Bunshin: DHT para aplicaciones distribuidas

Rubén Mondéjar, Pedro García, Carles Pairot
Dept. de Matemàtiques i Informàtica
Universitat Rovira i Virgili
43007 Tarragona
ruben.mondejar@estudiants.urv.es, {pedro.garcia,carles.pairot}@urv.net

Resumen

Presentamos Bunshin, una tabla de hash distribuida, la cual se construye sobre una red p2p estructurada y se caracteriza por su robustez y fiabilidad gracias a un sistema de replicación activo y un sistema de caching adaptativo.

Para dar soporte a otras arquitecturas middleware o aplicaciones finales, ofrece los servicios de inserción y recuperación de claves/valor de forma determinista, permite valores multicampo para cada clave y múltiples contextos para cada aplicación.

Encima de estos servicios se sitúa el motor de búsqueda. Basado en palabras clave, permite inserciones y consultas mediante estás. Además, incorpora mecanismos para establecer enlaces entre claves y notificaciones de nuevos enlaces sobre las claves que nos interesen.

1. Introducción

A lo largo de estos últimos años, la red Internet ha ido creciendo en número de usuarios. El ancho de banda de los nodos se ha visto incrementado considerablemente y la aparición de muchas aplicaciones de ámbito global ha popularizado el uso de la red de redes. Por otro lado, los ordenadores cada vez tienen una capacidad de cálculo mayor y las posibilidades de compartimiento de recursos han pasado a ser cada vez más y más importantes.

Un sistema basado en la arquitectura p2p no dispone de la figura de un nodo central que controle a los demás, tal y como ocurre en las arquitecturas *cliente-servidor* tradicionales. Los nodos actúan todos como *iguales*, de forma que ninguno de ellos asume, a priori, un mayor protagonismo que los otros. En este contexto, la computación p2p ha desplegado un gran número de aplicaciones orientadas a la comunicación y colaboración, así como computación distribuida. Una de sus características principales es la alta disponibilidad, gracias a la existencia de múltiples *peers* en un grupo, cosa que facilita que al menos un *peer* del grupo pueda satisfacer una petición de un usuario.

En la generación actual de redes p2p se pretende dar solución al problema comentado, de forma que la localización de los recursos sea **determinista**. Consecuentemente, si un recurso en concreto se encuentra en la red, entonces podremos obtenerlo. Para tal propósito, los nodos en la red se agrupan de manera **estructurada**, habitualmente, como **anillo** o **árbol**.

Los retos clave de estos sistemas son evitar los cuellos de botella que se pueden producir en determinados nodos y por tanto, se distribuyen las responsabilidades **de igual forma** entre los nodos existentes y adaptarse a las continuas entradas y salidas (y también caídas) de nodos.

Las redes *overlay* p2p son eficientes, resistentes a fallos, y auto-organizativas. Ejemplos existentes de este tipo de sistemas son Chord[1], Symphony[2], Pastry[3] o Kademlia[4]. Las redes *overlay* p2p descentralizadas reciben también el nombre de redes o sustratos **KBR** (*Key Based Routing*), ya que el enrutado de los mensajes es función de la clave de los nodos.

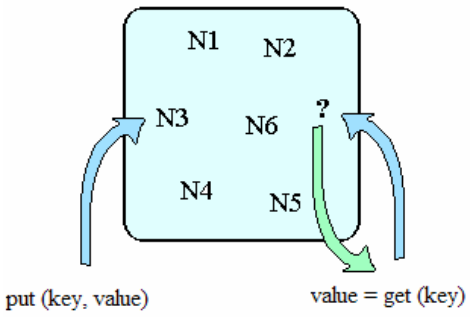


Figura 1. Interfaz básico de un DHT

Las redes KBR proporcionan servicios como tablas de hash distribuidas (DHTs). La abstracción que facilita una DHT con el interfaz estándar *put (key, value)* y *get (key)*, que podemos ver en la figura 1, es parecida a la utilización de las clásicas tablas de hash, pero en este caso, los *buckets* son los nodos físicos de la red, relacionando el rango de claves de éstos con los identificadores de los nodos a los que pertenecen.

Las DHT usan replicación para asegurar que los datos guardados sobrevivan a las caídas de los nodos y utilizan el caching para realizar copias transitorias y balancear la carga de peticiones sobre éstas.

En este artículo empezaremos describiendo el escenario donde nos situamos y a continuación expondremos la arquitectura de Bunshin, los servicios que ofrece y los resultados obtenidos. Por último, finalizaremos hablando de los trabajos relacionados con el tema, del trabajo futuro y de las conclusiones.

2. Escenario

El escenario donde se sitúa Bunshin es el proyecto Planet[5]. En lo que llevamos del mismo hemos desarrollado una serie de arquitecturas middleware para entornos distribuidos descentralizados. Bunshin nació de la necesidad de crear servicios de persistencia en Dermi[6]. Dermi es un middleware de objetos distribuidos construido encima de Pastry, que en primera instancia utilizaba el DHT de éste, PAST[7] para su servicio de directorio descentralizado.

Debido a que PAST no tenía una implementación fiable y que no era fácil realizar modificaciones, ya que había que readaptarlas a cada nueva actualización, decidimos implementar nuestro propio DHT.

Gracias a la experiencia como usuarios de un DHT al utilizar PAST y a las necesidades que vimos que necesitábamos, se implementó Bunshin (el cual esta construido sobre Pastry y al igual que éste se beneficia de los servicios que ofrece).

No sólo Dermi utiliza Bunshin, sino que a partir de entonces empezaron a surgir una serie de aplicaciones en el proyecto Planet que hacían uso de él. En la figura 2 podemos ver un ejemplo de la arquitectura típica de estas aplicaciones.

De estas podemos destacar **PlanetDR**, un servicio de repositorio descentralizado que utiliza el DHT para guardar la información sobre las comunidades y para realizar búsquedas sobre palabras claves.

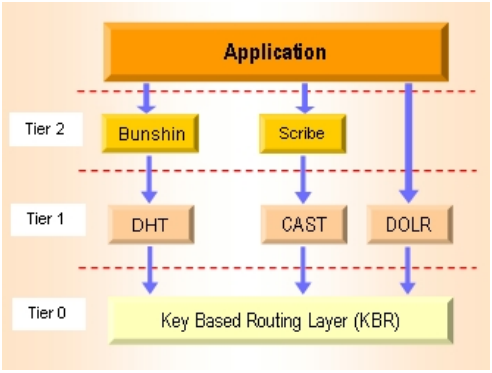


Figura 2. Arquitectura de tres capas en redes overlay

3. Arquitectura

La arquitectura de Bunshin se puede dividir en dos capas. La capa de aplicación DHT propiamente dicha y la capa de servicios que se construyen encima de ésta.

En la sección posterior se analizarán los servicios de la capa superior y veremos con detalle el diseño de la primera capa, comentando una a una las siguientes características:

- Sistema de replicación activo
- Esquema de caching adaptativo
- Diferentes modos de persistencia
- Valores multicampo
- Estructura multicontexto

3.1. Replicación

Para poder garantizar la fiabilidad en los datos que hay guardados en un entorno tan dinámico como las redes p2p, éstos no sólo necesitan estar en el nodo responsable, sino que debe haber más nodos que los contengan, de modo que si el nodo responsable deja la red por cualquier razón, sus datos no desaparecerán. Para conseguir este objetivo la información es replicada y gestionada de forma activa como ahora veremos.

Para poder decidir quienes serán los nodos que guardaran las replicas de una clave/valor

concreta, el nodo responsable elige a sus vecinos más cercanos, obteniendo la lista de sucesores de la red overlay. Con esto garantiza que si el nodo responsable cae, su inmediato sucesor ya tendrá los datos que necesitará Además, si el overlay que utilizamos agrupa los vecinos por proximidad, como es el caso de Pastry, las operaciones con éstos serán más eficientes.

Los candidatos a guardar las réplicas se obtienen mediante el método *replicaSet(),* que nos proporciona la red overlay, el cual nos retorna la lista de nodos sucesores ordenados de una clave en concreto. Gracias a ella también somos avisados de la entrada o salida de los nodos mediante el método *update()*. Además, para prevenir la perdida de mensajes o eventos realizan las siguientes comprobaciones:

1. Comprobar si todas las claves actualmente guardadas por un nodo aún le pertenecen a él, mediante la comprobación de si el nodo actual es el primer candidato que devuelve el método *replicaSet()*. Si descubre que algún par clave/valor ya no es responsabilidad suya lo reinsertará en la red, llegando así a su nuevo dueño.
2. Comprobar si los nodos en los que hay réplicas guardadas aún siguen activos, y además si el número de réplicas de cada clave es igual al número deseado. Sino se cumple alguna de las dos condiciones significa que hay que realizar más replicas y por tanto se escogerán los candidatos necesarios y se realizaran las réplicas que faltan.
3. Comprobación de que los nodos de los que tenemos réplicas aun siguen activos. Sino es así, se realizará el mismo procedimiento que cuando descubrimos que un par clave/valor ya no es nuestro. La diferencia en este caso es que con una probabilidad mucho más alta, al nuevo dueño le llegara más de una copia de esa clave/valor, ya que todas las réplicas se darán cuenta de ello. Para este caso, el criterio a seguir por el nuevo dueño es guardar el par clave/valor con un versionado más nuevo, siendo muy importante en este punto el **control de versiones** que se realiza en la actualización de los valores.

El control de versiones sigue una política atemporal, es decir, no se fija en la fecha en que fue actualizado el valor, ya que no podemos asimilar que todos los *peers* están sincronizados temporalmente. Para ello la técnica que se sigue es asignar un número de actualización sobre una copia viva en el sistema, siendo así fácil de reconocer si una replica nos envía un valor anterior al que ya tenemos. Si se trata de una copia sin número de versión asimilaremos que es una nueva actualización

3.2. Caching

La copia de claves/valor proporciona dos características importantes que son la fiabilidad y el rendimiento. Como ya hemos comentado, la fiabilidad y la tolerancia a fallos son proporcionadas por el sistema de replicación, así que para mejorar el rendimiento se utiliza la técnica de caching.

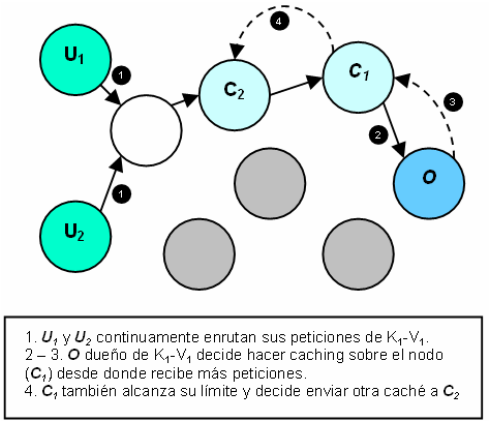
En este caso, y especialmente en entornos wide-area, algunos nodos pueden empezar a sufrir temporalmente sobrecarga de peticiones por clientes cercanos. Para solucionar este problema, se utiliza el **sistema de caching adaptativo**, la cual ahora pasamos ha explicar. El uso de estas caches es dinámico y adaptativo según el número de peticiones en un espacio de tiempo sobre cierta clave/valor, creándose así copias temporales que no mantienen el estado bajo demanda.

Para que el número de accesos a un solo nodo disminuya y se efectúe un balanceo de carga, el dueño de la clave/valor enviará una copia para que sea cacheada en el nodo inmediatamente anterior desde donde se recibe el número máximo de peticiones. De esta forma se mantendrá una copia temporal en caché y el dueño dejará de recibir peticiones de ese nodo durante el tiempo que dure la caché.

Si al nodo vecino que ahora tiene esta caché también le ocurriese lo mismo, pasado el límite, éste activaría una caché en el nodo vecino interesado. La idea es que la copia se vaya propagando en las caches de los nodos más interesados en ese momento, de forma que un nodo no sea saturado por ser dueño de una clave/valor muy popular, sino que se activen copias cerca de los nodos que realizan las peticiones.

Para poder interceptar las peticiones enrutadas en cada nodo, hacemos uso del método *forward()*. Este método que proporciona la red overlay sirve para decidir si el mensaje debe seguir

enrutándose. Así que nosotros además aquí es donde comprobamos si la clave de la petición tiene valor en caché en el nodo actual. Si la consulta es satisfactoria entonces el mensaje dejará de ser enrutado y se enviará el resultado directamente.



1. U_1 y U_2 continuamente enrutan sus peticiones de K_1-V_1.
2 – 3. O dueño de K_1-V_1 decide hacer caching sobre el nodo (C_1) desde donde recibe más peticiones.
4. C_1 también alcanza su límite y decide enviar otra caché a C_2

Figura 3. Activación adaptativa de caches.

En la figura 3, podemos ver un resumen de este sistema de caching adaptativo. También se puede resumir diciendo que se aplica una técnica *back-pressure* donde conseguimos que la caché se propague hacia el foco de interés en un determinado espacio temporal. Pasado un tiempo las caches caducan y si las ráfagas de peticiones ya han cesado, las caches irán desactivándose.

3.3. Modos de persistencia

Bunshin ha sido diseñado con la extensibilidad en mente y por tanto soporta distintos modos de persistencia:

- Memoria
- Disco
- *FileMapping*

El modo de **persistencia en memoria** es apropiado para los datos replicados o en caché y también para los datos de aplicaciones de prueba.

El modo estándar sería el **modo de persistencia en disco**, guardando todos los datos de un contexto en un directorio asignado. El más interesante ha destacar es el **FileMapping** que basándose en el anterior aporta mecanismos que permiten mapear los ficheros en el estado original en el cual fueron insertados en Bunshin.

Este modo de persistencia es muy útil para aplicaciones que quieran acceder a los ficheros directamente. Tenemos por ejemplo el caso de **Snap**, que es un portal descentralizado construido encima de una red p2p estructurada, que da soporte a aplicaciones web, donde una de ellas, un espacio de colaboración utiliza Bunshin como servicio de persistencia del servidor web y donde inserta los documentos, los cuales le interesa que sean accesibles mediante un cliente web.

3.4. Multicontexto y Multicampo

Bunshin no es un DHT convencional, en el sentido de estructuración clave/valor, sino que además proporciona, de forma opcional, la posibilidad de poder tratar los datos como si fueran una tabla de hash también, es decir, que para una clave podemos tener tantos valores como queramos, identificados con una subclave que llamamos **field**.

La otra posibilidad, a un nivel de abstracción más elevado, es poder ver un nodo no sólo como el contenedor de un bucket único, sino de tantos como necesitemos. Entonces para nosotros un bucket esta dentro de un **contexto**, y así en el caso de que la aplicación necesitara tener diferentes buckets no estará obligada a instanciar una aplicación Bunshin para cada uno de ellos, sino que sólamente tendrá que indicar en el contexto donde quiere trabajar.

En resumen, disponemos si es necesario N buckets con valores de M campos, dependiendo de las necesidades de las aplicaciones.

4. Motor de búsqueda

Como ya hemos comentado en la sección anterior, la arquitectura de Bunshin proporciona una capa superior donde se encuentran los servicios a parte de los básicos de DHT.

BunshinSearch se construye encima de Bunshin y utiliza los servicios explicados anteriormente para proporcionar los suyos:

- Servicios por palabras clave
- Enlaces entre claves
- Notificación de enlaces

A continuación detallaremos cada uno de estos servicios y el interfaz que proporciona cada uno.

Para los servicios por **palabras clave** se utiliza indexación mediante tablas de índices invertidos distribuidos. El interfaz que proporciona este servicio es parecida a la interfaz básica de un DHT pero introduciendo el concepto de palabra clave. El método *insert(keywords, key, value)*, se encarga de insertar la clave/valor y actualizar las tablas de índices invertidos correspondientes a las palabras clave. Por su parte, el método *query(keywords)* se utiliza para recuperar las claves que están ligadas al menos a una de las palabras clave de la consulta. Como resultado, se nos devolverán todas las claves que cumplan la condición, ordenadas por el número de coincidencias de palabras clave. Por último tenemos el método *remove(keywords, id)*, que elimina la clave/valor y además modifica las tablas de índices de cada palabra clave.

Aparte de PlanetDR, tenemos otras aplicaciones implementadas que hacen uso exclusivo de este servicio, como el caso de nuestro meta-buscador descentralizado. Esta aplicación realiza comprobaciones periódicas en busca de páginas web y sus contenidos, insertando la meta-información generada en Bunshin. El criterio de selección de palabras clave es parecido al de un meta-buscador convencional, seleccionando las palabras clave entre las palabras más repetidas en el documento o también entre las más significativas de los meta-datos, si dispone de de etiquetas con meta-información.

Gracias a los valores multicampo podemos diversificar el concepto clave/valor de forma que nos permite ofrecer el **servicio de enlaces**. Este servicio añade dos campos extras a una clave concreta. Estos dos nuevos campos tienen como valor las listas de enlaces de entrada y enlaces de salida para su clave correspondiente.

Para añadir los enlaces tenemos el método *addLinks (key,links)* y *removeLinks(key, links)* para eliminarlos, tratando automáticamente tanto el enlace de entrada como el de salida. Para la consulta de los enlaces tenemos los métodos *getIncomingLinks(key)*, enlaces de entrada de una clave, y *getOutgoingLinks(key)* para los enlaces de salida.

Siguiendo con el ejemplo anterior del meta-buscador, también a partir de la lectura de los anchor de la web, BunshinSearch puede enlazar las claves de los meta-datos insertados. A la hora de retornar resultados gracias a poder consultar la información de enlaces de cada web, podrá además modificarlos añadiendo el criterio de número de enlaces o de enlaces importantes, realizando previamente un cálculo estimado parecido al *PageRank* de Google[8].

Por último y complementando el servicio anterior, tenemos el **servicio de notificación de enlaces**. Este servicio ofrece a las aplicaciones la posibilidad de ser notificado cuando una clave es enlazada. Como resultado, primero se realiza la petición de notificación sobre la clav A, y a continuación, al añadir el enlace de la clave B hacia la clave A, se realiza la notificación correspondiente a este evento.

Siguiendo el modelo de delegación estándar de Java, los métodos de este servicio son, *setLinkListener(id,listener)* para asignar un notificador y *removeLinkListener(id,listener)* para eliminarlo. Tal hecho nos permite estar informados sobre los enlaces de entrada que se añaden a una clave en particular.

Este servicio es útil para las aplicaciones donde nos interesa tener noticias sobre enlaces a valores concretos, como por ejemplo una aplicación parecida a OverCite[9], una aplicación basada en DHT donde el foco de interés se centra en la publicación de artículos científicos.

5. Validación

Bunshin es un sistema bastante maduro ya que ha sido probado en multitud de aplicaciones y hemos realizado tanto simulaciones como pruebas experimentales en PlanetLab[10].

Para validar la fiabilidad que proporciona el sistema de replicación de Bunshin, hemos simulado la creación de una red de 1000 nodos e insertado 1000 pares clave/valor aleatoriamente, tanto la clave como el nodo desde donde insertamos. Para esta simulación hemos utilizado nuestro propio simulador PlanetSim[11] usando una red overlay Chord. Para cada clave/valor se realizan seis réplicas (incluyendo la copia principal en el nodo sucesor). Después de las inserciones, hacemos que una fracción de nodos falle y caiga de la red, sin posibilidad de avisar ni de realizar el movimiento de claves. Podemos ver en la figura 4 que estos resultados son muy

parecidos a los de CFS[12], validando nuestra aproximación.

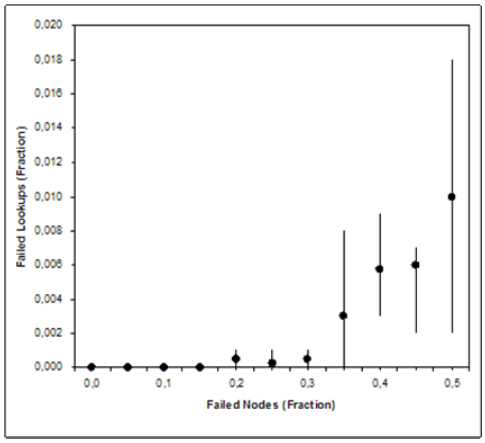


Figura 4. Porcentaje de peticiones no satisfechas frente al porcentaje de nodos caídos. Indicando la barra de error el mínimo y el máximo, y marcando la media, de las 5 pruebas realizadas.

Podemos observar que no empiezan a haber fallos en las consultas hasta que el 20% de los nodos ha caído y también que los fallos son muy pocos hasta que los nodos caídos no ascienden a 35%. Los fallos empiezan a ser considerables cuando se pierden las seis copias realizadas. Por ejemplo, si el 50% de los nodos cae, la posibilidad de perder completamente una clave/valor es muy baja, $0.5^6 = 0.0016$, parecido a los resultados obtenidos en la prueba, la media de los cuales, es de 0.010.

6. Trabajo relacionado

CFS es un sistema de fichero cooperativo construido encima de la red overlay Chord. Ofrece los servicios básicos de un DHT, y mantiene réplicas y caches. CFS esta orientado al almacenamiento de bloques y encima de el se construye un sistema convencional de ficheros similar al sistema UNIX. Cada bloque es guardado en múltiples nodos Chord que esta focalizado en almacenar bloques de datos. No ofrece otros servicios como podrían ser un motor de búsqueda, aunque si que utiliza un sistema de autentificación de clave pública.

PAST sigue una filosofía parecida a CFS, pero en el *overlay* Pastry. Intenta mejorar el sistema de CFS en almacenamiento de ficheros y *caching*, ayudándose de los servicios que ofrece el *overlay* Pastry, como son una mayor eficiencia y proximidad de las claves. Ofrece un sistema de ficheros de certificados para las operaciones básicas. Tampoco ofrece otro tipo de servicios.

La principal diferencia con CFS y PAST, es que Bunshin no está tan focalizado en ofrecer un servicio de sistemas de ficheros distribuido, sino de ofrecer más variedad de servicios que son interesantes a la hora de utilizar un DHT desde el punto de vista del programador de aplicaciones.

7. Trabajo futuro y conclusiones

En este artículo hemos presentado Bunshin, el cual da un enfoque diferente a las DHT convencionales, con las características de un sistema de replicación activo y sistema de caching adaptativo, dotándola de más flexibilidad, como son las características de multicontexto y multicampo, y con un servicio de búsquedas de palabras clave, de enlazado de claves, y de notificación de nuevos enlaces.

Aparte de los servicios ofrecidos por Bunshin que extienden la interfaz clásica de los DHTs, también se puede extender o ampliar con otras funcionalidades futuras. Un ejemplo podría ser que para el servicio de notificación de enlaces si se complementara con un servicio de propagación de eventos, como por ejemplo Scribe[13], podría hacer que un solo evento de este servicio se propagara eficientemente en un grupo de *peers* interesados.

Otro punto a remarcar es el servicio de seguridad que ofrece Bunshin a la hora de realizar las operaciones básicas. Este servicio es idóneo para dotar de privacidad los pares claves/valor tanto en las inserciones, como en las modificaciones o en las consultas de éstas. De momento, utilizamos un sistema de verificación con clave simétrica pero en breve se empezará ha utilizar un esquema de clave asimétrica para encriptar y desencriptar estos valores con las claves públicas y privadas de los usuarios.

Además hemos realizado, y seguimos realizando, numerosas pruebas en PlanetLab para

depurar y probar correctamente todas las funcionalidades y asegurar así su buen funcionamiento. Creemos que Bunshin proporciona interesantes y novedoso servicios al programador de aplicaciones p2p descentralizadas.

Agradecimientos

Financiado por el Ministerio de Ciencia y Tecnología, TIC2003-09288-C02-00

Referencias

[1] Stoica I., Morris R., Karger D., Kassshoek M.F. y Balakrishanan H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. ACM SIGCOMM, pp. 149-160. Agosto 2001.

[2] Manku G.S., Bawa M. y Raghavan P. Symphony: Distributed Hashing in a Small World. USENIX Symposium on Internet Technologies and Systems (USITS). Marzo 2003.

[3] Rowstron A. y Druschel P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp 329-350. Noviembre 2001.

[4] Maymounkov P. y Mazières D. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. International Workshop on Peer-to-Peer Systems (IPTPS), pp 53-65. Marzo 2002.

[5] Pairot .C, García P., Rallo R., Blat J. y Skarmeta A.F. The Planet Project: Collaborative Educational Content Repositories on Structured Peer-to-Peer Grids. ACM/IEEE CCGrid 2005. Second International Workshop on Collaborative and Learning Applications of Grid Technology and Grid Education (CLAG). Mayo 2005.

[6] Pairot C., García P. y Skarmeta A.F. Dermi: A New Distributed Hash Table-based Middleware Framework. IEEE Internet Computing. Vol 8, No. 3, pp. 74-84. Mayo/Junio 2004.

[7] Rowstron A. y Druschel P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. ACM Symposium on Operating Systems Principles (SOSP). Octubre 2001.

[8] Brin S. y Page L. The anatomy of a large-scale hypertextual web search engine. International World Wide Web Conference (WWW). Abril 1998.

[9] Stribling J., Councill I. Li J., Kaashoek M.F., Karger D.R., Morris R. y Shenker S. OverCite: A Cooperative Digital Research Library. International Workshop on Peer-to-Peer Systems (IPTPS). Febrary 2005.

[10] PlanetLab Website, http://www.planet-lab.org/

[11] García P., Pairot C., Mondéjar R., Pujol J., Tejedor H. y Rallo R. PlanetSim: A New Overlay Network Simulation Framework. Lecture Notes in Computer Science (LNCS), Software Engineering and Middleware (SEM).Vol. 3437, pp. 123-137. Marzo 2005.

[12] Dabek F., Kaashoek M. F., Karger D., Morris R. y Stoica I. Wide-area cooperative storage with CFS. Proceedings ACM Symposium on Operating Systems Principles (SOSP). Octubre 2001.

[13] Castro M., Druschel P., Kermarrec A.M. y Rowstron A. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No 8. Octubre 2002.

[14] Bunshin Website, http://ants.etse.urv.es/bunshin