrollado y el cumplimiento satisfactorio de la mayoría de los casos de uso definidos, y algunos de ellos quedaron pendientes para trabajo futuro. Hace falta usar una aplicación robusta que ponga a prueba el software y permita valorar realmente su capacidad del software. Además, dado que usar manualmente una aplicación para obtener archivos *log* con el registro de diferentes eventos que ocurren con el caché o con el sistema *DHT* es muy dispendioso, es importante que el desarrollador de aplicaciones pueda diseñar pruebas con datos reales que se puedan cargar rápidamente en el sistema y de este modo obtener resultados que sirvan para tomar decisiones que conduzcan a mejorar realmente el desempeño de sus aplicaciones.

Para el futuro, quedan elementos que deben ser tenidos en cuenta en el sistema planteado, como definir un mecanismo para que el desarrollador de aplicaciones *DHT* pueda especificar la arquitectura de caché distribuido aprovechando la información de enrutamiento de la capa *DLS* que permita probar las arquitecturas de caché distribuido que han sido desarrolladas; incorporar el uso de cachés en la capa de búsqueda; estudiar la cooperación entre los cachés de las diferentes capas en el mismo nodo y de los cachés de la misma capa en diferentes *peers*; hacer un mayor análisis del impacto del caché frente a las características de los sistemas *peer-to-peer* y realizar pruebas en una red real.

8. REFERENCIAS

[1] Advanced Networking Applications (ANA). Disponible en: http://www.cs.usask.ca/faculty/carey/projects/finalreport.html.
[2] G. Alsina. (2005). Microsoft incorpora tecnología P2P a Windows Vista. Disponible en: http://www.noticias.com/articulo/14-09-2005/guillem-alsina/microsoft-incorpora-tecnologia-p2p-windows-vista-4l7i.html.
[3] J. Álvarez. Arquitectura del computador. Disponible en: http://people.fluidsignal.com/~jalvarez/courses/sistemas_operativos /lecture/lecture-1.pdf.
[4] S. Androutsellis-Theotokis y D. Spinellis. (2004). A survey of peer-to-peer content distribution technologies. En: ACM Comput. Surv. 36, 4 (Dec. 2004), 335-371.
[5] A. Bonilla y J. Meler. Aplicaciones distribuidas: P2P. Disponible en: http://studies.ac.upc.edu/FIB/CASO/seminaris/2q0304/M9.pdf.
[6] M. Cai, M. Frank, J. Chen y P. Szekely. (2003). Maan: A Multi-Attribute Addressable Network for Grid Information Services. Proceedings of the Fourth International Workshop on Grid Computing.
[7] I. Cooper, I. Melve y G. Tomlinson. Request for Comments: 3040. Internet Web Replication and Caching Taxonomy. January 2001.
[8] G. Coulouris, J. Dollimore y T. Kindberg. (2005). Distributed Systems: Concepts and design. Fourth Edition. Addison Wesley, USA.
[9] F. Dabek. (2005). A Distributed Hash Table. En: PhD's thesis, Massachusetts Institute of Technology.
[10] F. Dabek, E. Brunskill, M. Kaashoek, D. Karger, R. Morris, I. Stoica y H. Balakrishnan. (2001). Building peer-to-peer systems with a Chord, a distributed lookup service. Proceedings of the Eighth Workshop on Hot Topics in Operating Systems.
[11] F. Dabek, M. Kaashoek, D. Karger, R. Morris y I. Stoica. (2001). Wide-area cooperative storage with CFS. Proceedings of the 18th {ACM} {S}ymposium on {O}perating {S}ystems {P}rinciples ({SOSP} '01).
[12] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz y I. Stoika. (2003). Towards a common API for structured peer-to-peer overlays. In IPTPS '03, Berkeley, CA.
[13] L. d'Orazio, F. Jouanot, C. Labbé, y C. Roncancio. 2005. Building adaptable cache services. In Proceedings of the 3rd international Workshop on Middleware For Grid Computing (Grenoble, France, November 28 - December 02, 2005). MGC '05, vol. 117. ACM Press, New York, NY, 1-6.
[14] O. Gnawali. (2002). A keyword-set search system for peer-to-peer networks. En: Master's thesis, Massachusetts Institute of Technology.
[15] Holmedahl, V., Smith, B., and Yang, T. 1998. Cooperative Caching of Dynamic Content on a Distributed Web Server. Proceedings of the Seventh IEEE international Symposium on High Performance Distributed Computing (July 28 - 31, 1998). HPDC. IEEE Computer Society, Washington, DC, 243.

[16] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker y I. Stoica. (2003). Querying the Internet with Pier. Proceedings of the international Conference on VLDB.
[17] I. Jacobson, G. Booch y J. Rumbaugh. El proceso unificado de desarrollo de software. Addison Wesley. Madrid, 2000.
[18] J. Kim. P2P computing in Advanced topics in computer networking. Disponible en: http://netmedia.gist.ac.kr/courses/dic1698-2006fa/lectures/lec04-p2p.pdf.
[19] J. Kurose y K. Ross. 2005. Computer networking: A top-down approach featuring the Internet. Third edition. Addison Wesley. USA.
[20] W. Liao y C. King. 2001. Proxy Prefetch and Prefix Caching. In: International Conference on Parallel Processing (ICPP '01) p. 0095.
[21] J. Mañas. Log: trazas de ejecución. Disponible en: http://www.lab.dit.upm.es/~lprg/material/apuntes/log/log.htm.
[22] Microsoft proxy server. Disponible en: http://www.microsoft.com/technet/archive/proxy/prxcarp.mspx?mfr=true.
[23] L. Navarro y V. Sossa. Distribución Geográfica de Documentos Web a Bibliotecas-proxy . Disponible en: http://www.rediris.es/rediris/boletin/46-47/ponencia13.html.
[24] G. Provost, How Caching Works en http://computer.howstuffworks.com/cache.htm.
[25] M.S. Raunak. Web caching reviews. Disponible en: http://lass.cs.umass.edu/ ~shenoy / courses/spring00/791u/web1.html.
[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp y S. Shenker. (2001). A Scalable Content Addressable Network. In Proc. ACM SIGCOMM 2001.
[27] P. Rodriguez, C. Spanner, y E. Biersack. 2001. Analysis of web caching architectures: hierarchical and distributed caching. IEEE/ACM Trans. Netw. 9, 4 (Aug. 2001), 404-418.
[28] A. Rowstron, P. Druschel. (2001). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware).
[29] A. Rowstron y P. Druschel. (2001). Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. ACM SOSP.
[30] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashock, F. Dabek y H. Bañakrishan. (2001). Chord: A scalable peer-to-peer lookup protocol for Internet applications. Proceedings of the 2001 ACM SIGCOMM Conference.
[31] R. Subramanian y B. Goodman. (2005). Peer-to-peer computing. The evolution of a disruption technology. Idea Group Publishing.
[32] L. Sung, N. Ahmed, R. Blanco, H. Li, M. Soliman y D. Hadaller. (2005).A survey of data management in peer-to-peer systems. School of Computer Science, University of Waterloo.
[33] A. Tanenbaum. (2004). Redes de computadoras. Cuarta edición. Pearson Educación. México.
[34] A. I. Vakali y G.E. Pallis. 2001. A study on Web caching architectures and performance. Proceedings of 5th World Multi-Conference on Systemics, Cybernetics and Informatics.
[35] M. Villamil. (2007). Manejo de datos en sistemas a gran escala. Magíster en Ingeniería de Sistemas y Computación. Universidad de Los Andes.

[36] M. Villamil, C. Roncancio y C. Labbe. (2004). PinS: Peer to Peer Interrogation and Indexing System. Website disponible en: http://www-lsr.imag.fr/Les.Personnes/Maria-Del-Pilar.Villamil/PinS/.
[37] M. Villamil, C. Roncancio y C. Labbé. (2006). Range queries in massively distributed data. Proceedings of the 17th International Conference on Database and Expert Systems Applications.
[38] Web site de FreePastry. http://freepastry.org/.
[39] Web site de Perseus. http://perseus.objectweb.org/
[40] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, y J. Kubiatowicz. (2003). Tapestry: A Resilient Global-Scale Overlay for Service Deployment. En: IEEE Journal on Selected Areas in Communications.
[41] B. Zhao, J. Kubiatowicz y A. Joseph. (2001). Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. UC Berkeley.

ANEXO

Descripción de *ACS* - *Adaptable Cache Services*

ADAPTABLE CACHE SERVICES

ACS (*Adaptable Caché Services*) [13], es un *framework* construido en *Java* que permite la construcción de servicios de caché a partir de componentes intercambiables. *ACS* está basado en *Perseus* y extiende sus posibilidades adicionando componentes que aumentan la flexibilidad de los sistemas de caché que pueden ser construidos a partir de *ACS*.

ACS fue realizado por Laurent d'Orazio, Fabrice Jouanot, Cyril Labbé y Claudia Roncancio en el *IMAG (Institut d'Informatique et Mathématiques Apliques de Grenoble* - Instituto de Informática y Matemáticas Aplicadas de Grenoble), Francia.

Según *ACS*, un caché se compone de tres partes fundamentales:

- Manejo de caché: direccionamiento y búsqueda de objetos en el caché.
- Manejo de reemplazo: determina el objeto u objetos que deben ser desalojados cuando el espacio disponible no permite insertar nuevos objetos en el caché.
- Manejo de resolución: define el proceso a seguir cuando hay un caché miss.

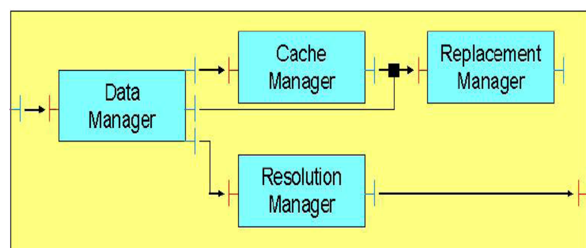
ACS está formado por un conjunto de clases abstractas e interfaces *Java*, y la forma como interactúan sus diferentes instancias.

Al facilitar el desarrollo de servicios de caché, el desarrollador puede enfocarse más en las estrategias de su aplicación que en el proceso de implementación.

Arquitectura

En la arquitectura del *framework ACS* existen cuatro componentes principales, los cuales se puede apreciar en la figura *23*.

Figura 23. Arquitectura de ACS



CacheManager

Es la *interface* vista funcional del servicio de caché. Todos los accesos al caché son realizados a través de la instancia del caché proporcionado en la aplicación.

El objetivo principal del servicio es mejorar el desempeño general del sistema. La ganancia en el desempeño se obtiene evitando costos de operaciones de entrada y salida relacionados con los dispositivos de almacenamiento y de red.

Las operaciones que soporta son:

- ***lookup*:** Este método busca en el caché el objeto especificado. Si el objeto ha sido adicionado al caché retorna una referencia al objeto, en caso contrario retorna *null*.
- ***bind*:** Este método permite almacenar en el caché el objeto especificado.
- ***fix*:** Este método notifica al *cacheManager* la intención de usar un objeto especificado mediante su identificador. El *cacheManager* marcará internamente este objeto de manera que el objeto no pueda ser desalojado del caché.
- ***unfix*:** Este método es usado para notificar al *cacheManager* que el objeto especificado ya no será usado. Después de ejecutar esta operación, el objeto correspondiente es candidato al desalojo de acuerdo con el algoritmo de reemplazo. Esto significa que el objeto quedará en el caché pero que podrá ser desalojado.
- ***touch*:** Este método es llamado cuando un objeto ha sido accedido. Así, este método da pistas acerca de lo reciente o frecuente que ha sido su uso. Estas pistas son usadas dentro de los algoritmos de reemplazo.

El *CacheManager* maneja objetos en el caché que sean instancias de *CacheEntry* (una *interface* que encapsula un objeto y su identificador).

ReplacementManager

Es una *interface* que especifica el comportamiento del caché cuando es necesario desalojar algún objeto para insertar un elemento nuevo. Recibe del *CacheManager* información sobre el uso de objetos que están en el caché con el fin de que sea tomada la decisión de los objetos que deben ser desalojados del caché de acuerdo con la política de reemplazo. Las implementaciones que tiene actualmente el *framework* son *LRU* y *FIFO*.

Los métodos principales son:

- ***addForReplacement*:** Este método es llamado por el CacheManager para señalar que una entrada ha sido adicionada al caché.
- ***removeForReplacement*:** Este método es llamado por el CacheManager para señalar que una entrada ha sido realmente desalojada del caché.
- ***adjustForReplacement*:** Este método llama un objeto que ha sido accedido. De este modo da pistas acerca de lo reciente o frecuente ha sido su uso. Estas pistas son usadas dentro del algoritmo de reemplazo.
- ***forceFree*:** Este método obliga al *ReplacementManager* a liberar instancias de CacheEntry del caché.

ResolutionManager

Es un aporte de *ACS* con respecto a *Perseus*. Este componente es el encargado de recuperar un objeto solicitado cuando el objeto no está en el caché. Puede ser utilizado cuando se configura un caché jerárquico.

Igual que en los manejadores de caché y de reemplazo, el manejador de resolución también es una *interface Java* y las implementaciones que vienen con el *framework* son las siguientes.

- ***StorageResolutionManager*:** En caso que se presente una falla de caché, la acción a seguir es buscarlo en una fuente de datos.
- ***ParentResolutionManager*:** En caso que se presente una falla de caché, la acción a seguir es buscarlo en un caché de nivel superior.

El único método que ofrece este manejador es: *load*, el cual se encarga de traer un objeto y luego dicho objeto es almacenado en el caché.

DataManager

El *DataManager* es una *interface* estandar para los clientes del sistema de caché (aplicaciones y cachés). El *DataManager* es usado como pegamento entre todos los componentes y como una *interface* para los clientes del servicio de caché. Esa *interface* es necesaria para acceder a los métodos de los componentes usados por los clientes del caché. Ayuda en el manejo de soluciones jerárquicas.

Las operaciones ofrecidas son:

***lookup*:** Busca el objeto solicitado en el caché asociado al nodo.
***load*:** Busca el objeto solicitado, bien sea en el caché asociado al nodo o donde corresponda de acuerdo con la política de resolución.

ACS está estrechamente relacionado con *Perseus Cache Manager*, un proyecto del *ObjectWeb Consortium* una comunidad *Open Source* dedicada a la tecnología *middleware* [39]. A continuación, una descripción de *Perseus Cache Manager*.

PERSEUS CACHEMANAGER

Perseus proporciona varios componentes que pueden ser utilizados para construir manejadores diferentes de persistencia, entre los que se destaca el componente *CacheManager* [39].

El objetivo principal del *CacheManager* es mejorar el desempeño general del sistema. La ganancia en desempeño es obtenida evitando costos de entrada y salida relacionados con el almacenamiento y la operación de los dispositivos de red. El *CacheManager* es el medio a través del cual son usados varios servicios como el de persistencia y de replicación, entre otros [39].

El componente *CacheManager*

El componente *CacheManager* permite construir un manejador de caché de objetos *Java*. Cada objeto almacenado en caché debe tener un identificador. El objeto y su identificador componen una entrada de caché (*CacheEntry*). El *CacheManager* tiene las siguientes funcionalidades [39]:

- **Política de reemplazo:** La política de reemplazo es la responsable de elegir el objeto que se debe desalojar del caché cuando el especio se hace necesario para alojar otro objeto. La implementación actual proporciona tres políticas: *LRU*, *MRU* y *FIFO*. Para escoger la política de reemplazo, usted tiene que reconfigurar el componente cambiando el manejador de reemplazo por defecto (*LRU*) por otro manejador de reemplazo.
- **Limpieza asincrónica en *background*:** Cunado el tamaño del caché alcanza un umbral, definido por el usuario, se activa un *thread* en *background* para liberar espacio en el caché.

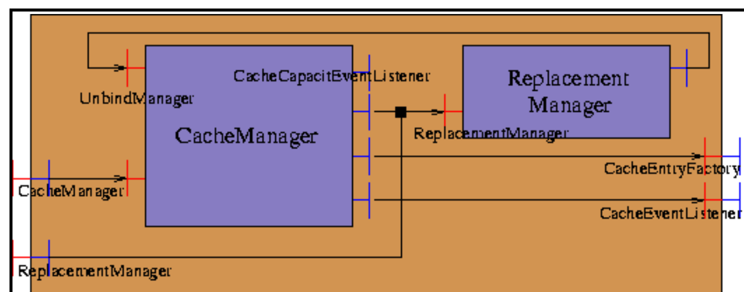
Los atributos del *CacheManager* son:

- ***MaxSize*:** El máximo tamaño del caché en número de objetos.
- ***MemoryMaxSize*:** El máximo tamaño del caché en términos de memoria (no implementado aún).
- ***AutoCleanSize*:** El número de elementos que deben ser eliminados cuando el caché está lleno o cuando se ejecuta *thread* el limpiador en *background*. Puede ser un valor absoluto o un porcentaje del tamaño máximo del caché.
- ***AutoCleanThreshold*:** Es el valor del tamaño del caché a partir del cual intenta disminuir el número de entradas. Puede ser un valor absoluto o un porcentaje del tamaño máximo del caché.

Arquitectura

El rol del *CacheManager* es mantener objetos en memoria. Un *CacheManager* maneja entradas representadas por instancias de *CacheEntry*. La figura *24* representa la arquitectura general del componente *CacheManager*. Ella muestra una composición de componentes que exportan dos interfaces realmente implementadas como dos subcomponentes: una primitiva *CacheManager* que mantiene un mapeo de entradas de caché y un manejador de reemplazo que aplica políticas de desalojo de entradas. Además, un sensor (*CacheEventListener*) permite al usuario de un caché manager recuperar un manejo de entradas de caché. Finalmente, el usuario del caché manager debe especificar una fábrica de instancias *CacheEntry* [39].

Figura 24. Arquitectura del CacheManager de Perseus



CacheEntry

Un *CacheEntry* es una *interface* que define una entrada al *CacheManager*. Un *CacheEntry* referencia el identificador de un objeto y el objeto. *CacheEntry* ofrece dos métodos [39]:

- ***getCeObject:*** Retorna el objeto referenciado.
- ***getCeIdentifier:*** Retorna el identificador del objeto referenciado.

Para implementar políticas de reemplazo y para separar asuntos adicionales como uso y edad de las entradas del caché, el concepto de CacheEntry es extendido en dos interfaces *FixableCacheEntry* y *ReplaceableCacheEntry*.

FixableCacheEntry

Permite señalar el uso de una entrada del caché. Esta *interface* define tres métodos [39]:

- ***fixCe:*** Indica el comienzo del uso de un objeto.
- ***unfixCe:*** Indica el final del uso de un objeto.
- ***getCeFixCount:*** Retorna el número de usos del objeto.

ReplaceableCacheEntry

Permite manejar la edad de una entrada en el caché. El manejador de reemplazo puede usar la edad de las entradas para escoger la siguiente entrada a eliminar. Esta *interface* define dos métodos [39]:

- ***setCeAge:*** Establece la edad de una entrada en el caché.
- ***getCeAge:*** Retorna la edad de una entrada en el caché.

CacheEntryFactory

Una *CacheEntryFactory* es un componente capaz de crear nuevas instancias de *CacheEntry* a partir de un objeto y su identificador. Esta *interface* solo define un método [39]:

- ***create:*** Retorna la *CacheEntry* dados el objeto y su identificador.

El uso de una fábrica de entradas de caché separada permite tener una implementación genérica del caché.

CacheManager

El *CacheManager* es una clase de componente que mantiene parejas (identificador, entrada). A un identificador le corresponde una sola entrada. La *interface CacheManager* define cinco métodos [39].

- ***bind:*** Adiciona una entrada al caché.
- ***lookup:*** Busca una entrada en el caché.
- ***fix:*** Envía la señal que el objeto especificado va a ser usado de manera que no sea desalojado del caché.
- ***unfix*:** Envía la señal que el objeto especificado ya ha dejado de usarse.
- ***touch*:** Este método es llamado cuando un objeto ha sido accedido. Así, este método da pistas acerca de lo reciente o frecuente que ha sido su uso. Estas pistas son usadas dentro de los algoritmos de reemplazo.

UnbindManager

Esta *interface* define el manejador un método que es usado por el manejador de reemplazo para eliminar una entrada del *CacheManager*. En algunas implementaciones la entrada no es eliminada sino marcada para que sea recogida por el *garbage collector* de *Java*. Si la entrada es eliminada realmente, el *CacheManager* envía un evento *unbound* [39].

CacheAttributeController

Esta *interface* contiene métodos de configuración. Permite establecer el tamaño máximo del caché en términos del número de objetos o en términos del tamaño de la memoria. Además, tiene un mecanismo sencillo par especificar el número de entradas a desalojar. En esta *interface* se definen tres métodos [39].

- ***setMaxObjects:*** Configura el tamaño del caché.
- ***setMemorySize:*** Configura el tamaño del caché.
- ***setAutoCleanSize:*** Configura el número de elementos que deben ser eliminados cuando el caché está lleno o cuando se ejecuta *thread* el limpiador en *background*. Puede ser un valor absoluto o un porcentaje del tamaño máximo del caché.

Cambio en el tamaño del caché

Cuando el tamaño máximo del caché es cambiado a través de la *interface* de configuración, el *CacheManager* envía un CacheCapacityEvent a todos los sensores (*CacheCapacityEventListener*) registrados junto con el tamaño anterior y el nuevo [39].

Esta interface ofrece tres métodos:

- ***getEventId:*** Obtiene el identificador del evento ocurrido.
- ***getOldSize:*** Obtiene el tamaño anterior del caché.
- ***getSize:*** Obtiene el tamaño actual del caché.

Cuando se reduce el tamaño del caché, el *CacheManager* pregunta al *ReplacementManager* intenta desalojar una entrada usando el modo "mejor esfuerzo".

ReplacementManager

El rol de un *ReplacementManager* es escoger el conjunto de objetos que deben ser desalojados del caché cuando el espacio sea necesario para insertar nuevos objetos al caché. En el *ReplacementManager* se incluyen las políticas *LRU*, *MRU* y *FIFO*. Los principales métodos definidos en esta *interface* son [39]:

- ***addForReplacement:*** Debe ser usado este método para indicar que el *ReplacementManager* debe manejar una nueva entrada.
- ***adjustForReplacement:*** Este método debe ser usado cuando un objeto ha sido accedido. De este modo, da pistas sobre lo frecuente o reciente que ha sido su uso. Este método es usado por los algoritmos de reemplazo.
- ***forceFree:*** Este método obliga al *ReplacementManager* a liberar una o más entradas en el caché.

- ***unbind:*** Este método remueve una *CacheEntry* del caché. El parámetro bolean indica si la entrada debe ser removida o intentar ser removida. En el caso de una entrada *fix*, no debe ser desalojada.