Universitat de Lleida

Computación Distribuida en Entornos Peer-to-Peer con Calidad de Servicio

Damià Castellà Martínez

Escola Politècnica Superior
Departament d'Informàtica i Enginyeria Industrial

Computación Distribuida en Entornos Peer-to-Peer con Calidad de Servicio

Memoria presentada para lograr el grado de
Doctor por la Universitat de Lleida
por

Damià Castellà Martínez

Dirigida por

Francesc Giné de Sola **Francesc Solsona Tehàs**
Universitat de Lleida

Programa de doctorado en Ingeniería

Lleida, mayo de 2011

Prólogo

Actualmente, las necesidades de cálculo computacional crecen día tras día. Diferentes ámbitos de la ciencia (climatología, medicina, física....) plantean problemas que requieren grandes capacidades de cómputo, que gestionan y procesan cantidades enormes de datos. Estas necesidades de cómputo han comportado una continua evolución de los sistemas de computación logrando unas cotas de rendimiento impensables hasta hace poco tiempo.

La potencia de cómputo de los monoprocesadores actuales han alcanzando los límites físicos de estabilidad, ya sea por el factor de integración, la disipación de calor, etc. Este hecho ha motivado que la tendencia actual en la industria de los procesadores se dirija a los sistemas multiprocesador; donde desde hace unos años (aproximadamente desde 2005) se están integrando múltiples procesadores en un mismo chip (Dual Core, Quad Core, etc), o bien desde hace más tiempo, centros de investigación y organizaciones especializadas emplean sistemas distribuidos compuestos por muchos computadores que interaccionan entre ellos, mediante una conexión en red (Clusters, Grid y Peer-to-Peer).

Las características de cómputo y conectividad de los sistemas informáticos han inducido, en los últimos años, al surgimiento de un conjunto de aplicaciones que operan bajo el modelo de cómputo distribuido, denominado Peer-to-Peer (P2P). La computación P2P representa una alternativa emergente de bajo coste para proporcionar acceso a los recursos de cómputo distribuidos, de forma escalable y tolerante a fallos. Las arquitecturas P2P se aprovechan de la infrautilización de los recursos de cómputo, integrándolos en una plataforma de compartición de recursos entre iguales. Estas plataformas utilizan los ciclos de procesador y otros recursos ociosos de miles de ordenadores conectados a través de Internet, para proporcionar altas prestaciones a aplicaciones

paralelas/distribuidas.

La computación distribuida P2P, paradigma en el que se enmarca esta tesis, ha comportado el nacimiento de nuevas técnicas de programación respecto al modelo secuencial tradicional, que permite el tratamiento y procesado de problemas hasta la fecha inimaginables, y su resolución en un tiempo acotado. Asimismo, ha comportado la necesidad de crear y establecer nuevos entornos y políticas de gestión de recursos, que permitan el uso eficiente de estos recursos distribuidos a lo largo de la red. Este paradigma ha sido la motivación y objeto de estudio de la presente tesis.

La propuesta que se presenta en el presente trabajo se centra en el diseño y desarrollo de una arquitectura de cómputo distribuido en entornos Peer-to-Peer con calidad de servicio, llamada DisCoP (Distributed Computing P2P Platform). La filosofía de esta arquitectura consiste en crear una red global de recursos computacionales, donde todos los nodos son tratados por igual y con capacidad de poder compartir y controlar más de un recurso por nodo, como por ejemplo, el procesador, la memoria principal, la memoria secundaria...

A diferencia de otros sistemas de computación distribuida P2P ya desarrollados como *Boinc*, *Folding@home* o *SETI@home*, los cuales solo permiten ofrecer recursos para aplicaciones especificadas por el proveedor, éste sistema pretende ofrecer al usuario la posibilidad de ejecutar sus propias aplicaciones paralelas/distribuidas, convirtiendo la computación P2P en una alternativa al paradigma de computación distribuida tradicional, sin ningún coste adicional.

La arquitectura propuesta, compuesta por tres niveles, posee propiedades comparables a otras plataformas y overlays de la literatura. En muchos aspectos supera las prestaciones de topologías ampliamente utilizadas actualmente, como es el caso de Chord, Baton, Butterfly, etc. Esta arquitectura asegura, mediante el primer nivel, la ordenación de los peers según la potencia de sus recursos computacionales. La calidad de servicio se garantiza en el segundo nivel, puesto que al disponer de una topología de tipo estructurado, los tiempos de respuesta están acotados. Además ofrece un amplio rango de caminos alternativos para localizar recursos, hecho que aumenta las prestaciones del sistema ya que de esta forma evita la saturación y los cuellos de botella del sistema.

Para optimizar los costes de mantenimiento de la red, los nodos se auto-organizan y colaboran entre sí de forma descentralizada, evitando la necesidad de que el sistema sea controlado de forma centralizada por un peer o por un conjunto reducido de peers, hecho que aumentaría de forma muy significativa la probabilidad de fallo total o parcial del sistema.

En este trabajo se presentan mecanismos esenciales que garantizan la integridad y conectividad del sistema, funcionalidades como la inserción, desconexión de peers y el mantenimiento de la red que son mecanismos que garantizan la robustez del sistema, ofreciendo calidad de servicio aún cuando se produzcan fallos. Todos estos servicios están integrados en la arquitectura propuesta. Asimismo se define un nuevo procedimiento para calcular la *Localidad*, métrica usada para medir el grado de proximidad entre dos nodos con características similares. Finalmente, se realiza un análisis experimental de la robustez de la plataforma, su tolerancia a fallos y su localidad.

La búsqueda eficiente de recursos computacionales es un gran reto y a su vez, un requerimiento imprescindible que ha de proporcionar la plataforma. Esto comporta que no solo se deben localizar con precisión los recursos computacionales solicitados por los usuarios de la plataforma, sino que dichas búsquedas han de ser eficientes. En esta tesis se ha propuesto también diferentes algoritmos de búsqueda, cada uno de ellos adaptándolos y optimizándolos para localizar recursos computacionales de diferentes formas, exacta, aproximada, multi-atributo y por rangos. Tanto la arquitectura propuesta como las funcionalidades que ofrece la plataforma DisCoP, están diseñados para ofrecer garantías de calidad de servicio.

Asimismo, esta plataforma ofrece la posibilidad de diseñar nuevas políticas de planificación de tareas acorde a las necesidades de cómputo de las aplicaciones distribuidas.

Agradecimientos

Deseo expresar mi más sincero agradecimiento al Dr. Francesc Giné de Sola y al Dr. Francesc Solsona Tehàs por su magistral dirección y constante dedicación en la supervisión del trabajo, por la gran ayuda recibida en la redacción de artículos y el apoyo moral que me han ofrecido en todo momento.

Un especial agradecimiento al resto de profesores del Grupo de Computación Distribuida (GCD) de la Universitat de Lleida (UdL), Concepció Roig, Fernando Cores, Fernando Guirado, Josep Lluís Lérida, Josep M. Solà, Valentí Pardo, Albert Saiz y Xavier Faus. Por haberme hecho sentir uno más del grupo, tanto a nivel de trabajo como a nivel personal.

A Héctor Blanco, compañero y amigo de despacho, por su compañerismo, escuchando y aguantando mis rollos, altibajos y sueños imposibles. A Ivan Teixidó por el gran soporte técnico ofrecido. Ignasi Barri y Josep Rius, que me suplieron en mis dos primeros congresos. A Miquel Orobitg por los buenos ratos que hemos pasado juntos. Y finalmente un especial agradecimiento a los últimos compañeros predoctorales recién llegados, Anabel Usié y Alberto Montañola, unos grandes amigos.

Dedico también un especial agradecimiento a mis amigos de Cervera, Marc Freixes y Xavier Cañabate por el apoyo moral que me han ofrecido muchísimas veces.

Y como no, a toda mi familia. A mis padres, mis hermanos, mis tíos porque a pesar de mi gran dedicación a la tesis, siempre han sabido entender que ellos,

para mi, son lo más importante.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

En este capítulo se introduce al lector en la definición, conceptos y características de las principales redes Peer-to-Peer. De este modo, se definen los conceptos principales de estas redes, sus requerimientos y además se realiza una clasificación según el tipo de conectividad, el grado de descentralización, y su aplicación en la industria y en la investigación. Finalmente se introducen las topologías Peer-to-Peer más significativas y utilizadas en el ámbito de computación en entornos P2P. Una vez explicados todos estos conceptos, se presentan los objetivos principales de la tesis doctoral y la estructura de esta memoria.

1.1. Computación Distribuida

Actualmente, las necesidades de cálculo crecen día tras día. Distintos ámbitos de la ciencia (biología, matemáticas, climatología, medicina, física, ...), ingeniería (materiales, energía, informática,...) e industria (automovilística, farmacéutica, química,...), plantean problemas que requieren gran capacidad de cómputo y que gestionan enormes cantidades de datos. En determinadas aplicaciones, la potencia de cálculo y los recursos necesarios para solucionar estos problemas resultan intratables en un sistema monoprocesador.

Algunos ejemplos concretos de este tipo de problemas son:

- Análisis de modelos físicos.

- Problemas de mecánica aplicada.
- Simulaciones moleculares y atómicas.
- Simulaciones geofísicas.
- Proceso y visualización de grandes cantidades de datos.

Una de las posibles soluciones para poder tratar este tipo de problemas es la utilización de supercomputadores. Un supercomputador es un computador formado por muchos procesadores que trabajan conjuntamente, disponiendo de este modo de una gran capacidad de cálculo.

Sin embargo, los supercomputadores tienen sus limitaciones, como es su coste de fabricación y mantenimiento, además de poseer una estructura poco flexible o escalable.

Para solucionar estas limitaciones, existe el modelo de *Computación Distribuida*, basada en el uso de un gran número de computadoras distribuidas y conectadas entre si mediante una infraestructura de red.

De la aplicación de este modelo han surgido las siguientes estructuras de cómputo:

- **Cluster:** Consiste en un conjunto de computadoras, normalmente homogéneas, conectadas mediante redes locales de alta velocidad LAN[1].
- **Grid:** Es una tecnología que permite utilizar de forma coordinada todo tipo de recursos de cómputo, almacenamiento y aplicaciones específicas, que no estén sujetos a un control centralizado. En este sentido, los recursos heterogéneos (supercomputadores, clusters, multi-clusters,...) se encuentran conectados mediante redes de área extendida, como es el caso de Internet.
- **Computación Peer-to-Peer:** Consiste en el uso de una red de computadoras (denominada también red de pares, red entre iguales, red punto a punto o más conocida con las siglas P2P, en ingles Peer-to-Peer) de área extensa (p.e. Internet) donde los nodos se comportan como iguales

[1]LAN: Local Area Network

entre sí. Los nodos pueden actuar simultáneamente como clientes y servidores permitiendo realizar tareas de forma distribuida, escalable y eficiente. La diferencia principal entre la computación Grid y P2P, es que un Grid es un entorno dedicado formado por infraestructuras muy caras adquiridas únicamente por grandes organizaciones, empresas o universidades; mientras que la computación Peer-to-Peer está sujeta solamente al uso de una aplicación concreta y de las máquinas de los usuarios que integran la plataforma P2P. El coste energético extra de una plataforma P2P es nulo, mientras que los sistemas Grid son unos ávidos consumidores energéticos. Asimismo, los entornos Grid son más estáticos y complejos, requiriendo de una sofisticada configuración y mantenimiento por parte de un administrador altamente cualificado técnicamente; y en cambio las redes P2P son más simples, dinámicas y están solo bajo el control de la propia aplicación P2P.

La tesis doctoral descrita en el presente documento se enmarca en el ámbito de la computación P2P.

Con objeto de mostrar las posibilidades que ofrecen los recursos disponibles en entornos de cómputo distribuido P2P, a continuación se explica un simple ejemplo que cuantifica la potencia desaprovechada de estos sistemas: "Los computadores domésticos tienen una potencia de cómputo superior a los requerimientos computacionales necesarios para la mayoría de sus usuarios. Actualmente, un equipo medio de un usuario estándar dispone de dos núcleos de procesador con una potencia media de 6 GigaFlops y 3 Gigabytes de Memoria RAM y además con alta conectividad a Internet. Acorde con diferentes estudios [AES97], la mayor parte del tiempo que estas máquinas están conectadas, solo se utilizan como mucho, un 50 % de su capacidad. Por lo tanto, si se pudieran aprovechar los recursos ociosos de 100 máquinas como las mencionadas, se dispondría de un total de 300 GigaFlops de procesamiento y 150 Gigabytes de RAM". Este simple ejemplo refleja la potencia ociosa de la cual se podría sacar provecho si se utilizasen en segundo plano todos los recursos de los computadores domésticos. El objetivo de esta tesis es diseñar una infraestructura P2P que permita usar de un modo eficiente los recursos ociosos distribuidos a lo largo de la red global Internet.

1.2. Conceptos Previos de Peer-to-Peer

En esta sección se explican los conceptos generales utilizados en el campo de las redes Peer-to-Peer, con objeto de ayudar a entender la solución propuesta en los capítulos posteriores. También se describen sus principales características y se realiza una clasificación general según la estructura y capacidad de distribución de las redes. Asimismo se comentan las aplicaciones y ventajas de esta tecnología, profundizando en las técnicas de búsqueda de los recursos en este tipo de entornos [Tej06].

1.2.1. Elementos de una red Peer-to-Peer

Los nodos de una red P2P pueden ser de dos tipos:

- **Peers simples:** Son nodos conectados a la red P2P. Son los propietarios de los recursos computacionales y de los servicios, que a su vez pueden ser compartidos entre todos o varios nodos de la red. Los peers suelen tener una naturaleza muy dinámica, conectándose a la red de forma intermitente, entrando y saliendo arbitrariamente. Además son muy heterogéneos, es decir, tienen capacidades de procesamiento, ancho de banda y almacenamiento, muy distintos.
  A nivel de conectividad, suelen tener un nivel de accesibilidad limitado porque normalmente están conectados detrás de un cortafuegos y/o un servidor NAT. En consecuencia, son el tipo de nodos de menor responsabilidad de la red P2P.

- **Superpeers o Rendevouz Peers:** Son nodos con un rol especial que poseen gran capacidad computacional y/o conectividad. Éstos ayudan a que los peers simples encuentren a otros peers o a otros recursos computacionales. Los peers envían solicitudes de búsqueda de recursos a los superpeers, y éstos les indican dónde conseguirlos. Normalmente, cada superpeer controla un grupo determinado de peers simples y/o gestionan una memoria caché con las últimas búsquedas efectuadas.
  A nivel de conectividad, los superpeers no se encuentran detrás de un

cortafuegos y tienen una dirección IP pública fija. Los superpeers también pueden ayudar a que los peers simples se puedan comunicar entre ellos, si estos están detrás de un cortafuegos o un enrutador NAT[2].

1.2.2. Características de las redes Peer-to-Peer

Las aplicaciones P2P tienen unos requerimientos muy exigentes, los cuales determinan a su vez las principales características de las redes P2P: descentralización, escalabilidad, anonimato, propiedad compartida, conectividad ad-hoc, rendimiento, seguridad, tolerancia a fallos e interoperabilidad.
A continuación se explican detalladamente estas características deseables en las redes P2P:

- **Descentralización:** En el modelo cliente-servidor tradicional, la información se concentra en los servidores. Los usuarios suelen acceder a esta información mediante programas cliente. De este modo, los sistemas centralizados son ideales para ciertas aplicaciones y tareas; por ejemplo, la publicación de un portal de información web en Internet, servicios de juegos on-line, búsqueda de recursos, etc. Sin embargo, la topología de los sistemas centralizados lleva inevitablemente a ineficiencias del servicio, cuellos de botella, recursos infrautilizados y ataques *DoS*[3]. En las redes P2P, todos los nodos son iguales y no existe una visión global de todos los recursos que éstos proporcionan, hecho que dificulta enormemente la implementación y el control del sistema. Encontrar recursos en la red es también muy complicado. Sin embargo, las ventajas (responsabilidad distribuida, mayor capacidad de recursos, coste de mantenimiento bajo, fomento de grupos sociales y de trabajo) que conlleva la descentralización, compensa la utilización de estas redes.

- **Escalabilidad:** Un ventaja inmediata de la descentralización es la mejora de la escalabilidad. Por escalabilidad se entiende la capacidad que

[2]NAT (Network Adress Translation): Es un mecanismo utilizado por enrutadores IP para intercambiar paquetes entre dos redes que se asignan mutuamente direcciones incompatibles.
[3]Ataque *DoS* (Denial of Service): ataque que satura un servidor mediante el envío masivo de peticiones de servicio.

tienen las redes P2P para albergar un número de nodos ilimitado en su sistema, sin afectar el tiempo de respuesta de la red y conservando las prestaciones. Las redes P2P tienen un alcance mundial, con cientos de millones de usuarios potenciales. En general, el objetivo es que puedan soportar un número ilimitado de nodos, asegurando una calidad de servicio mínimo.

- **Anonimato:** El anonimato permite a los usuarios utilizar un overlay de comunicación sin preocuparse de cuestiones legales o de otro tipo, como ataques a su intimidad. Existen varias técnicas para alcanzar el anonimato en las redes P2P, como la creación de grupos de multicasting para que el contenido de un mensaje no sea interceptado, o que el receptor de un mensaje no pueda ser identificado, ocultación de la IP e identidad del emisor, comunicación empleando nodos intermedios a pesar de que sea factible contactar directamente con el destinatario, o ubicación involuntaria y fragmentada de los contenidos.

- **Propiedad compartida:** Una de las premisas de la computación distribuida P2P es la propiedad compartida. La propiedad compartida reduce el coste de gestión de los sistemas y contenidos, así como el de mantenimiento. El coste del sistema global se ve reducido además, porque se aprovecha de las capacidades de cálculo, almacenamiento y ancho de banda ociosas. Esto es aplicable a cualquier tipo de sistema P2P, aunque quizás sea más obvio en los sistemas de computación distribuida. Por ejemplo, *SETI@home* [SET99] registró el año 2009 una potencia superior a los 769 TeraFlops, no muy lejos del supercomputador más potente del mundo *Cray Jaguar*, que tenía registrados 2331 TeraFlops en el mismo año. Por otro lado, en los sistemas P2P de colaboración y comunicación, como *Groove* o *Skype*[Gro, Sky] respectivamente, la eliminación de servidores centrales para el almacenamiento de la información y la gestión de las interacciones entre nodos, reduce enormemente los costes de mantenimiento asociados.

- **Conectividad Ad-hoc:** La naturaleza ad-hoc de la conectividad en entornos P2P significa que los usuarios no tendrán disponible en todo

momento los recursos deseados. Esto depende del tipo de sistema P2P. Por ejemplo, en el caso de la computación distribuida, las aplicaciones paralelizables no pueden ser ejecutadas en todos los nodos al mismo tiempo dado que algunos de los nodos se encontrarán disponibles durante la mayor parte del tiempo y otros solamente de forma intermitente.
En aplicaciones P2P de compartición de ficheros, los usuarios esperan poder acceder a los distintos contenidos intermitentemente, sujetos a la conectividad de los proveedores de dichos contenidos. No obstante, la conectividad puede mejorarse mediante *la replicación de contenidos* en nodos auxiliares. Por ejemplo, el sistema de almacenamiento de Popular Telephony, *PeerioData* [Pee], manejó esta situación fragmentando y encriptando los ficheros a guardar y distribuyendo copias entre varios nodos.

- **Rendimiento:** Los sistemas P2P pretenden mejorar el rendimiento en varios aspectos. En primer lugar, mejoran la calidad de sus servicios agregando capacidad de almacenamiento distribuido (*Kazaa* [ZF]) y ciclos de computación (*Distributed.net* [dis]) a medida que van entrando peers. En segundo lugar, mejoran el tiempo de respuesta de una petición de búsqueda, ya que incluso aumentando el número total de nodos en el sistema, la distancia máxima entre dos nodos tenderá a ser de orden logarítmico. En tercer lugar, se ha probado que los sistemas que incorporan superpeers ofrecen un rendimiento superior, dado que solo un subconjunto de peers controlan la información de todos sin saturar el sistema y por lo tanto, la información solicitada se localiza más rápidamente. Además existen tres mecanismos claves para optimizar el rendimiento general: la *replicación*, el *almacenamiento caché* y el *encaminamiento inteligente*. La *replicación* se entiende como un mecanismo de seguridad que, como su mismo nombre indica, replica los datos más populares de los peers entre nodos de la red, con el fin de asegurar la conservación de la información más importante y al mismo tiempo servir las solicitudes según la cercanía entre los nodos solicitantes y los servidores. El *almacenamiento caché* es una técnica empleada para guardar las direcciones de búsqueda temporalmente entre diferentes nodos, para ser localizadas

con mayor rapidez en nuevas peticiones de búsqueda. El *encaminamiento inteligente* consiste en ubicar los nodos con intereses comunes en zonas próximas para reducir el número de mensajes y la latencia de la red.

- **Seguridad:** La seguridad en entornos P2P es una característica crítica, acorde con su naturaleza. Los sistemas actuales proporcionan seguridad aplicando relaciones de confianza entre nodos y objetos distribuidos, mediante esquemas de intercambio de claves de sesión, con objeto de evitar los nodos maliciosos y la manipulación de los resultados de ejecución de una tarea distribuida; asegurando la protección de los recursos de la red. No obstante, es necesario nuevos mecanismos de protección como: encriptación multi-clave, *sandboxing* o caja de arena, gestión de derechos digitales, reputación, etc...

- **Tolerancia a fallos:** Un objetivo básico de diseño de una red P2P es que ésta no pierda su funcionalidad debido a fallos asociados a desconexiones de nodos o caídas de la red. El gran problema relacionado con la desconexión es la pérdida de disponibilidad de los recursos. El gran dinamismo que caracteriza un sistema Peer-to-Peer (conexiones y desconexiones de nodos de forma continua y sin previo aviso) y el gran número de componentes que pueden formar parte de éstos, hace que la posibilidad de que se produzca un fallo sea elevada. Por lo tanto, son necesarios mecanismos que eviten o reparen los fallos que puedan acontecer. Por ejemplo, implementar protocolos de encaminamiento de tráfico cuando el camino hacia un nodo destino esté cortado o aplicar mecanismos de replicación de recursos clave, puede aliviar el problema.

- **Interoperabilidad:** A pesar de la existencia de muchos sistemas P2P, la mayoría de ellos no son interoperables, es decir, no son capaces de comunicarse y entenderse entre sí. La razón de esta falta de interoperabilidad ha sido la inexistencia de una entidad de estandarización mundial, como el IEEE (Institute of Electrical and Electronics Engineers) o la OMG (Object Management Group). No obstante, se ha mejorado gracias a la formación del grupo de trabajo en sistemas P2P *Internet2* [Gru96] y a la

aparición del Proyecto *JXTA* [ABJM04, AJN05]. El objetivo del grupo de trabajo P2P Internet2 es reunir la comunidad de desarrolladores de aplicaciones P2P y establecer un grupo común para establecer especificaciones que posibiliten el entendimiento común entre todas ellas. El Proyecto JXTA es un importante esfuerzo para conseguir una librería de código abierto (API) para el desarrollo de aplicaciones P2P, intentando así convertir la arquitectura en un estándar de facto.

1.2.3. Redes P2P estructuradas vs. no estructuradas

Las arquitecturas P2P se pueden clasificar en estructuradas y no estructuradas. A continuación se describe esta clasificación.

1.2.3.1. Arquitecturas no estructuradas

Se forman cuando los enlaces de la red se establecen de forma arbitraria. Estas redes se construyen fácilmente. Son redes resistentes a la constante entrada y salida de peers ya que no necesitan mecanismos para reestructurar y mantener la topología y además soportan ataques *DoS*. Este tipo de redes también soportan consultas complejas.
El servicio de búsqueda de recursos en estas redes se basa en que cada petición tiene que recorrer todo o parte del sistema con la finalidad de encontrar uno o más peers que compartan estos recursos. La desventaja principal de este tipo de redes consiste en que las peticiones no siempre pueden ser resueltas. Un contenido popular es muy probable que esté disponible en varios peers del sistema, por tanto, cualquier búsqueda de este tipo de contenidos resultará exitosa. Por otro lado, si un peer realiza una búsqueda de algún recurso o dato poco popular, es decir, que pocos peers alberguen dicho contenido, es muy probable que la búsqueda no tenga éxito o que requiera un elevado tiempo de búsqueda.
El rendimiento de estas redes depende del mecanismo de búsqueda empleado. Si la búsqueda es a ciegas, entonces tardará más en encontrar resultados y por lo general la red será poco escalable (punto débil). Las búsquedas a ciegas normalmente ocasionan "*flooding*", definido como inundación de la red debido

a la gran cantidad de tráfico que circula por ella. Sin embargo, si la búsqueda se basa en una heurística, será más rápida y la red podrá albergar más nodos.

Algunos ejemplos de redes peer-to-peer no estructuradas son: *Kazaa* y *Gnutella* [ZF, KC04]. En la Fig. 1.1 se puede ver un esquema de una red peer-to-peer de tipo no estructurada. Muestra un ejemplo de un peer que realiza una consulta "xyz". El mensaje se propaga a ciegas a través de los enlaces y de forma recursiva a todos los nodos vecinos hasta encontrar el nodo con el resultado deseado. Después, éste responde enviando la respuesta al nodo solicitante.

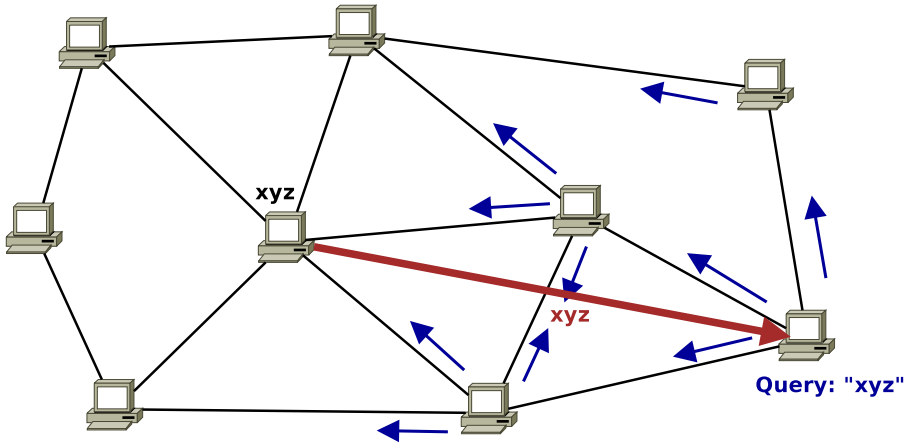

Figura 1.1: Esquema de una red peer-to-peer de tipo no estructurada.

1.2.3.2. Arquitecturas estructuradas

Las redes estructuradas son sistemas P2P donde los nodos guardan información útil de encaminamiento (routing) para dirigir las búsquedas. Gestionan una Tabla Hash Distribuida (*DHT*[4]) y permiten que cada peer sea responsable de una parte específica del contenido de la red. Estas redes utilizan funciones Hash distribuidas y asignan valores a cada contenido compartido y a cada peer

[4]Las Tablas de Hash Distribuidas (Distributed Hash Tables, DHT) son una clase de estructura distribuida empleada en sistemas P2P descentralizados que reparten la propiedad de un conjunto de claves (keys) entre los nodos que participan en una red, donde estas son capaces de encaminar de manera eficiente mensajes al dueño de una clave determinada.

de la red.

Cada nodo sigue un protocolo de actuación que relaciona contenidos con peers responsables de estos contenidos. De esta forma, siempre que un peer desee buscar ciertos contenidos, utiliza este protocolo para determinar el/los responsable(s) de los datos y después dirige la búsqueda hacia el/los peer(s) responsable(s). Algunos ejemplos de protocolos utilizados en redes peer-to-peer estructuradas son: *Chord*, *Pastry*, *Tapestry*, *Content Adressable Network (CAN)*, y *Kadmelia* [SMK+01, RD01, ZKJ01, RFH+01, MM02].

La Fig. 1.2 muestra un esquema de una red peer-to-peer de tipo estructurada. Cuando un peer se inserta en la red P2P de tipo estructurada, todos los objetos que comparte, tienen que estar hasheados con una clave identificadora ($K1$) y posteriormente ser enviada conjuntamente con la dirección de red del peer propietario ($K1$,$I1$) (resaltado en color azul) al peer sucesor $K1$, donde finalmente éste guarda la información en su tabla de datos. Por otro lado, cuando un peer solicita una consulta de la clave $K1$ (en rojo), éste envía una petición al peer sucesor $K1$, y cuando llega la petición le devuelve la dirección $I1$ del peer propietario de los datos compartidos $K1$. A partir de entonces, el peer solicitante ya puede descargar los datos de la dirección $I1$.

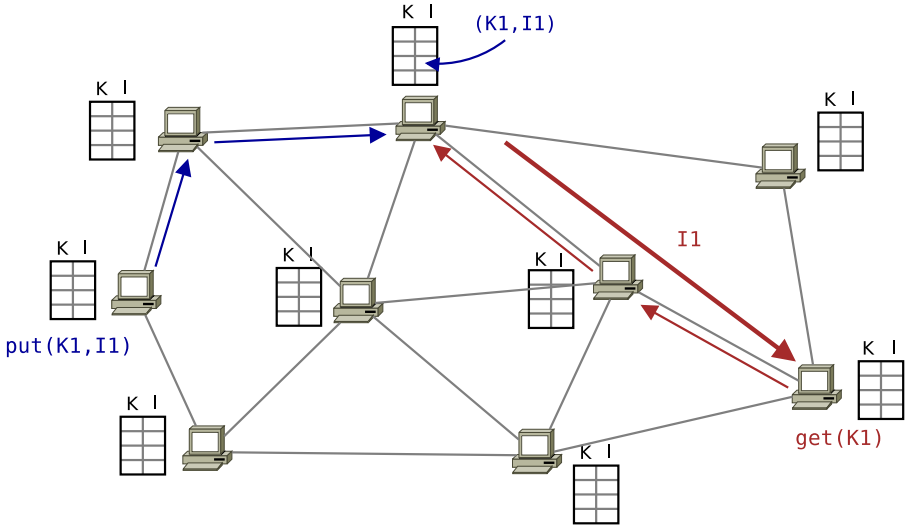

Figura 1.2: Esquema de una red peer-to-peer de tipo estructurada.

1.2.4. Clasificación de las redes P2P estructuradas

Existen diferentes redes Peer-to-Peer estructuradas según su grado de centralización. Éstas obedecen a una topología según el modelo cliente-servidor, P2P descentralizado o híbrido-mixto. A continuación se explica en detalle esta clasificación.

1.2.4.1. Modelo P2P centralizado

La primera generación de redes P2P empleaba una estructura cliente-servidor. En este caso, el servidor central mantiene una base de datos con información de los contenidos compartidos por cada peer y sus respectivas direcciones. Cada vez que un cliente se conecta o desconecta de la red, se actualiza la base de datos del servidor. En este modelo, todos los mensajes de búsqueda y control son enviados al servidor centralizado. El servidor centralizado compara la solicitud de sus clientes con el contenido de su base de datos y envía la información de la dirección al cliente en cuestión. Una vez que éste es informado, el cliente contacta con el peer directamente y accede al recurso solicitado. Los contenidos nunca son almacenados en el servidor central.
La arquitectura P2P centralizada proporciona un rendimiento muy elevado cuando se trata de localizar recursos. Todos los peers de la red deben registrarse, lo cual asegura que todas las búsquedas van a ser ejecutadas rápida y eficientemente, siempre y cuando el servidor esté bien dimensionado. Debido a ser un modelo centralizado, todo el coste y responsabilidad recae sobre una sola máquina y por lo tanto presenta problemas de escalabilidad y seguridad. Del mismo modo, es un sistema costoso de mantener porque requiere la compra y mantenimiento de máquinas dedicadas para realizar el servicio. Además, también presentan problemas relacionados con la privacidad de datos de los usuarios. Algunos ejemplos de redes peer-to-peer centralizadas son: *Napster* y *Audiogalaxy* [Nap, Aud]. En la Fig. 1.3 se puede ver un esquema de una red peer-to-peer centralizada.

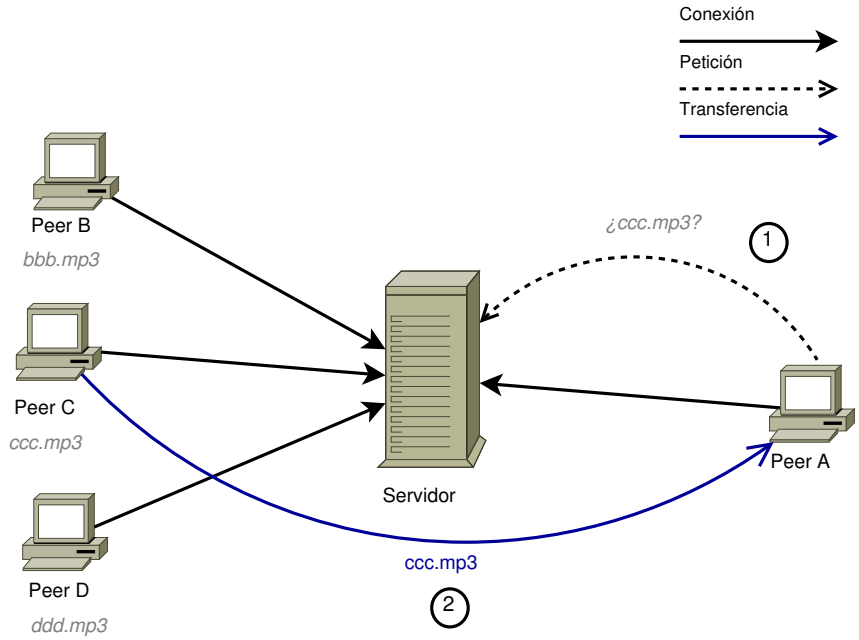

Figura 1.3: Esquema de una red peer-to-peer centralizada.

1.2.4.2. Modelo P2P descentralizado

Ésta es la segunda generación de entornos P2P y son muy comunes. Cada peer tiene el mismo rol y actúa como servidor y cliente a la vez. Cada peer, dentro de esta arquitectura, trata de mantener un cierto número de conexiones con otros peers durante todo el tiempo. Este conjunto de peers, conectados entre sí, soporta todo el tráfico de red, que está formado esencialmente por peticiones y respuestas, y varios mensajes de control utilizados, por ejemplo, para descubrir nodos. El modelo P2P descentralizado no reside en un servidor centralizado y, por consiguiente, es mucho más robusto y económico que el modelo P2P centralizado. La principal desventaja es el tiempo y sobrecarga de ancho de banda que suponen las búsquedas de información en la red. Una solicitud puede requerir viajar a través de cientos de usuarios antes de que se consiga el resultado. Además, puede ser que el recurso buscado exista y sin embargo, no se encuentre.

Algunos ejemplos de redes peer-to-peer descentralizadas "puras" son: las primeras versiones de *Ares Galaxy*, *Gnutella* y *Freenet* [Are, KC04, CSWH01].

En la Fig. 1.4 se puede ver un esquema de una red peer-to-peer descentralizada.

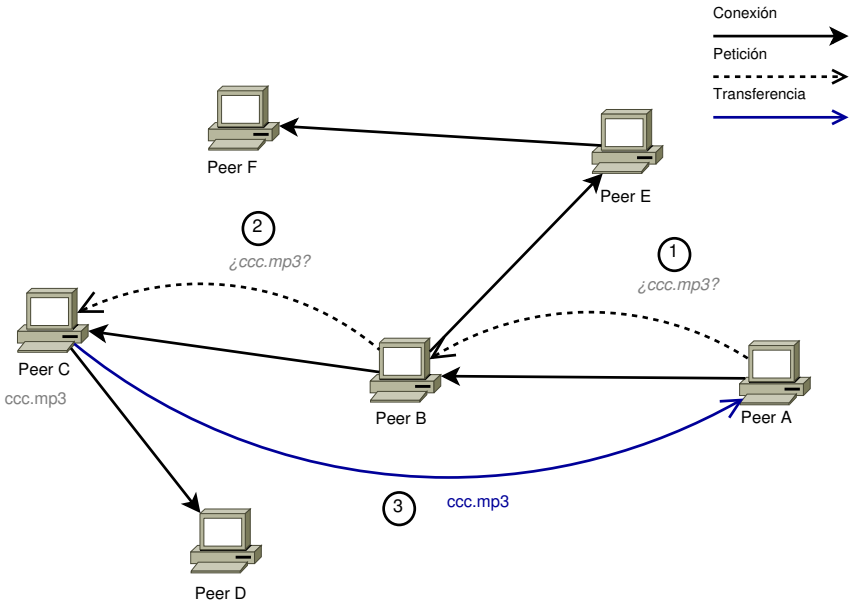

Figura 1.4: Esquema de una red peer-to-peer de tipo descentralizada.

1.2.4.3. Modelo P2P mixto

Es la tercera generación de redes P2P. Dentro de este modelo, ciertos peers de la red son seleccionados como superpeers y ayudan a gestionar el tráfico generado por otros peers. Los superpeers se seleccionan según sus capacidades computacionales o su ancho de banda. Cada nodo gestiona un determinado número de conexiones, donde a cada una de ellas se le asocia un superpeer. Las velocidades de respuesta a las solicitudes, dentro de este entorno P2P descentralizado, es comparable al de un entorno P2P centralizado, pero a la vez es mucho más escalable y tolerante a fallos.

En este entorno P2P híbrido, cada peer, al conectarse, envía una lista de sus ficheros o recursos compartidos a su superpeer. De este modo, cada solicitud es dirigida al superpeer apropiado, y luego éste puede o no reenviar el mensaje al resto de superpeers.

Existen varias aplicaciones peer-to-peer híbridas, entre las que destacan las versiones actuales de *eDonkey2000* [HBMS04, HLP+04], *BitTorrent* [Bit], *Kazaa*

[ZF], *Freenet* [CSWH01] y *Gnutella2* [Gnu]. En la Fig. 1.5 puede verse un esquema de una red peer-to-peer híbrida o mixta.

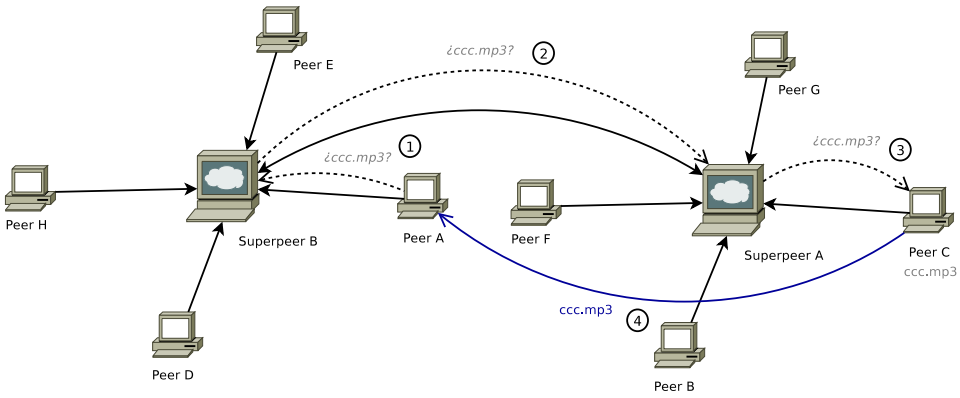

Figura 1.5: Esquema de una red peer-to-peer híbrida o mixta.

1.2.5. Aplicaciones de las redes Peer-to-Peer

Actualmente las redes Peer-to-Peer se utilizan en muchos ámbitos. Por este motivo, a continuación, se realiza una breve descripción de las aplicaciones más populares de esta tecnología:

1. **Colaboración:** Estas aplicaciones surgieron a partir de la idea de que en el mundo empresarial es necesario, muchas veces, trabajar en grupo para realizar proyectos comunes, hecho que requiere de un gran esfuerzo. Estas aplicaciones, de una manera descentralizada, permiten compartir el correo electrónico, el teléfono, el chat, la mensajería instantánea o la videoconferencia. Por ejemplo, Groove [Gro] tiene una arquitectura P2P y es equivalente al conocido Lotus Notes [Lot], con una arquitectura cliente-servidor.
   Dentro de este grupo de aplicaciones existen otras variantes descritas a continuación:

   - **Juegos en red:** En los juegos en red es necesario realizar una representación del estado del mundo, admitir conexiones de los jugadores, cambiar el estado del mundo con cada interacción de cada

jugador y difundir el nuevo estado del mundo, parcial o totalmente a cada jugador. Este tipo de aplicaciones son un área muy atractiva para entornos P2P porque son escalables a un gran número de jugadores y permite un mejor control de las comunidades de forma distribuida, en contraposición a los servidores centrales que tienen que soportar todo el peso del juego. El potencial y la aplicabilidad de la tecnología P2P ha sido demostrada en varios juegos como JXTA Chess [BGKS02], que es un juego de ajedrez desarrollado con tecnología JXTA.

- **Mensajería instantánea:** Es un sistema de comunicación intermedio entre el chat y el correo electrónico. A diferencia del chat tradicional, es más personal porque permite acceder a aquellas personas que el propio usuario autoriza y por otro lado es más inmediato que un correo electrónico, lo que supone mantener una conversación en tiempo real entre ambas partes. Actualmente, estas aplicaciones también añaden otras funciones como la transferencia de ficheros multimedia, telefonía IP, etc.
- **Telefonía IP:** Su principal ventaja es que es más barata que la telefonía tradicional y permite hacer llamadas internacionales a precio local, videoconferencia IP, etc. Algunas de las aplicaciones de Telefonía IP más populares, pero centralizadas, son MSN Messenger o Net2Phone [Mes, Net]. Skype [Sky] fue la primera alternativa de telefonía IP en P2P. Existió otra solución a la Telefonía IP con P2P llamada PeerioBiz [Pee], solución P2P empresarial que permitió crear redes de telefonía IP sin disponer de una centralita IP y compatible con Skype.

2. **Compartición de ficheros:** Las aplicaciones de compartición de ficheros son las más populares y, a la vez, las más problemáticas de todas las aplicaciones de las tecnologías P2P. Una característica destacable es que la información se distribuye de forma totalmente flexible, segura y dinámica. Esto hace que su control sea realmente complicado. En este campo existe el problema de la difusión "ilegal", en donde se distribuyen

copias ilegales de programas y ficheros audiovisuales con *copyright*. En el pasado existieron programas como *Kazaa* y *eDonkey* [ZF, eDo] o en la actualidad existen programas como *Emule* o *BitTorrent* [Emu, Bit] que fueron o son utilizados habitualmente para compartir y descargar ficheros con contenidos protegidos por derechos de autor.

3. **Compartición de capacidad de procesamiento:** Los ordenadores personales mejoran continuamente la velocidad de procesamiento, memoria principal, almacenamiento secundario e incluso el precio. Además, hoy en día, existe mucha facilidad de acceso a Internet con un elevado ancho de banda. Las aplicaciones y protocolos P2P pueden utilizar la potencia de cálculo y almacenamiento ocioso de Internet para crear sistemas completamente distribuidos de computación paralela. Aunque esto es complicado de llevar a la práctica existen, desde hace ya algunos años, aplicaciones operando de este modo. Algunas de ellas son *SETI@home* [SET99], cuyo objetivo es la búsqueda de vida extraterrestre mediante la detección de patrones que demuestren inteligencia en las ondas de radio procedentes del espacio. Otro ejemplo es *Distributed.net* [dis], cuya finalidad es descifrar códigos de encriptación con el fin de mejorarlos. Cabe destacar que nuestra tesis se enmarca en este apartado.

1.2.6. Tipo de consultas de búsqueda de recursos

A medida que se han ido desarrollando nuevas topologías P2P, éstas han ido evolucionando y han ido apareciendo nuevas técnicas de búsqueda optimizadas para realizar diferentes tipos de consultas. A continuación se describen cada una de ellas.

1.2.6.1. Búsquedas multi-dimensionales o multi-atributo

Normalmente, los sistemas P2P estructurados con DHT están limitados a realizar búsquedas de recursos computacionales con un solo atributo, conocidas como *búsquedas exactas*. Por ejemplo, una consulta de este tipo solo permite especificar $Q=\{CPU = 4\}$. Algunos sistemas han sido mejorados para permitir realizar búsquedas con más de un atributo, denominadas *búsquedas*

multi-atributo, tal como se muestra en la Fig. 1.6. Por ejemplo, hay redes que permiten realizar consultas con $Q_1 = \{CPU = 4, MEM = 2, RED = 3\}$ o $Q_2 = \{CPU = 1, MEM = 6, RED = 7\}$. En el caso del ejemplo mostrado, la red devuelve 5 nodos con la petición Q_1 y 3 con la petición Q_2. Algunas propuestas en la literatura actual que implementan estas búsquedas son [Jag06, Gut84, BKK96].

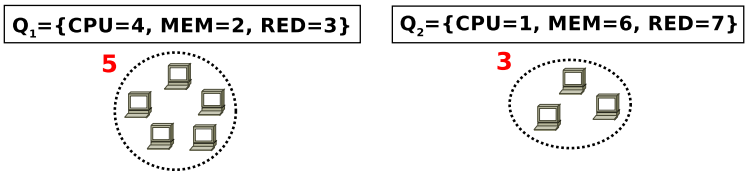

Figura 1.6: Ejemplo de búsqueda multi-atributo.

1.2.6.2. Búsquedas de resultados aproximados

Otra capacidad que añaden algunos sistemas es realizar búsquedas con resultados aproximados [GAA02, BCG05, JYF07, jJ06]. Este tipo de búsquedas sirven para localizar nodos próximos con valores similares cuando no se encuentra el recurso deseado (ver Fig. 1.7). En este ejemplo se aprecia como se buscan recursos con los valores $\{CPU = 1, MEM = 2, RED = 1\}$. Al no existir ningún nodo que los cumpla, el mensaje se redirige al nodo vecino con recursos similares $\{CPU = 2, MEM = 2, RED = 1\}$. A continuación, éste devuelve la dirección de 6 nodos con los resultados aproximados.

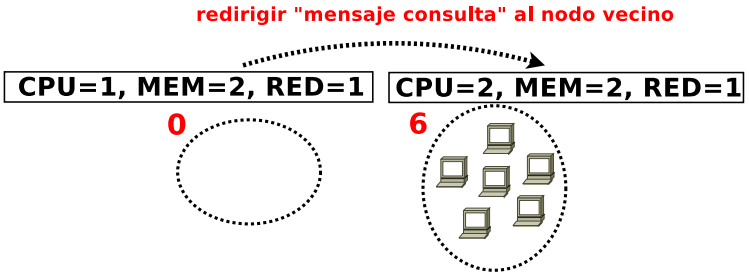

Figura 1.7: Ejemplo de búsqueda de resultados aproximados.

1.2.6.3. Búsquedas basadas en rango de atributos

Finalmente, otro tipo de búsquedas son las utilizadas para encontrar un conjunto de datos que pertenecen a un rango de atributos, como por ejemplo $Q = \{CPU >= 2 \wedge CPU <= 4\}$. En la Fig. 1.8 se muestra un ejemplo de este tipo de búsquedas. Normalmente, en este tipo de consultas se busca primero el valor del atributo más bajo ($CPU = 2$), y cuando se encuentra el nodo con dicho valor se continúa la búsqueda, vecino a vecino, hasta alcanzar el nodo con el límite más alto ($CPU = 4$). Las redes que soportan este tipo de consultas son las que tienen capacidad de ordenación de los nodos, como por ejemplo [JOV05, Jag06, DHJ+05, AX02].

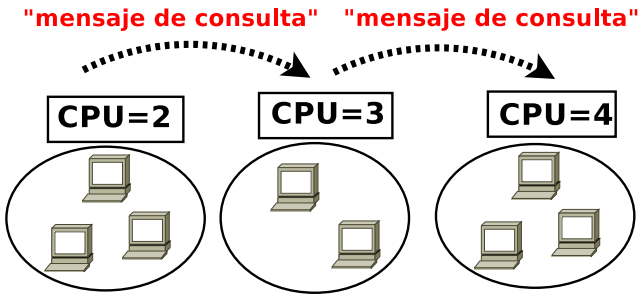

Figura 1.8: Ejemplo de búsqueda basada en rango de atributos.

1.3. Redes o topologías Peer-to-Peer existentes

En la Tabla 1.1 se muestra un esquema general de las características principales de las topologías Peer-to-Peer estructuradas más relevantes. En cada topología se muestra el tipo de arquitectura, el protocolo de búsqueda empleado, los parámetros del sistema, el rendimiento en número de saltos del algoritmo de búsqueda o *routing*[5], el grado de enlaces[6], el coste en número de saltos de la entrada/salida de peers al sistema, la fiabilidad y la resistencia a fallos.

[5] *Routing*: Se refiere al mecanismo o protocolo empleado para dirigir los mensajes de una red P2P de un nodo a otro con el fin de encontrar su destino.

[6] Grado de enlaces: Se refiere al número de enlaces que mantiene cada peer.

Taxonomía de las Topologías P2P	**CAN**	**Chord**	**Tapestry**	**Pastry**	**Kadmelia**	**Viceroy**	**Bruijn**	**B-Tree**
Arquitectura	Red de tipo Multi-Torus	Red de tipo Anillo	Red de tipo Plaxton	Red de tipo Plaxton	Métrica XOR	Red *Butterfly*(Mariposa)	Grafo completo de Bruijn	Topología de árbol
Protocolo de Búsqueda	Emparejar clave con ID del nodo	Emparejar clave con ID del nodo	Emparejar el sufijo con el ID del nodo	Emparejar la clave y el prefijo con el ID del nodo	Emparejar clave con el ID del nodo	Bajar a través de los niveles del árbol usando enlaces ring y enlaces level-ring	Algoritmo de Greedy desplazando la clave hacia izquierda/derecha y sumando un índice entre 0 y $k-1$ para alcanzar la clave destino	Emparejar prefijo con el ID del nodo o Búsqueda basada en árboles Binarios de Búsqueda [JOV05]
Parámetros del sistema	N- número de peers en la red d- número de dimensiones	N- número de peers en la red	N- número de peers en la red B- Base de la clave usada como ID de un nodo	N- número de peers en la red B - Base de la clave usada como ID de un nodo.	N- número de peers en la red B- Base de la clave usada como ID de un nodo	N- número de peers en la red	N- número de peers en la red k- Base de la clave usada como ID de un nodo	N- número de peers en la red B- Base de la clave usada como ID de un nodo
Rendimiento del Routing	$\theta(d \cdot N^{(\frac{1}{d})})$	$\theta(log_2 N)$	$\theta(log_B N)$	$\theta(log_B N)$	$\theta(\log_B N) + c$ donde $c =$ es una pequeña constante	$\theta(\log_2 N)$	$\theta(\log_k N)$	$\theta(\log_B(N \cdot (B-1)+1)-1)$
Grado de enlaces	$2 \cdot d$	$\log_2 N$	$B \cdot \log_B N$	$B \cdot log_B N + B \cdot log_B N$	$B \cdot \log_B N + B$	7	k	B
Entrada / Salida de Peers	$2 \cdot d$	$(\log_2 N)^2$	$\log_B N$	$\log_B N$	$\log_B N + c$ donde $c =$ una pequeña constante	$\log_2 N$	$k \cdot \log_k N$	$\log_B(N \cdot (B-1)+1)-1$
Fiabilidad / Resistencia a Fallos	Los fallos de peers no causa una amplia caída de la red							La fallida de peers puede causar una amplia caída de la red
	Múltiples peers son responsables de cada elemento de datos	Datos replicados a través de múltiples consecutivos peers	Datos replicados a través de múltiples peers			Carga de búsqueda repartida uniformemente entre participantes	Rutas uniformemente distribuidas y multi-direccionales	Capacidad de ordenar datos y búsqueda por rangos
	Por cada fallo la aplicación reintenta la reconexión		Mantiene guardado la traza de múltiples caminos en cada peer					Resistente si hay nodos *padre* replicados

Tabla 1.1: Esquema general de las características principales de las topologías P2P estructuradas

A continuación se explica brevemente cada una de las topologías que aparecen en la Tabla 1.1.

- **CAN [RFH+01]:** Es una topología que utiliza un espacio de coordenadas cartesianas multi-dimensional en forma de torus. Cada peer tiene asignado una única porción del espacio de coordenadas. Además, cada peer gestiona una tabla con información sobre sus vecinos: IP y coordenadas virtuales. El coste de entrada/salida de peers y el grado de enlaces es $2 \cdot d$, donde d es el número de dimensiones del torus. En general, el grado de enlaces es pequeño porque de lo contrario, podrían producirse muchos problemas de desconexiones. Mediante una función Hash, se obtiene el identificador del recurso que equivale a un punto en el espacio cartesiano. Para las búsquedas, emplea un mecanismo de routing que utiliza las coordenadas de los vecinos y aplica un algoritmo *Greedy*, donde en cada salto dirige el mensaje al vecino más cercano al nodo destino. La Fig. 1.9 ilustra una red CAN de tipo torus de 3 dimensiones, por lo tanto $d = 3$. Cada sección del torus equivaldría a un espacio de coordenadas asignado a un peer. CAN se suele utilizar en sistemas de gestión de almacenamiento como OceanStore [KBC+00], Farsite [BDET00] y Publius [WRC00].

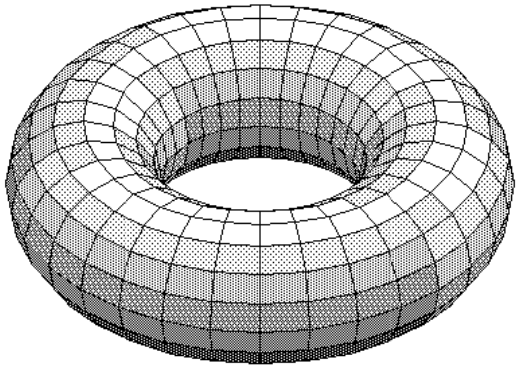

Figura 1.9: Esquema de una red P2P de tipo *CAN* con $d = 3$.

- **Chord [SMK+01]:** Es otra topología muy común que utiliza una función Hash para asignar a los peers, claves de una longitud extensa en

bits (originalmente 160 bits). En la Fig. 1.10 se observa un ejemplo de una topología de tipo Chord. Su forma es de anillo, conectado mediante enlaces sucesores y predecesores. Además, cada peer gestiona una tabla de nodos de gran tamaño, $\log_2 N$, siendo N el número de peers, que guarda la clave y la dirección IP de otros nodos. Las entradas de esta tabla guardan la relación siguiente: si i es una entrada de la tabla de un peer n, entonces almacena el enlace del $sucesor(n + 2^{i-1})$. De este modo, el protocolo de routing puede realizar saltos en potencias de 2 para llegar más rápido a su destino. No obstante, requiere un elevado coste de mantenimiento, $\log_2^2 N$, ya que por cada entrada de la tabla (con un total de $\log_2 N$ entradas), hay que actualizar y buscar el nuevo sucesor de la entrada, que comporta $\log_2 N$ saltos. Esta topología se utiliza en aplicaciones como *Cooperative mirroring*, *Cooperative File System* (CFS) [DKK+01] o servicios de DNS[7], Chord-based DNS [CMM02].

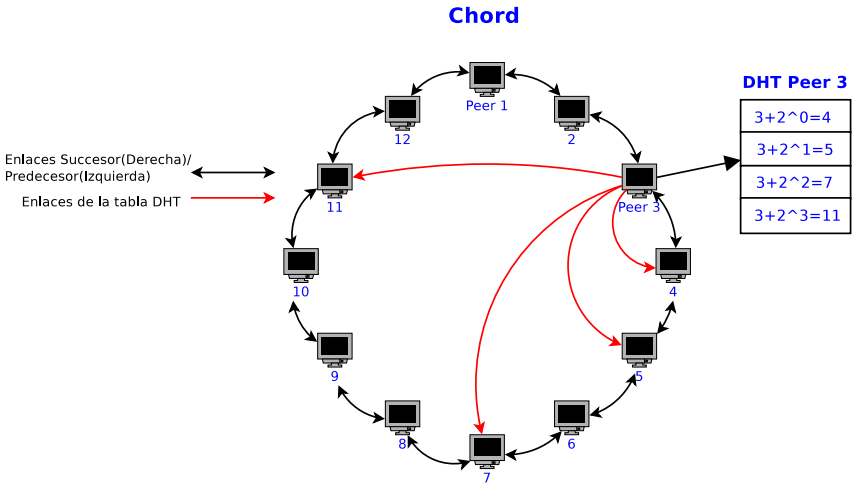

Figura 1.10: Esquema de una red P2P de tipo *Chord* [SMK+01].

- **Tapestry [ZKJ01]:** Es una red P2P de tipo malla Plaxton[PRR97] con una estructura de datos distribuida, optimizada para localizar objetos de datos en los peers. Tapestry implementa, además, múltiples réplicas

[7] DNS (Domain Name Service): es una base de datos distribuida y jerárquica que almacena información asociada a nombres de dominio en redes como Internet.

por cada objeto de datos para evitar un único punto de fallo. El mapa de routing está compuesto por $\log_B N$ niveles, siendo B el número de enlaces y donde cada uno de ellos representa el sufijo de una posición del espacio de dígitos del identificador. Cada peer gestiona en su tabla de routing, B entradas por cada nivel i, guardando las direcciones de los B peers más próximos que tienen en común los i dígitos sufijos de la clave. Su tamaño es $B \cdot \log_B N$ entradas. El protocolo de routing consiste en dirigir el mensaje, emparejando la clave destino con el sufijo actual, hasta llegar al destino. Tapestry se emplea como un componente fundamental de OceanStore system [RWE+01, KBC+00].

- **Pastry [RD01]:** Es similar a Tapestry porque también utiliza la malla Plaxton. No obstante, éste emplea un protocolo de routing emparejando la clave destino con el prefijo actual. Cada nodo gestiona una tabla de routing de vecinos y enlaces a nodos hoja. Del mismo modo, la tabla de routing gestiona $\log_B N$ filas, pero esta vez guarda $B - 1$ entradas. La relación que mantiene en este caso es que la fila n guarda la dirección de los peers que comparten los n primeros dígitos de la clave, pero el dígito $n + 1$ contiene los otros $B - 1$ posibles valores respecto al de su clave. Este sistema es empleado por aplicaciones como Scribe [RKCD01], que es una infraestructura multicasting que soporta un gran número de grupos y miembros por grupo para la difusión colectiva de mensajes. Squirrel [IRD02] utiliza Pastry para crear un sistema distribuido de caché y navegación web virtual.

- **Kadmelia [MM02]:** Al igual que Chord, cada nodo asigna un identificador clave de 160 bits. Su protocolo de búsqueda se basa en la métrica XOR para obtener la distancia entre puntos en el espacio de claves, de modo que la distancia entre dos puntos a y b es $d(a, b) = a \oplus b$. No obstante, cada peer tiene que gestionar una doble lista de k nodos, llamada *k-buckets*, donde en cada posición guarda los nodos más recientes entre las posiciones 2^i y 2^{i+1}, es decir, son claves que tienen el mismo prefijo comprendido entre $[0 - i]$ y además i representa el nivel del bucket, $(0 \leq i \leq 159)$. Normalmente el valor de k es 20. Actualmente se uti-

liza en aplicaciones P2P de descarga de ficheros como Emule, LPhant [Emu, LPh], etc.

- **Viceroy [MNR02]:** Está diseñada para gestionar el descubrimiento y localización de datos y recursos en una red aproximada del tipo *Butterfly*. Aplica funciones DHT uniformes para distribuir el conjunto de datos. La topología está organizada en $\log_2 N$ niveles, donde cada nivel es un anillo que contiene un número equitativo de peers conectados mediante enlaces $level - ring$. Además la interconexión entre niveles se realiza a través de enlaces llamados *Down-Left* y *Down-Right edge*. Esta topología destaca sobre las otras porque requiere un número constante de enlaces, 7 por cada peer. En la Fig. 1.11 se puede ver un esquema que ilustra la topología empleada por este sistema P2P.

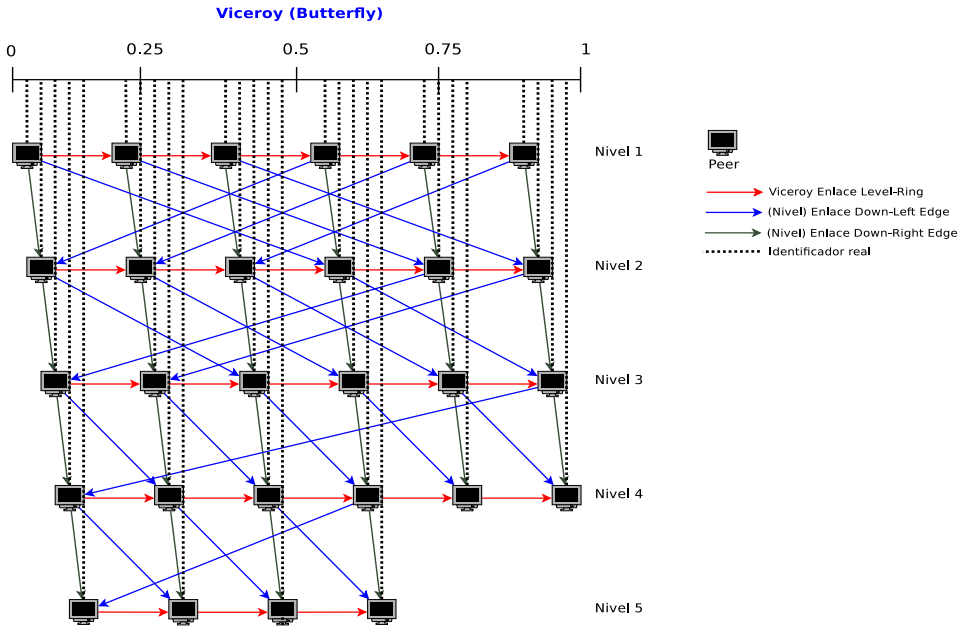

Figura 1.11: Esquema de una red P2P de tipo *Viceroy* [MNR02].

- **Bruijn [LKRG03, NNL04, EH85]:** El grafo de Bruijn es un grafo próximo a los grafos óptimos de grado constante y de diámetro logarítmico $log_k N$. Cada peer contiene k enlaces de entrada y k enlaces

de salida. Además, como toda topología P2P, cada peer tiene asignado un identificador. La construcción de los enlaces de cada peer se basa en que un peer con la clave H enlaza a todos los peers con clave $\{H << 1+0, H << 1+1, ..., H << 1+k-1\}$, donde $<<$ es la operación de desplazamiento de dígitos hacia la izquierda. Este procedimiento se muestra en la Fig. 1.12 (izquierda). Por otro lado, en la Fig. 1.12 (derecha) se muestra la topología Bruijn completa con una clave H de 3 dígitos y de base k igual a 2 (binaria).
El resultado final es un grafo conexo[8] y dirigido[9] por defecto, en donde los enlaces están distribuidos uniformemente por toda la red. Además de ser de grado constante, como Viceroy, es más flexible porque su construcción permite cambiar el grado de conexiones por nodo, según la base k de la clave usada. Las topologías de árbol en redes P2P no suelen utilizar tanto los mecanismos DHT. Tan solo se utilizan para ordenar los recursos de los peers. Una aplicación práctica de este grafo es la creación de multiprocesadores Bruijn [SP89], mediante la tecnología VLSI, en un solo chip utilizados por ejemplo para resolver ciertos problemas paralelos como N-point Fast Fourier Transform, *FFT*. Otra aplicación es Koorde [KK03], que es una red distribuida que emplea una mezcla de Chord y Bruijn y destaca porque alcanza un rendimiento de búsqueda $\log_2 N$, con un bajo número de enlaces, igual a 2. Por otro lado, también es capaz de realizar búsquedas con $\theta(\frac{\log N}{\log(\log N)})$ saltos por petición, cuando existen $\theta(\log N)$ vecinos por nodo.

- **B-Tree [CBR+08, CRB+09, CBGS11, HKO05]:** Es una topología en árbol completamente jerárquica. La Fig. 1.13 ilustra la topología, construida a partir de un *root* peer (nodo raíz), el cual se conecta a un conjunto de B peers, llamados *hijos* y que están situados en un nivel inferior jerárquicamente. Éstos a su vez son nodos *padre* de un conjunto de B hijos situados en otro nivel inferior, y así sucesivamente. Los nodos finales que no tienen hijos se les llama nodos *hoja*. Se caracterizan por tener el número de niveles igual a $\log_B(N \cdot (B-1)+1)-1$. Las topologías

[8] *Grafo Conexo*: grafo cuyos vértices están conectados entre si mediante aristas
[9] *Grafo Dirigido*: grafo donde sus aristas tienen una dirección.

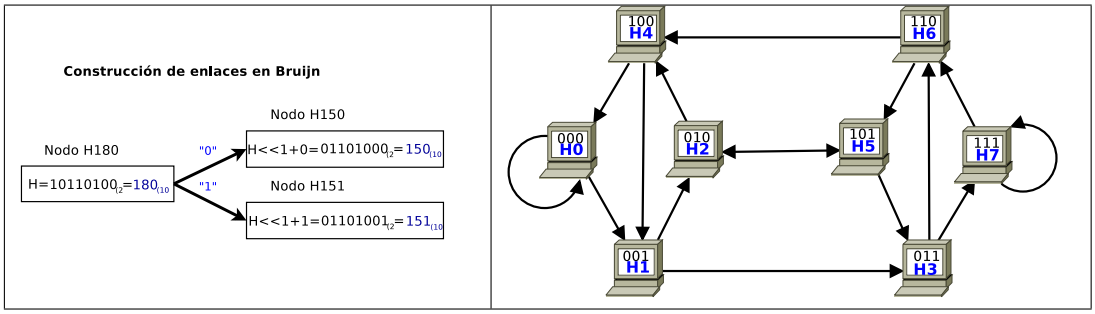

Figura 1.12: Esquema de construcción de los enlaces Bruijn(izquierda). Esquema de una red P2P de tipo Bruijn de 8 nodos y grado $k = 2$(derecha).

de árbol en redes P2P no suelen utilizar tanto los mecanismos DHT. Tan solo se utilizan para ordenar los recursos de los peers, como es el caso de la topología BATON [JOV05], que emplea Arboles Binarios de búsqueda para ordenar los objetos por todo el árbol y permitir así hacer búsquedas basadas en rango.

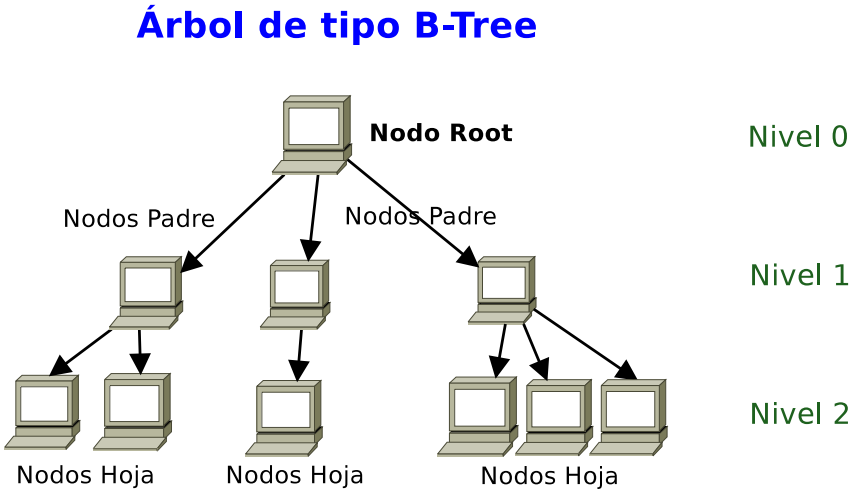

Figura 1.13: Esquema de una red P2P de tipo *B-Tree*.

1.4. Objetivos de la tesis

El objetivo principal de esta tesis es diseñar y modelar los aspectos fundamentales que conlleva la implementación de una plataforma de cómputo distribuida con calidad de servicio basada en el paradigma P2P, como una alternativa emergente de bajo coste para proporcionar acceso a recursos de cómputo compartido. Por Calidad de servicio (QoS[10]) se entiende que la plataforma será capaz de dar siempre un servicio óptimo, tolerante a fallos y con un tiempo de respuesta rápido en la búsqueda de recursos de cómputo.

Por lo tanto, el reto principal recae en diseñar y modelizar una arquitectura peer-to-peer de cómputo distribuido para obtener un entorno de cómputo con unas prestaciones equivalentes a un supercomputador de bajo coste, aprovechando los recursos disponibles de Internet.

A continuación se detallan los objetivos específicos del diseño y de la modelización de la arquitectura propuesta:

1. El sistema de cómputo distribuido ha de disponer de las siguientes *funcionalidades básicas*: inserción, mantenimiento y salida de peers para controlar el constante dinamismo de entrada y salida de peers en el sistema, sin afectar la lógica y rendimiento general del sistema.

2. Otra apuesta de nuestro sistema es que tenga *Calidad de Servicio*. Por este motivo, el sistema tiene que implementar técnicas que eviten irregularidades en el funcionamiento de sus servicios y ofrezcan un tiempo de respuesta bueno en los servicios ofrecidos.

3. Asimismo, el sistema debe controlar y organizar la heterogeneidad de los recursos no dedicados de los nodos que forman el sistema P2P para mejorar la búsqueda de recursos.

4. También queremos que el sistema sea capaz de localizar recursos de cómputo en el menor tiempo posible: para esto es necesario implementar un sistema P2P estructurado y un *algoritmo de búsqueda eficiente* y adaptado a la topología diseñada.

[10] QoS: Quality of Service

5. Además, la plataforma ha de ser flexible en cuanto a las peticiones de localización de recursos computacionales. Al ser una plataforma de cómputo distribuido, es necesario que sea capaz de realizar *búsquedas de múltiples atributos/recursos* a través de la red con un tiempo mínimo, así como *búsquedas basadas en rango*.

6. El sistema tiene que ofrecer no sólo búsquedas exactas, sino también *búsquedas aproximadas*, capaces de buscar aquellos recursos que más se asemejan al recurso solicitado y que no se encuentra en la red.

7. Para que el sistema P2P pueda tener alcance mundial, el diseño de su topología *ha de ser escalable*, admitiendo multitud de nodos sin disminuir excesivamente el rendimiento global del sistema.

8. Para lograr robustez en el sistema global de cómputo, ha de ser también *tolerante a fallos*. Esto significa que si cualquier peer del sistema se desconecta de forma esperada o inesperada, el sistema tiene que repararse o reestructurarse de forma eficiente asegurando en todo momento el correcto funcionamiento del sistema. Al mismo tiempo, el *protocolo de routing* tiene que tener flexibilidad para poder seleccionar rutas alternativas en caso de fallo de nodos intermedios de las rutas a los peers seleccionados.

Una vez desarrollado el sistema, se tiene que comprobar el correcto funcionamiento del modelo en un entorno de simulación, comparando su rendimiento con otras propuestas de la literatura, expuestos en cada capítulo de la memoria.

1.5. Estructura de la memoria

En este apartado se describe brevemente el contenido de los capítulos que integran la memoria:

- **Capítulo 1: Introducción**. En este capítulo se realiza una descripción y clasificación de los conceptos principales asociados al ámbito de la Tesis, así como la tecnología utilizada. También se definen los objetivos de la tesis y la estructura general del presente documento.

- **Capítulo 2: Estado del arte**. En este capítulo se describen las principales aportaciones en la literatura y el estado actual de las plataformas P2P, cuales son sus metas y sus puntos fuertes y sus carencias con respecto a nuestra propuesta. Ello permitirá entender como funcionan algunas de ellas y enmarcar mejor las aportaciones realizadas en esta tesis doctoral.

- **Capítulo 3: DisCoP: una nueva arquitectura de cómputo P2P**. En este capítulo se explican detalladamente las decisiones tomadas sobre el diseño del modelo P2P propuesto y el porque de dichas decisiones. Se explica por tanto el diseño y los componentes principales de la arquitectura de cómputo distribuido desarrollada en esta tesis, llamada DisCoP (*Distributed Computing P2P Platform*). Asimismo se define un nuevo procedimiento para calcular la *Localidad*, métrica usada para medir el grado de proximidad entre dos nodos con características similares. Finalmente, se realiza un análisis experimental de la fortaleza de la plataforma, su tolerancia a fallos y su localidad.

- **Capítulo 4: Funcionalidades del Sistema**. En este capítulo se explican detalladamente todas las funcionalidades y políticas de búsquedas de recursos diseñadas y optimizadas para el sistema *DisCoP*. Para cada una de ellas, se describirán sus modelos algorítmicos y el análisis experimental realizado.

- **Capítulo 5: Conclusiones y trabajo futuro**. Se exponen las conclusiones extraídas de la realización del trabajo y se reflexiona sobre las posibles lineas de trabajo futuro. Asimismo se enumeran las principales publicaciones, fruto de las aportaciones realizadas en esta tesis.

Capítulo 2

Estado del arte

En el capítulo anterior se ha comentado que recientemente la computación Peer-to-Peer ha emergido como uno de los nuevos paradigmas de Computación Distribuida [FI03]. Este hecho ha provocado que la comunidad científica haya volcado su interés en el estudio de nuevas técnicas que optimicen las prestaciones de la computación P2P.

Este capítulo describe las principales aportaciones en congresos y revistas de la comunidad científica en el campo de la computación Peer-to-Peer. Acorde con este objetivo, se realiza una descripción de la evolución de los sistemas P2P, clasificándolos como sistemas basados en DHT puros, DHT modificados y sin DHT, con el fin de determinar el tipo de mecanismo de búsqueda óptimo en el diseño de nuestro sistema. Asimismo, en este capítulo también se hace una descripción de los sistemas de computación distribuida P2P más importantes, analizando su relevancia respecto a la propuesta realizada en esta tesis así como sus ventajas, limitaciones y posibles mejoras. Esto nos ayudará a comprender mejor la situación actual de los sistemas de cómputo P2P y las aportaciones de nuestra propuesta.

2.1. Sistemas de búsqueda en entornos P2P

En esta sección se clasifican y describen los principales sistemas de búsqueda empleados en diferentes entornos P2P. En la sección 2.1.1 se introduce los sistemas P2P con *DHT*, en la sección 2.1.2, los sistemas *DHT pseudo-*

modificados y finalmente en la sección 2.1.3 los *DHT completamente modificados* (o sin DHT).

2.1.1. Sistemas P2P con DHT

Las tablas Hash distribuidas, conocidas habitualmente como *DHT (Distributed Hash Tables)* son un tipo de algoritmos o sistemas distribuidos descentralizados, desarrollados para proporcionar una infraestructura de búsqueda y almacenamiento eficiente, robusta y escalable, con una interfaz de programación clara y sencilla. Las tablas Hash son unas estructuras de datos incorporadas en cualquier librería estandard de muchos lenguajes de programación (C++, Java, Perl, etc.), pudiendo ser empleadas para desarrollar de manera rápida y eficiente servicios de consulta más complejos. La utilización de DHT's permite realizar algoritmos de búsqueda descentralizados muy eficientes que permiten descubrir rápidamente nodos responsables de un determinado conjunto de datos en un tiempo mínimo.
En un sistema P2P con DHT, la tabla Hash se distribuye uniformemente entre todos los nodos, almacenando cada uno una parte de ella. Cada nodo obtiene un único identificador, creado normalmente a partir de una función Hash (SHA[1], MD4 o MD5[2] [MD5, SHA]), que tiene como parámetro de entrada la dirección IP del nodo y calcula un identificador aleatorio que determina la posición en la topología. Los datos del sistema, como puede ser ficheros, reciben del mismo modo también una única clave. Estas claves numéricas, junto con la dirección del nodo que lo contiene, son enviadas y repartidas (*put*) a un nodo llamado *sucesor*, nodo que tiene como identificador el valor igual o inmediatamente más grande al identificador del objeto a compartir. De este modo, el sistema ofrece tres funcionalidades básicas sobre los datos: búsqueda (*lookup*), extracción (*get*) y almacenamiento (*put*). El objetivo de una DHT es localizar rápidamente los datos utilizando las claves generadas por la función Hash. Una vez localizado el dato, su valor ya depende del tipo de aplicación

[1]Secure Hash Algorithm: es una familia de funciones criptográficas Hash diseñadas por la NSA (National Security Agency[NSA]) y publicadas como estandard por el Gobierno de EEUU.

[2]Message Digest: similar a las función *SHA-1*, son desarrolladas por el profesor Ronald L. Rivest del MIT (Massachussets Institute of Technology [MIT]).

P2P que está utilizando el sistema DHT. Las DHT requieren que las redes o topologías estructuradas estén construidas en función de la distribución generada por ellas mismas. De este modo, las peticiones de búsqueda requieren un menor número de saltos para encontrar el resultado. Como consecuencia de ello, muchas topologías ([SMK+01, RD01, ZKJ01, LKRG03, MNR02]) alcanzan una distancia máxima logarítmica entre dos nodos cualquiera.
Estos sistemas pueden ser utilizados para desarrollar sistemas de almacenamiento persistente, sistemas de ficheros distribuidos cooperativos, sistemas de nombres distribuidos, sistemas de caches Web cooperativas, sistemas de multicasting, bases de datos, etc.

Las primeras aplicaciones P2P estaban diseñadas para compartir ficheros a través de Internet usando redes no estructuradas, como [KC04, CSWH01]. Estos sistemas consumían un gran ancho de banda de red en la búsqueda de información, debido al excesivo flooding de mensajes realizado para localizar ficheros compartidos. Estos servicios pasaron a ser obsoletos a favor de diseños más eficientes. De aquí el surgimiento de los primeros sistemas DHT como son Chord, CAN y Pastry [SMK+01, RFH+01, RD01].

Sin embargo, las redes DHT tienen una serie de limitaciones. En primer lugar, las búsquedas eficientes de estas redes P2P están optimizadas generalmente para gestionar recursos inmutables como son ficheros, registros de bases de datos, documentos, etc, que no cambian durante el tiempo. Por lo tanto, las DHT son una mala opción cuando los recursos cambian de estado durante el transcurso del tiempo como por ejemplo la cantidad de potencia de CPU disponible o la memoria libre. Por este motivo, en esta tesis hemos orientado nuestros esfuerzos hacia el diseño de una arquitectura que no utilice DHT's, pero que sea capaz de soportar el gran dinamismo de los peers.
Otras propuestas como [STS02] proponen un sistema de ficheros distribuido por toda la red de peers. Implementan métodos capaces de poder modificar libremente el contenido de sus ficheros.

El proceso de búsqueda DHT es eficiente cuando se realiza una consulta exacta de algún recurso en la tabla Hash; pero no ocurre así cuando se intenta obtener recursos con referencias parciales (substring), ya que en este caso, resultan bastante ineficientes. Por lo tanto, una limitación importante se da

en los casos en que no tenemos una idea exacta de qué contenidos o recursos debemos buscar. No obstante, existen ciertas soluciones que consisten en realizar peticiones de búsqueda de datos partiendo de un nombre incompleto como clave de búsqueda. Este tipo de búsquedas se explican en la sección 2.1.2, y se caracterizan porque son búsquedas basadas en sufijos o prefijos.
Por otro lado, existe otro tipo de búsquedas basadas en rango, consistentes en buscar todos aquellos datos o recursos que se encuentren dentro de un determinado rango de búsqueda. Por ejemplo, $CPU = [2 - 5]$ o $CPU \geq 3$. Las propuestas [JOV05, DHJ+05, SGAA04] se enmarcan en este tipo de búsquedas. A primera vista, mediante el uso de DHT's parece imposible poder realizarlas, pero existen varias propuestas como [GS04, CJ07, ZSLS06] que modifican el diseño original de una DHT, permitiendo soportar de manera eficiente este tipo de consultas basadas en rango.

2.1.2. Sistemas DHT pseudo-modificados

Los sistemas DHT pseudo-modificados son similares a los DHT originales pero realizan modificaciones dentro de su estructura (preferentemente en una capa superior) para mantener la *localidad* de datos en búsquedas de rangos y aproximadas. El paradigma se llama LSH, y se basa en que las funciones Hash uniformes (DHT) son reemplazadas por funciones Hash sensitivas a la localidad (LSH). Esto significa que algunos overlays basados en DHT que indexan directamente datos, como Chord, soportan eficientemente consultas basadas en rango [SC08, AX02, DHJ+05, SP03, LLW+06] usando este tipo de funciones.
Una función HASH es sensitiva a la localidad cuando la probabilidad de colisión de dos puntos muy similares es muy alta y la probabilidad de colisión es muy baja cuando éstos están lejos. Un modelo común empleado para una función LSH es el siguiente: dado un valor de entrada $p \epsilon S$, y una familia de funciones Hash, $h_1, h_2, ..., h_k$ uniformemente aleatorias, LSH convierte p en un identificador también llamado bucket, con la etiqueta $g(p) = (h_1(p), h_2(p), ..., h_k(p))$, empleando un conjunto aleatorio de funciones Hash. El número de buckets empleados como sitios para guardar los datos transformados por la LSH es mucho menor que el dominio de entrada S. De este modo, el algoritmo de

búsqueda solo tiene que recoger todos los datos mapeados en el bucket con el identificador $g(p)$.

Otras investigaciones, como [GAA02], proponen que las funciones *LSH* tengan la capacidad de realizar consultas aproximadas cuando se realiza una búsqueda basada en rango. Cuando se realiza una consulta en rango, se utilizan L grupos (llamados g_i) de funciones *LSH* para poder obtener un conjunto de resultados de todas las búsquedas aproximadas.
Para realizar consultas aproximadas de una manera más eficiente y optimizada, [BCG05] construye árboles de prefijos, llamados *LSH-Trees*. Para implementar las búsquedas aproximadas, emplea L LSH-Trees. A todo el conjunto se le llama *LSH-Forest* y cada *LSH-Tree* se genera a partir de una función *LSH* diferente. A través de un nuevo diseño de los algoritmos de inserción, eliminación y consulta, la exploración de las estructuras *LSH-Forest* hace posible encontrar resultados ordenados en rango con un coste eficiente. Los autores de este modelo de búsqueda lo emplean como esquema de indexación para entornos P2P, memoria principal, consultas en disco, etc.

Otro trabajo [JYF07], propone un nuevo esquema de indexación y búsqueda basado en palabras clave, llamadas *keytoken*. En él se propone un nuevo esquema de indexación sensible a la localidad y a la aproximación mediante el mapeo de palabras clave extraídas de un objeto O en un Hipercubo multidimensional. A partir de un objeto O se le extraen todas las palabras clave que lo describen. Para cada palabra clave, se comprueba sus caracteres y por cada carácter que aparece se activa su bit asociado a 1 en la clave final de r bits. Esto implica que la longitud de la clave final de r bits será igual al número de caracteres del alfabeto $|\mathcal{A}|$ utilizado para describir los objetos O.
La idea de hacer búsquedas aproximadas es similar al juego de encontrar palabras a partir de un conjunto de caracteres y hacer palabras a partir de él y obtener nuevas palabras modificando solo un bit de la clave original asociada. Esta clave sirve como vector en un hipercubo r-dimensional, donde los nodos vecinos del hipercubo son los que tienen la misma clave menos un bit de diferencia. De este modo, capacita al esquema de búsqueda para la ordenación de palabras clave por rango o por proximidad de una manera eficiente. Este esquema virtual puede ser trasladado fácilmente a cualquier red DHT, donde

se logra unos resultados muy buenos. Además, los autores dicen que aplicando este modelo se evita desbalanceo de carga, puntos calientes (hot spots), escasa tolerancia a fallos, redundancia en el almacenamiento de datos e incapacidad de facilitar rangos y expansión de palabras.

KISS [jJ06] es una propuesta de los mismos autores, pero esta vez mejora el esquema de índice y búsqueda de palabras clave optimizado para hacer búsquedas de prefijos en redes estructuradas P2P con DHT. La idea es la misma que la del trabajo anterior, pero ahora todas las palabras clave asociadas a un objeto O se codifican usando cada carácter de la palabra más la posición i donde está ubicado para formar un identificador de r bits, empleado para realizar la búsqueda en el hipercubo. La clave general del sistema de r bits es mayor que la de la propuesta anterior. Sin embargo, este esquema permite hacer búsquedas aproximadas y en rango mucho más precisas ya que la clave contiene información de la posición de los caracteres.

En la Fig. 2.1 se puede ver el proceso de transformación de todas las palabras claves extraídas de un objeto O concreto a un identificador de un nodo de un hipercubo r-dimensional. Este proceso facilita mucho la búsqueda de subpalabras y consultas aproximadas, ya que a través de una exploración en expansión de un árbol binomial de un nodo del hipercubo, se obtienen todos los posibles resultados en rango y aproximados.

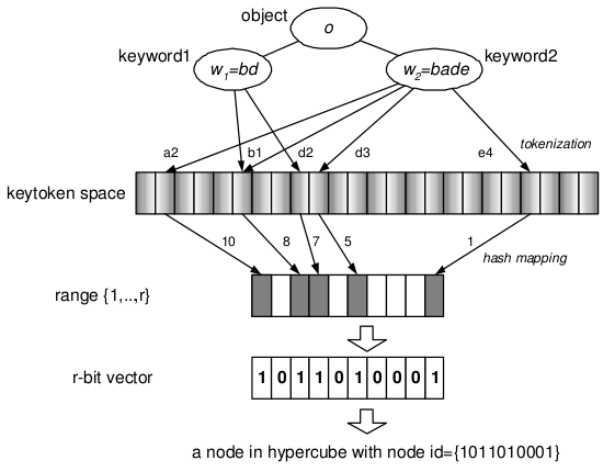

Figura 2.1: Esquema de indexación y búsqueda de palabras basado en "*keytokens*" (*KISS* [jJ06]).

Finalmente, un último trabajo [BVV03], propone el diseño de una estructura de árbol virtual, llamado *Cone*, también orientada a prefijos que está enlazada con la capa DHT de nodos físicos. El árbol se sitúa en la parte superior de la capa física, donde cada nodo físico corresponde a una hoja de la estructura de árbol. Por cada par de nodos vecinos e hijos, sube de nivel el nodo que contiene la clave más grande y de forma recursiva se aplica en cada nivel hasta construir todo el árbol. De este modo, se pueden obtener búsquedas basadas en rango con un coste mínimo. El coste de una búsqueda en esta propuesta es del orden $\log_2(N)$, donde N es el número actual de peers en el sistema. En la Fig. 2.4 se ilustra el esquema general de la implementación de búsqueda del sistema *Cone* sobre una red DHT.

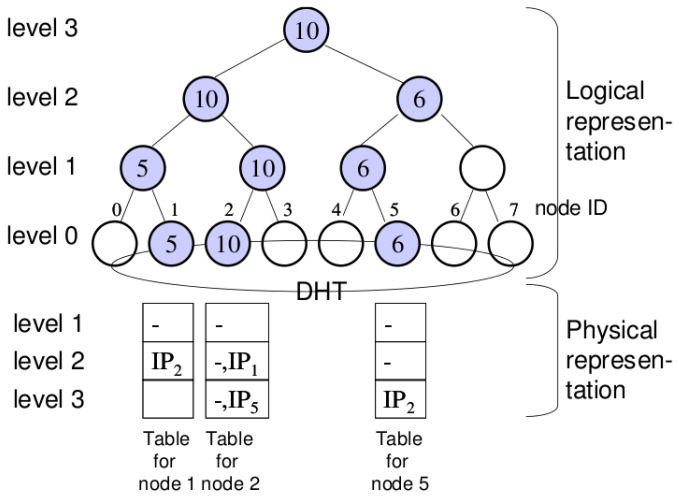

Figura 2.2: Funcionamiento de Cone. Esquema de índice y búsqueda basado en árboles sobre una red DHT [BVV03].

2.1.3. Sistemas DHT completamente modificados

Los DHT completamente modificados tienen un esquema de indexación completamente libre y no tienen nada en común con los DHT's completos. Son por lo tanto nuevas propuestas de overlays. A continuación se describen los más importantes [AS07, CLGS04, CLM+07, JOV05, Jag06].

El *grafo Skip* [AS07] es una estructura de datos distribuida y jerárquica

similar a un árbol y está compuesta por una serie de listas en cada nivel llamadas *listas Skip* de claves ordenadas. En la Fig. 2.3 se ilustra el esquema de una *lista Skip*. El funcionamiento es simple, todos los nodos del sistema P2P están enlazados y ordenados en la lista del nivel 0. Para la creación de las listas de los niveles superiores se selecciona un nodo con una determinada probabilidad p, y así para todos los niveles. De este modo, este esquema permite realizar búsquedas ordenadas. No obstante, los autores no emplean estas listas sino que las modifican dando como resultado el *grafo Skip*.
El *grafo Skip* difiere de una *lista Skip* en que la topología puede tener varias listas en el mismo nivel y cada nodo participa en una de ellas. Es equivalente a una colección de más de n listas Skip donde los nodos están situados en más de una *lista Skip* en los niveles inferiores. De este modo, se convierte en una estructura optimizada para realizar búsquedas basadas en prefijos y hacer consultas de palabras en rango y aproximadas.

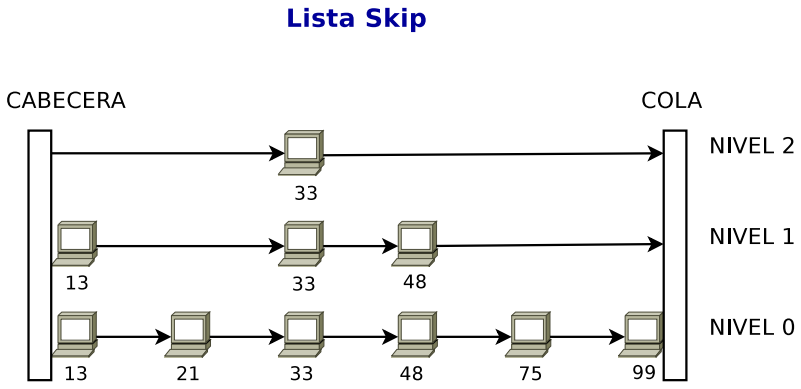

Figura 2.3: Descripción del modelo de *grafo Skip* [AS07] basado en una serie de listas ordenadas llamadas *listas Skip*.

Los grafos *Skip* son altamente resistentes y tolerantes a una gran fracción de nodos fallidos sin perder conectividad y mejoran el balanceo de carga respecto a muchos otros sistemas.

Otros sistemas, como *PTree* [CLGS04] y *PRing* [CLM+07], son redes P2P que implementan estructuras de árboles llamadas *B-Trees* de forma distribuida, tolerantes a fallos y escalables. En *PTree*, cada peer gestiona un árbol $B^+ - Tree$ semi-independiente respecto de los otros nodos. Las estructuras $B^+ - Tree$ son muy comunes en búsquedas de bases de datos. Su objetivo es

crear un árbol índice de claves ordenadas de más pequeño (izquierda) a más grande (derecha), donde cada clave tiene su contenido asociado. De este modo se puede encontrar eficientemente el objeto deseado con $log_2 N$ saltos y también hacer búsquedas basadas en rango. Estos árboles resultan eficientes en la inserción, eliminación y búsquedas exactas y en rangos.

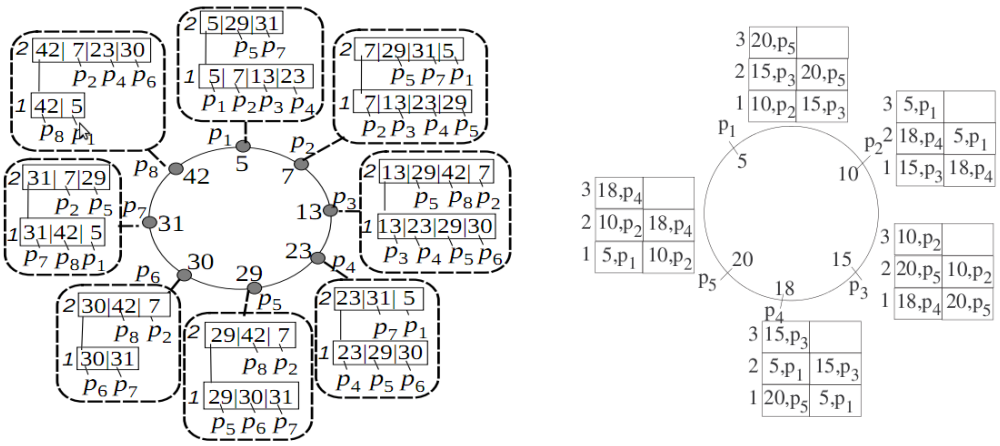

Figura 2.4: Ilustración del modelo de *P-Tree*(izquierda) [CLGS04] y *P-Ring*(derecha) [CLM+07] basado en las estructuras de árbol $B^+ - Tree$ distribuidas.

De forma resumida, un árbol $B^+ - Tree$ consta de una tabla de varios niveles donde cada nivel contiene d sucesores consecutivos. El nivel 0 guarda los índices de los d primeros sucesores. En el nivel 1 guarda los d sucesores de cada d saltos. En el nivel 2 d sucesores de cada d^2 saltos, y así sucesivamente hasta llegar al último nivel.

PRing implementa un sistema similar a *PTree*, también llamado HR^3 en relación a las estructuras $B^+ - Tree$, pero se diferencia en el mecanismo de creación de los árboles $B^+ - Tree$ de cada nodo. Mientras *PTree* construye la tabla a cada nivel empezando a partir del índice del propio peer, *PRing* la construye a partir del índice del último nodo insertado en el nivel inferior. Ésto consigue mejorar el balanceo de carga y además permite obtener resultados de búsquedas exactas y en rangos con un coste $\log_d N$, donde d es fácilmente configurable.

[3]HR (Hierarchical Ring): Anillo jerárquico

Del mismo modo, Baton [JOV05] es un *overlay* con una estructura de árbol binario de búsqueda balanceado que soporta consultas exactas y en rangos, sin el uso de DHT's, empleando también tablas de routing de saltos en potencia de 2 desde cada nivel. El rendimiento de búsqueda es igual que el de la topología Chord, $\log_2(N)$, pero tiene un coste de mantenimiento mucho menor, $\log_2 N$, respecto $\log_2^2 N$ de Chord. Además, añadiendo un número pequeño de enlaces, junto con los ejes del árbol, esta topología es capaz de obtener una tolerancia a fallos excelente y una congestión balanceada.

VBI-Tree [Jag06] es un framework general que tiene como objetivo mapear un índice multidimensional a partir del existente dentro del árbol binario de Baton. Este sistema permite mapear datos multidimensionales y soportar consultas basadas en rango. Por cada nodo asigna una porción del espacio dimensional asociado a un atributo. También por cada nodo crea nuevos enlaces, además de los propios de *Baton*. De este modo permite soportar varios sistemas populares multi-dimensionales como R-Tree, X-Tree, [Gut84, BKK96], etc.

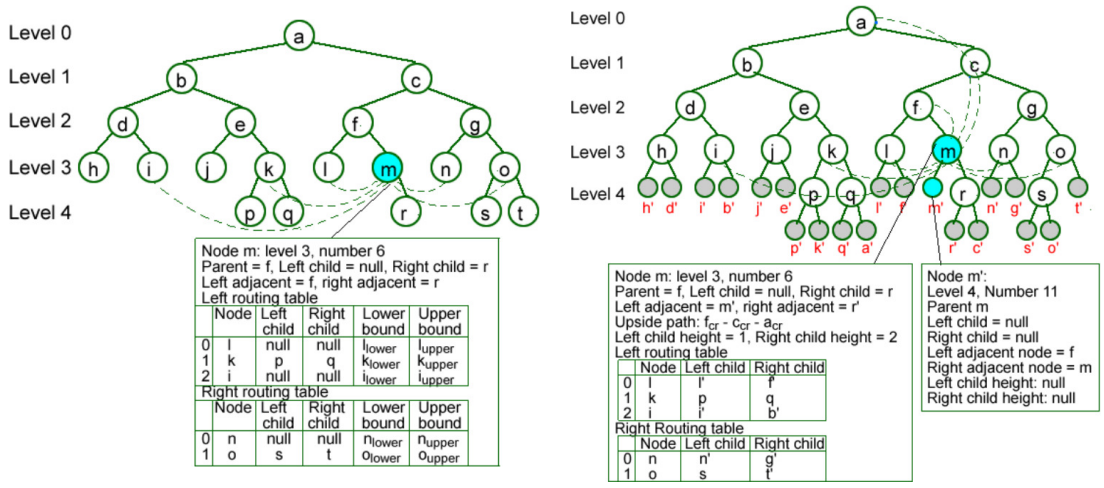

Figura 2.5: Ilustración del modelo de *Baton*(izquierda) [JOV05] y *VBI-Tree*(derecha) [Jag06] donde este último es un nuevo índice basado en la topología de *Baton*.

Otro trabajo relevante es Mercury [BAS04], que gestiona consultas multi-atributos creando un anillo jerárquico en forma de *hub* de routing por cada atributo/dimensión. Las consultas se envían a uno o varios *hubs* asociados a los atributos a consultar. Cada *hub* de routing se organiza dentro de un overlay

circular de nodos donde los datos están desplazados de forma contigua en el anillo. Ya que Mercury no utiliza funciones Hash, el particionamiento entre nodos puede ser no uniforme y que exista un gran desbalanceo de datos. Debido a ello requiere un esquema de balanceo de carga explícito, es decir, no trivial. Finalmente, un trabajo similar al presentado en esta tesis es Squid [SP08]. Squid utiliza un esquema multi-atributo conservando la localidad, basada en la *SFC*[4] de Hilbert. La novedad de este sistema es que reduce muchísimo la dimensionalidad del espacio multi-atributo, mapeando de manera efectiva la información multi-atributo a los peers físicos, preservando la localidad. Squid soporta búsquedas complejas que contengan palabras clave parciales, rangos y comodines. Hay que destacar que su arquitectura es muy similar a nuestra propuesta ya que también utilizamos *Hilbert SFC* para el mapeo multidimensional.

2.2. Entornos distribuidos P2P de computación

En esta sección se explicarán en los seis primeros apartados, las propuestas de computación distribuida Peer-to-Peer encontradas en la literatura. Finalmente, en el último apartado se explica el proyecto de computación P2P, implementado en la Universidad de Lleida, llamada CoDiP2P [CBR+08, CRB+09, CBGS11] y que constituye el punto de partida de la plataforma DisCoP presentada en este trabajo.

2.2.1. El proyecto CompuP2P

CompuP2P [GSS06] es una arquitectura peer-to-peer implementada mediante Java, diseñada para establecer un sistema de comercio de mercados de recursos computacionales, tales como ciclos de CPU, espacio en disco, etc. CompuP2P crea diferentes mercados según las características o capacidades de los recursos de computación de cada peer. Los nodos responsables del funcionamiento de un mercado se les denomina *MO*[5].

[4]SFC (Space Filling Curve): Es una familia de curvas que permiten pasar de un espacio multidimensional a un espacio unidimensional.

[5]MO (Market Owners): Propietarios de un mercado.

Los *MO* son dinámicamente reasignados a medida que los peers entran y salen de la red. Un MO realiza la tarea de negociador entre vendedores y compradores, y gestiona información de los vendedores. Esta información incluye la potencia de cómputo, espacio de disco, sistema operativo y versión, precio, etc. Después de recibir una petición de un cliente, el *MO* envía la información del vendedor que mejor encaja con los requerimientos del cliente.
CompuP2P emplea un esquema de red estructurada de tipo Chord, donde todos los nodos son distribuidos en diferentes mercados según la capacidad de cómputo contratada antes de entrar en el sistema. Al mismo tiempo, estos nodos también son propietarios de un mercado (MO) donde en un segundo y más simple *overlay* guardan las direcciones de las máquinas que han vendido una parte de su ciclos de cómputo en este mercado.
CompuP2P está diseñado para incentivar a los usuarios a compartir sus recursos de cómputo y controlar a los usuarios maliciosos. Con estos objetivos emplea conocimientos de teoría de juegos y microeconomía para poner un precio a los recursos computacionales.

Relevancia respecto de nuestra propuesta
Por el echo de emplear una topología estructurada como Chord, la hace más escalable que muchos de los otros entornos distribuidos P2P. El principal problema de esta topología es que contrata ciclos de cómputo a unas máquinas no dedicadas, donde no hay una base fiable para que éstas tengan los ciclos disponibles cuando sean alquiladas por usuarios compradores. La lógica y sentido de su estructura de crear una economía de mercado sobre recursos no dedicados puede alejarse de la potencia real de los recursos puestos en venta. De modo similar, nuestra propuesta utiliza la idea de mercados para agrupar los recursos computacionales con similares características. De todos modos, el diseño de nuestra propuesta va más allá, y no contempla solo la ordenación de mercados por ciclos de CPU, sino también por otros recursos como: memoria, disco, ancho de banda, etc. Cabe destacar que en el capítulo 3 se describe en detalle el sistema de cómputo propuesto, su arquitectura, capacidades y características principales.

2.2.2. El Proyecto JXTA

El Proyecto JXTA [ABJM04, AJN05] es una plataforma Peer-to-Peer de código abierto creada por Sun Microsystems en Febrero del año 2001, bajo la dirección de Bill Joy y Mike Clary y con la colaboración de investigadores de varias instituciones académicas. JXTA es una plataforma modular que define un conjunto de protocolos basados en XML que permite que se comuniquen de forma descentralizada un amplio rango de dispositivos (computadoras, teléfonos móviles, PDA, sensores electrónicos) para el desarrollo de un amplio rango de servicios y aplicaciones basadas en P2P.
La tecnología JXTA es el framework P2P más completo que existe actualmente. Está basado en un conjunto de protocolos abiertos simples, por lo tanto, permite que sea portado a cualquier lenguaje de programación moderno, Java, C/C++, Perl, etc. La versión implementada con el lenguaje Java es la más estable de todas. En JXTA, todos los nodos conectados mediante una red virtual *ad-hoc* pueden conectar con el resto de peers sin importar la ubicación, el tipo de dispositivo, o el sistema operativo. Además, permite conectar nodos entre sí aunque éstos se encuentren detrás de firewalls y enrutadores o utilicen distintos protocolos de transporte. Así, el acceso a los recursos en la red no está limitado a las incompatibilidades de la plataforma o a las restricciones de la arquitectura jerárquica cliente-servidor.
Además, los servicios JXTA pueden ser implementados para interoperar con otros servicios, facilitando el desarrollo de nuevas aplicaciones P2P. Por ejemplo, un servicio de comunicación P2P de mensajería instantánea puede ser fácilmente agregado a una aplicación P2P de compartición de ficheros si es que ambos soportan protocolos JXTA.
Hay que destacar que al ser un framework implementado con código abierto, permite que los desarrolladores de aplicaciones P2P puedan utilizar de forma gratuita todos los servicios de JXTA, incluso contribuir en el proyecto.

La topología de JXTA es estructurada e híbrida, donde el rol de superpeer está asignado a un grupo de peers llamados *rendevouz peers* que guardan información sobre los recursos compartidos y publicados por el resto de peers. Asimismo, la topología tiene una forma abstracta, admitiendo cualquier forma mientras que los rendevouz peers se puedan encontrar entre ellos y tengan un

alcance de toda la red.

Relevancia respecto de nuestra propuesta
JXTA, al ser un framework de desarrollo de aplicaciones P2P, ayuda a diseñar aplicaciones P2P ya que el motor y estructura del tipo de red, protocolos de comunicación y servicios de búsqueda ya están resueltos. No obstante, no existe ningún problema si se quiere realizar una nueva arquitectura P2P partiendo desde cero e implementando un nuevo protocolo de búsqueda o de resolución de direcciones, porque JXTA permite seleccionar servicios o librerías que interesan y descartar los que no interesan. Asimismo, nuestro trabajo va más allá puesto que sobre la arquitectura propuesta se definen una serie de funcionalidades, que permiten ordenar y gestionar los recursos computacionales dispersos en la red de un modo más eficiente.

2.2.3. El framework P2PComp

P2PComp [JdSJ10] es un framework dedicado al cómputo distribuido y paralelo. Está implementado mediante un conjunto de clases Java y utiliza la librería JXSE como infraestructura para crear sistemas P2P. JXSE es una implementación Java del estandard JXTA, descrito en la sección 2.2.2.
P2PComp utiliza un modelo de cómputo donde todos los peers pueden realizar las mismas funciones y no hay distinción entre nodos suministradores de trabajos, workers y peers monitores[6].
P2PComp utiliza funcionalidades de JXTA para crear catálogos de peers y publicarlos en la red. Cada peer se distingue mediante un identificador creado con la función Hash del DHT interno de la librería JXTA.
Al principio de la ejecución, un peer de P2PComp crea un anuncio personalizado que informa sobre su capacidad de procesamiento, la media de carga, el número de procesadores (o cores) y la capacidad de memoria. Después de esto, a través de un mensaje simple basado en un protocolo *ICMP*, cada peer crea una matriz que almacena el coste de comunicaciones, es decir, informa-

[6]Peers monitores: Son peers que monitorizan el estado del sistema de cómputo distribuido para obtener resultados estadísticos de la ejecución de un trabajo, la red, etc.

ción sobre el *Round-trip Time*[7] de todos los nodos de la red. De este modo, accediendo a esta matriz, el peer puede identificar y transmitir aplicaciones a otros peers con una baja latencia de comunicación, minimizando la sobrecarga de comunicación.

Relevancia respecto de nuestra propuesta
P2PComp no es escalable porque gestiona una matriz con los costes de comunicaciones de todo el sistema es intratable, si éste crece a una escala de millones de peers. Del mismo modo, P2PComp se centra más en el diseño y programación de clases que no a nivel topológico. Esto hace pensar que esta plataforma aún está en una fase inicial. Muchos requisitos contemplados en nuestra plataforma no se pueden implementar en P2PComp, como por ejemplo, escalabilidad y capacidad de hacer búsquedas multi-atributos, etc.

2.2.4. P2PCompute

El sistema P2PCompute [MA07] se basa en una infraestructura de cómputo que soporta un modelo propio de tres capas: la capa cliente, la capa servidor y la infraestructura. El funcionamiento se basa en que los nodos obtienen una lista de servidores de cómputo a través de una infraestructura de servidores de registros UDDI.
Los servidores de registros UDDI funcionan como servidores de cómputo dedicados a ejecutar tareas. Así, los clientes pueden conectarse a los nodos servidores y enviarles tareas para ser procesadas. Los nodos del servidor ejecutan las tareas y al final envían los resultados al cliente. La finalidad de P2PCompute es implementar una infraestructura de negocio pseudo-dedicada capaz de proveer a los clientes la potencia de cómputo solicitada, con servidores de cómputo que satisfagan sus requerimientos.

[7]Round-trip time (RTT): Tiempo empleado para enviar y retornar un mensaje de un nodo a otro a través de la red.

Relevancia respecto de nuestra propuesta
La propuesta P2PCompute no es un sistema distribuido P2P puro, sino más bien híbrido, lo cual consideramos que se desvía un poco de la filosofía Peer-to-Peer. Además, hay mucha distinción entre los peers porque no todos tienen las mismas funciones. Algunos tienen roles de compartir cómputo, otros tienen solo el rol de ejecutar tareas y otros de hacer de directorio de recursos. Además, cuando el sistema crece mucho, puede llegar a saturarse debido a la demanda efectuada por los nodos clientes.

2.2.5. P2PDisCo: Sistema de cómputo distribuido basado en Chedar P2P

Peer-to-Peer Distributed Computing (P2PDisCo) [NVW+05] ha sido desarrollado para ejecutar aplicaciones intensivas de cómputo en estaciones de trabajo, básicamente de universidades. Se basa en la idea de que las estaciones de trabajo de las universidades permanecen ociosas la mayor parte del tiempo, de modo que se puede usar su potencia de cómputo disponible durante períodos de tiempo que no interfieran en el rendimiento habitual de los computadores. P2PDisCo es una capa de software que trabaja por encima del middleware Peer-to-Peer Chedar. Cada nodo, en Chedar, contiene un identificador único y una base de datos de recursos compartidos (ficheros, hardware, software) locales. Además, los nodos mantienen una lista de nodos vecinos, que se actualizan según el número de respuestas a consultas enviadas a los nodos vecinos. P2PDisCo utiliza un algoritmo de búsqueda a ciegas para localizar máquinas ociosas de la capa inferior Chedar, llamado *Breadth-First searching algorithm* (BFS[LCP+05]). Los trabajos, programados en Java, requieren que la interfície sea específica del framework P2PDisCo, como los métodos lanzar, parar y testear tareas, porqué posteriormente serán invocadas por P2PDisCo durante la ejecución.
Actualmente, este sistema de cómputo se utiliza para acelerar el entrenamiento de redes neuronales con algoritmos evolutivos.
En la Fig. 2.6 se puede ver un esquema de los pasos necesarios para distribuir tareas entre los nodos Chedar y en su finalización, devolver los resultados.

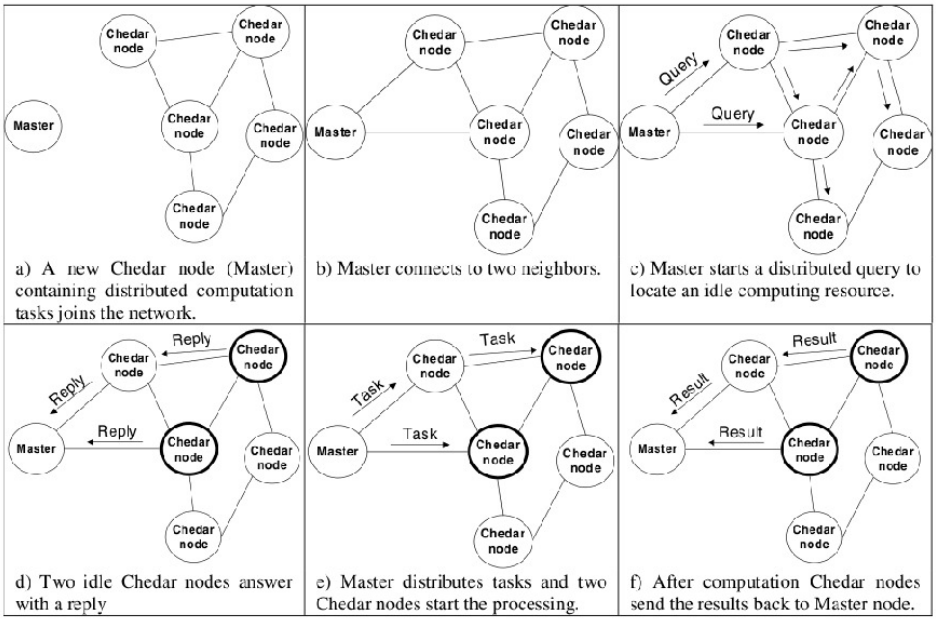

Figura 2.6: P2PDisCo: Proceso de distribuir las tareas y recoger los resultados. [NVW+05]

2.2.6. ParCop: Un sistema P2P descentralizado de computo distribuido

ParCop[ADT04] es un sistema P2P descentralizado, implementado totalmente en Java. No existe un único punto central de control; los datos y las tareas fluyen libremente por los recursos computacionales (peers) sin tener que pasar por ningún servidor central. ParCop se ejecuta en cualquier máquina y sistema operativo que tenga activado Java, sin necesidad de applets ni navegadores Web adicionales. Sus características principales son:

- Dinamismo: ParCop puede trabajar eficientemente en un entorno dinámico, donde miles de peers entran y salen libremente.
- Escalabilidad: ParCop aprovecha los recursos computacionales de miles de máquinas. Ni los recursos del sistema ni la red se saturan porque la arquitectura es totalmente descentralizada.
- Seguridad: ParCop asegura que los recursos computacionales estén

Relevancia respecto de nuestra propuesta
Este sistema emplea el middleware Chedar que está basado en el diseño de redes no estructuradas. Por lo tanto, tiene problemas de escalabilidad como la mayoría de redes no estructuradas. Sus algoritmos de búsqueda se basan en realizar consultas a ciegas por todos los nodos hasta alcanzar el tiempo máximo de vida TTL[8] del mensaje. Esto implica que no asegura que todos los nodos existentes puedan ser localizados. Por otro lado, aplica una heurística de ordenación de nodos según el número de respuestas transmitidas y consultas efectuadas. Esto ayuda a hacer la topología más eficiente y evitar así flooding por la red. Su finalidad se basa en encontrar la cantidad de nodos libres necesarios, independientemente de sus capacidades computacionales, para ejecutar un determinado trabajo. Después ejecuta las tareas y finalmente devuelve los resultados al nodo solicitante. Su servicio de búsqueda está diseñado solo para encontrar nodos voluntarios o libres para ejecutar tareas. De este modo, se hace más fácil implementar un sistema escalable y tolerante a fallos, ya que el orden de los nodos o de sus recursos no importa. Este sistema se aleja de los objetivos iniciales de nuestra propuesta, orientada a redes estructuradas, que son más eficientes para diseñar un sistema que sea capaz de gestionar los recursos de la red según sus capacidades computacionales y así implementar servicios de búsqueda rápidos y con QoS.

disponibles solo para usuarios de confianza. Por lo tanto, un usuario que comparte su máquina estará protegido de ataques maliciosos.

- Rendimiento: ParCop, según sus autores, es capaz de obtener igual o mejor *speedup* que los de otros sistemas P2P distribuidos.

ParCop [ADT04] utiliza una red no estructurada basada en conexiones arbitrarias, donde los peers solo están conectados con los peers que conocen. No obstante, en la ejecución de tareas emplea un mecanismo de roles *master/worker*. El *master* peer es el encargado de distribuir las tareas, coleccionar los resultados y devolverlos al usuario. El *worker* recibe tareas del *master* peer, ejecuta las tareas y devuelve el resultado al *master*.
Igualmente, su finalidad se basa en encontrar nodos libres para ejecutar tareas, independientemente de sus capacidades computacionales.

Relevancia respecto de nuestra propuesta
ParCop es un sistema escalable y tolerante a fallos. Además aplica técnicas para no saturar la red mediante flooding de mensajes cuando el master tiene que encontrar un worker libre. Este sistema, respecto a nuestra propuesta, se ve como una red de peers no estructurada y aleatoria que sigue el modelo *master/worker*, capaz de enviar (*masters*) o ejecutar (*workers*) tareas. En cambio, nuestro objetivo va más allá porque busca implementar un sistema de cómputo distribuido con calidad de servicio donde el sistema sea capaz de ordenar los recursos según sus capacidades para realizar búsquedas de recursos de forma eficiente en un tiempo máximo acotado.

2.2.7. Rheeve: Una plataforma de cómputo P2P Plug-n-Play

Rheeve [kPC02] es una plataforma de cómputo P2P que sigue la filosofía de JXTA e implementa sus especificaciones y algunas mejoras más. El objetivo de esta plataforma es dar soporte a los programadores para el desarrollo de aplicaciones P2P eficientes y tolerantes a fallos, utilizando protocolos de comunicación HTTP y la interacción con los servicios de *Rheeve*. Rheeve está construido bajo un modelo de arquitectura modular, con mecanismos escalables para una conexión eficiente de peers y servicios de búsqueda de recursos. Además, *Rheeve* utiliza una interfície de programación visual que, junto con la tecnología *JavaBeans*, soporta el diseño de aplicaciones de estilo Plug-and-Play[9]. Sus características principales son:

- Sistema de arquitectura modular jerárquica.
- Contiene herramientas gráficas con interfaz visual para el diseño de aplicaciones P2P.
- Establecimiento efectivo y tolerante a fallos de las conexiones entre peers.

[9]Plug-and-Play: JavaBeans permite activar un conjunto de clases para ser agrupadas como un módulo y no como un paquete "jar". De este modo la implementación de una aplicación puede ser realizado enlazando módulos y los arcos entre módulos son eventos.

- Gestión eficiente de los recursos del sistema.
- Mecanismos de distribución y ejecución de trabajos empleando diferentes lenguajes de programación, plataformas y protocolos heterogéneos.
- Comunicación a través de la tecnología *HTTP-enabled*, utilizada para activar la computación Peer-to-Peer a través de Internet.

La topología de la plataforma *Rheeve* es una propuesta híbrida, la cual facilita la conexión y el descubrimiento de nodos. *Rheeve* selecciona como servidores (superpeers), nodos bien conocidos para hacer públicas sus direcciones IP. El resto de los nodos están conectados en subredes.
Además, aplica broadcasting y utiliza agentes móviles para localizar nuevos recursos y evitar grupos de nodos disjuntos. Los agentes móviles son procesos auxiliares que viajan a través de la red usando la lista de peers conocidos por los superpeers, en busca de nuevos nodos y recursos no publicados.

Relevancia respecto de nuestra propuesta
Al igual que las DHT, Rheeve tampoco emplea mecanismos de distribución de recursos compartidos. Este sistema se centra en el diseño e implementación modular de clases y protocolos de comunicaciones, utilizados para sincronizar las comunicaciones de los peers. A nivel topológico, *Rheeve* utiliza una red híbrida, algo mejor que las anteriores propuestas y bastante similar a la nuestra. De este modo evita la saturación mediante flooding de mensajes por la red ya que los recursos están publicados en los superpeers. No obstante, nuestra propuesta es mejor porque no solo contempla evitar la congestión en la red, creando una organización de superpeers, sino que también posee un buen esquema de búsqueda para realizar búsquedas multi-atributo y en rangos.

2.2.8. CoDiP2P

CoDiP2P es una arquitectura P2P de cómputo distribuido implementada por el grupo GCD (Grup de Computació Distribuida) de la Universidad

de Lleida, junto con la empresa Indra[10]. Los inicios de esta tesis doctoral se basaron también en el diseño de esta topología. Las publicaciones más importantes son [CBR+08, CRB+09, CBGS11], en donde se explica la motivación y el objetivo de *CoDiP2P*, su arquitectura, su escalabilidad, roles de los peers y sus principales funcionalidades. La topología de este entorno es jerárquica, en forma de árbol B-ario, donde cada nodo tiene B hijos. En la Fig. 2.7 se visualiza la topología de *CoDiP2P*. Cabe destacar que su topología se divide en niveles y a su vez, cada nivel en áreas, que son agrupaciones lógicas de peers.

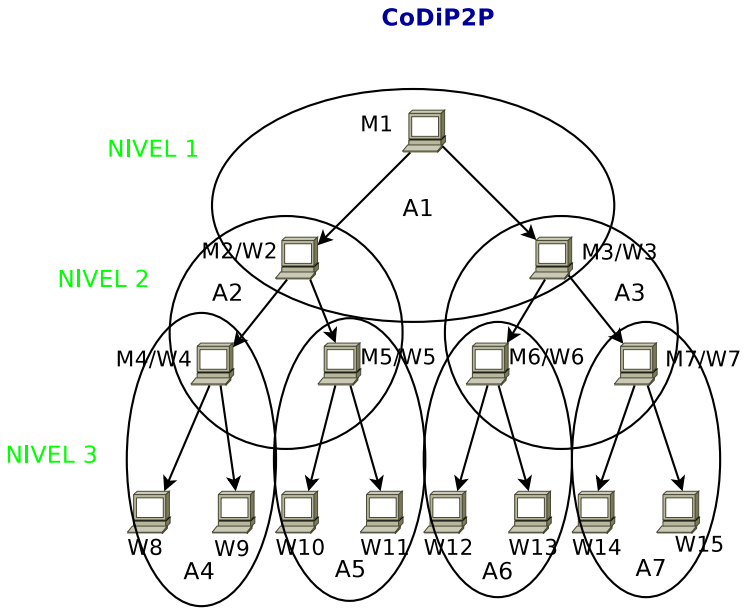

Figura 2.7: Topología CoDiP2P en forma de árbol.

Cada peer puede realizar dos tipos de roles: *manager* y *worker*. El rol *manager* controla y gestiona a los peers conectados inmediatamente en el nivel inferior mediante un subárbol. El rol de *worker* es el encargado de ejecutar tareas y enviar información sobre el estado de sus recursos a su manager.
Un área es un espacio lógico formado por un manager más sus workers hijos. Un área es es una organización lógica pensada para ejecutar localmente aplicaciones intensivas de cómputo y si es necesario en las áreas vecinas. En cada área, un subconjunto de workers con el rol de manager replicado, tienen

[10]Este proyecto fue soportado por MITyC-Spain bajo los contratos TSI-020100-2009-240, TSI-020100-2008-21 y FIT-340000-2007-44.

la misión de substituir al manager de su área cuando éste se desconecta del sistema.
Las funcionalidades de CoDiP2P son inserción, mantenimiento, salida, planificación y ejecución de peers. En la implementación inicial se han utilizado las funcionalidades de JXTA.

Relevancia respecto de nuestra propuesta
El objetivo principal de CoDiP2P fue diseñar una topología en forma de árbol B-ario para conseguir un coste logarítmico $\theta(\log_B(N \cdot (B-1)+1)-1)$ en la inserción, mantenimiento y salida de peers gracias a su estructura de árbol en base B. De este modo se consigue un rendimiento superior al $\log_2(N)$ en las búsquedas, además de un grado constante de enlaces respecto a Chord, Kadmelia y Viceroy. Al mismo tiempo, la estructura de árbol es óptima para controlar la mutabilidad de los recursos porque permite propagar los cambios de estado de los recursos de los peers de abajo a arriba, en todo el árbol, a través del algoritmo de mantenimiento. No obstante, el diseño no contempla la capacidad de realizar búsquedas multi-atributo, ni búsquedas aproximadas, porque los recursos de los peers no están ordenados por sus características computacionales. Además, el sistema no es muy escalable debido al poco balanceo de carga de comunicaciones que provoca la congestión en el root peer. Incluso, para aumentar la escalabilidad y resistencia a fallos se propuso un algoritmo de reestructuración del árbol [CBGS11].

Capítulo 3

DisCoP: Una nueva arquitectura de cómputo P2P

En el capítulo anterior se describieron las principales propuestas de la literatura en el campo de la computación P2P. A partir de las mismas, en este capítulo se presenta la principal aportación de esta tesis, la arquitectura del sistema de cómputo distribuido P2P, llamada *DisCoP* (*Distributed Computing P2P Platform*).

En la sección 3.1 se explica el esquema general de nuestra arquitectura. En la sección 3.2 se describe en detalle los principales niveles que componen la arquitectura *DisCoP*. A continuación, en la sección 3.3 se presentan las mejoras realizadas sobre la arquitectura propuesta, con objeto de adaptarla a las necesidades de un entorno de computación P2P. En la sección 3.4 se encasilla nuestra arquitectura dentro de la clasificación dada en la sección 1.3 del capítulo 1. En la sección 3.5 se evalúa, a nivel de topología, el rendimiento de la arquitectura *DisCoP*, comparándola con otras propuestas ampliamente utilizadas en la literatura. Finalmente, la sección 3.6 propone un nuevo método para evaluar la *Localidad*, orientado a entornos de computación P2P, y se aplica a la arquitectura propuesta, comparándola con otras topologías clásicas.

3.1. Análisis Previo

Con objeto de tener una idea global del propósito de la arquitectura *DisCoP*, la Fig. 3.1 muestra el esquema general de la arquitectura *DisCoP*, representando la interconexión de los diferentes niveles o capas que configuran el sistema. En este esquema se puede apreciar que la arquitectura *DisCoP* está compuesta por tres niveles jerárquicos: *Hilbert SFC*, *Bruijn* y *Mercados*. De este modo se pretende agrupar eficientemente los nodos de entrada en los mercados que les corresponde, según sus capacidades de cómputo.
Un sistema de cómputo distribuido debe ofrecer a los usuarios del mismo, un conjunto de recursos de cómputo: CPUs, memoria, ancho de banda, disco, etc. *DisCoP* tiene que ser capaz de gestionar todos estos recursos de cómputo que caracterizan a los nodos que integran la plataforma. Sin embargo, si se desea optimizar la búsqueda de recursos en los nodos, el control de múltiples atributos por nodo supone aumentar la complejidad de la arquitectura P2P. Acorde con este objetivo, la arquitectura DisCoP está dividida en tres niveles/capas. El primer nivel (denominado Hilbert), realiza la codificación de los múltiples atributos computacionales de un nodo en un único código H_i. A continuación, el segundo nivel (denominado Bruijn), se encarga de la localización del Mercado, representado por el código H_i, en la red de interconexión de Mercados. Finalmente, el tercer nivel (denominado Mercados), tiene el objetivo de insertar el nuevo nodo dentro del Mercado asociado al código H_i. Un Mercado es una topología P2P de tipo $B-Tree$ que contiene todos los nodos que ofrecen los mismos atributos de cómputo.

3.2. Arquitectura de DisCoP

Tal como se ha comentado en el análisis previo, la arquitectura de DisCoP está compuesta por tres capas principales: *Hilbert, Bruijn y Mercados*. El diseño de estas tres capas permite ordenar los nodos según sus recursos computacionales. El esquema detallado de la arquitectura de *DisCoP* se puede ver en la Fig. 3.2. A continuación se describe la idea principal de su funcionamiento.

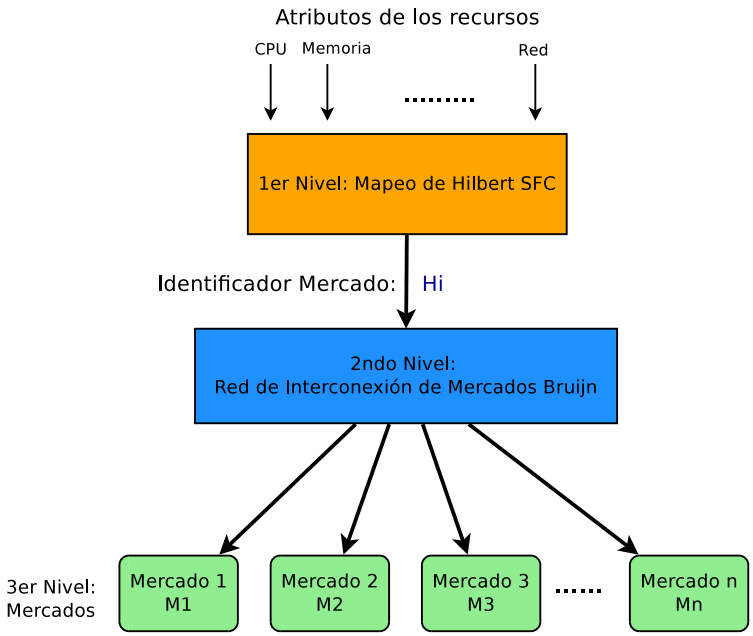

Figura 3.1: Ilustración del diagrama de flujo de interconexión del sistema P2P de cómputo distribuido *DisCoP*.

En primer lugar, los recursos de cómputo de los nodos insertados en *DisCoP* son evaluados mediante un *benchmark*[1] e indexados dentro de un rango de valores. De este modo, cada nodo es representado por unas coordenadas k-dimensionales $(X_1, ..., X_i, ..., X_n)$, donde n es el número de atributos y $X_i \in \mathbb{Z}^+$ es el valor del atributo i-ésimo de dicho nodo. A continuación, el primer nivel de *DisCoP*, mediante la curva Hilbert, mapea el vector n-dimensional en un único identificador Hilbert H_i. En el ejemplo mostrado en la Fig. 3.2 se observa como cada nodo es representado por $n = 2$ atributos, CPU y Memoria, de modo que el nodo con atributos $CPU = 3$ y $Memoria = 1$ se transforma, mediante la función Hilbert, en el identificador Hilbert H_{12} (Caso A de la Fig. 3.2). Posteriormente, el identificador H_{12} se utiliza para localizar el Mercado donde irá ubicado el nuevo nodo insertado dentro del grafo *Bruijn* (Caso B de la Fig. 3.2). Una vez localizada la posición H_{12} dentro del grafo Bruijn, el nodo se insertará en el Mercado que cuelga de dicha posición, Caso C de la Fig. 3.2. En este mismo ejemplo se puede observar como la estructura del Mercado,

[1]Benchmark: Herramienta de software que obtiene los componentes hardware de un computador y mide su potencia asignando una puntuación final.

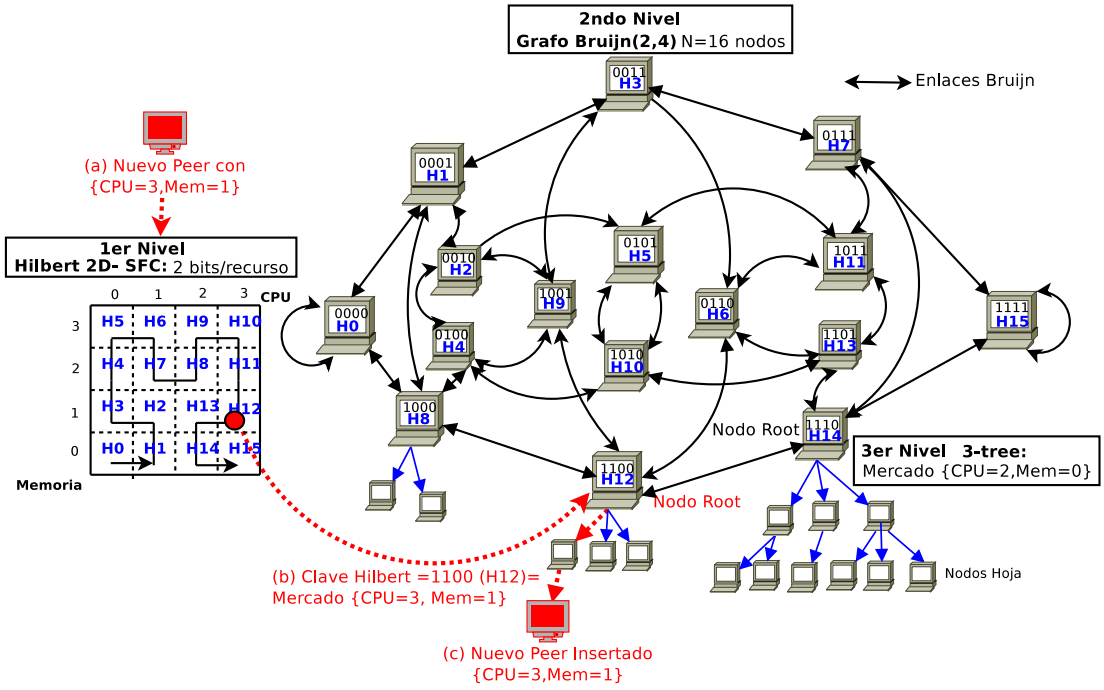

Figura 3.2: Ilustración de la arquitectura de tres capas *Hilbert + Bruijn + Mercados B-Tree* del sistema cómputo distribuido P2P *DisCoP*.

nivel 3 de DisCoP, se corresponde a un árbol del tipo $B-Tree$. En el caso de que no existiese ningún Mercado en la posición H_{12} del grafo Bruijn, el nuevo nodo insertado constituiría un nuevo Mercado.
A continuación se detallan cada uno de las tres capas/niveles que configuran *DisCoP*.

3.2.1. Primer nivel: Hilbert SFC.

Hilbert Multi-Dimensional Space-Filling Curve (SFC) es una curva multi-dimensional utilizada en nuestra arquitectura como función Hash para mapear un espacio multidimensional de datos dentro de un espacio lineal. La Fig. 3.3 muestra un ejemplo de dos curvas SFCs, *Hilbert* (izquierda) y *Peano/Zorder* (derecha). En ambas curvas se observa como un SFC es un hilo que recorre todos los puntos del espacio multidimensional, visitando cada uno de ellos una sola vez. Hay que destacar que las dos curvas SFC, mostradas en el ejemplo, han sido creadas con 2 atributos (dimensiones) y 3 bits/atributo. En el caso

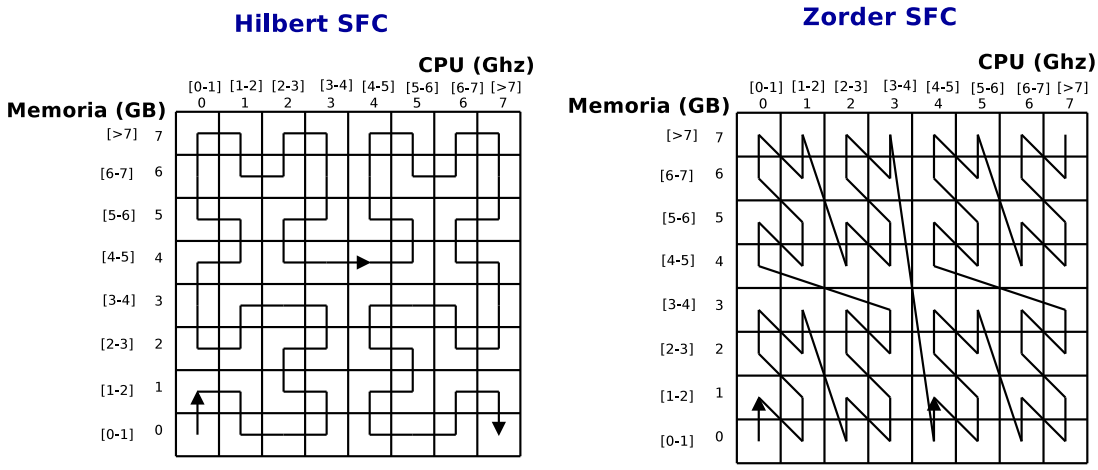

Figura 3.3: (Izquierda)Ejemplo de un *Hilbert SFC* y (Derecha) de un *Peano/Zorder SFC* de 2 atributos y 3 bits/atributo.

de la Fig. 3.3 (izquierda) se muestra un ejemplo de un espacio 2D (CPU y Memoria) de Hilbert, donde cada eje, CPU o Memoria, representa un rango de valores posibles. En el caso concreto de la computación P2P, la curva SFC se utiliza para identificar los n atributos que caracterizan a cada nodo mediante una simple clave, que representará la identificación del mercado donde tiene que ir ubicado dicho nodo. De este modo, el siguiente paso, que inserta un nodo con atributos multidimensionales dentro del grafo Bruijn, se realizará de un modo más eficiente.
En la literatura [MAK03] existen cinco **SFCs** ampliamente utilizadas: *Sweep*, *Scan*, *Peano*, *Gray* y *Hilbert*. De entre todas ellas, se ha seleccionado la curva Hilbert como primer nivel de la arquitectura *DisCoP* atendiendo a su propiedad de *clustering* [MJFS96]. Se entiende por *clustering*, la capacidad de que la localidad de objetos en un espacio multidimensional se conserve en el espacio lineal, hecho que permite realizar búsquedas aproximadas o en rangos sobre el grafo Bruijn de un modo muy eficiente.

3.2.2. Segundo nivel: Grafo Bruijn.

Como ya se comentó anteriormente, el grafo Bruijn ha sido seleccionado como segundo nivel de la arquitectura *DisCoP*. Un grafo Bruijn se caracteriza

por dos parámetros: (a) el grado k o número de enlaces de salida y de entrada de cada nodo (el grado k también designa la base numérica de la clave asignada a cada nodo) y (b) el diámetro d, es decir, la máxima distancia entre cualquier par de nodos (equivalente a la longitud de la clave de los nodos). De este modo, el número máximo de nodos del grafo es $N_{Bruijn} = k^d$. Siguiendo esta notación, para describir un grafo de un tamaño concreto emplearemos la nomenclatura Bruijn(k,d). En la Fig. 3.4 se muestra un ejemplo de grafo Bruijn(2,3) dirigido con $N_{Bruijn} = 8$ nodos. No obstante, en el capítulo 4 se empleará en los mecanismos de búsqueda de recursos diseñados para *DisCoP*, grafos Bruijn *no dirigidos*, es decir, grafos donde sus arcos o enlaces son bidireccionales, para así reducir la distancia entre nodos.

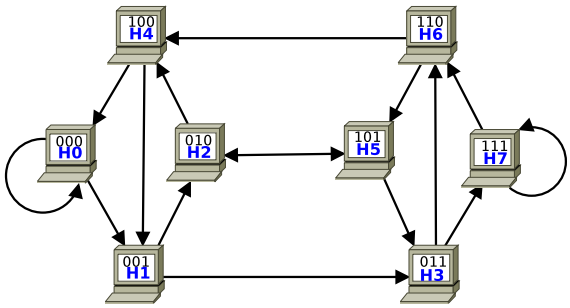

Figura 3.4: Grafo Bruijn(2,3) dirigido con $N_{Bruijn} = 8$ nodos.

El grafo Bruijn ha sido seleccionado como red de interconexión entre Mercados del sistema *DisCoP* porque es mucho más óptimo que otras topologías clásicas, como Chord, CAN, Pastry, etc... En primer lugar, Bruijn es uniforme, multi-direccional y simétrico. Estas propiedades permiten que el tráfico de mensajes sea balanceado a lo largo de la topología, evitando congestión y cuellos de botella.

Otra ventaja de Bruijn es que un nodo solo tiene que mantener un número pequeño y constante de enlaces, lo que hace que la topología tenga un coste de construcción bajo y sea más robusta. Asimismo, Bruijn se caracteriza por una elevada capacidad de localización de recursos con un coste $\log_k(N_{Bruijn})$, inferior al coste común $\log_2(N)$ de otras topologías, como Chord, Baton, Viceroy, etc.

En la Tabla 3.1 se pueden ver unos resultados comparativos de diversas

Grafo	Grado k	Diámetro
Baton	$h + c$	$log_2(N)$
Chord	$log_2(N)$	$log_2(N)$
Pastry	$2 \cdot k \cdot log_k(N)$	$log_k(N)$
Bruijn	k	$log_k(N)$
Viceroy	k	$2 \cdot \log_k(N)(1 - o(1))$

Grafo	Grado k				
	2	3	4	10	20
Baton	-	40	26	13	10
Chord	-	-	-	-	20
Pastry	-	-	-	-	20
Bruijn	20	13	10	6	5
Viceroy	31	20	16	10	8

Tabla 3.1: (Izquierda) Tabla de complejidades Grado-Diámetro de ciertas topologías P2P. (Derecha) Diámetro de topologías P2P con $N = 10^6$ nodos.

topologías P2P, incluyendo Bruijn, con respecto a su diámetro y a su complejidad grado-diámetro, para un tamaño fijo de $N = 10^6$ nodos. Acorde con estos resultados, la mejor relación grado-diámetro la obtiene Bruijn, donde con un grado 2 alcanza un diámetro igual o mejor que la mayoría del resto de topologías que requieren un mayor número de enlaces. En el caso de la red *Viceroy*, también de grado constante, se puede observar en la Tabla 3.1 (derecha) como la red de *Bruijn* también es más óptima en diámetro.

3.2.3. Tercer nivel: Mercados de tipo B-Tree.

En el capítulo 2 se explicó que los *Árboles B-Tree* son muy apreciados en el mundo de las bases de datos, sistemas de ficheros y en la programación, en general. Asimismo, en el mismo capítulo 2 se comentó que la investigación inicial de esta tesis doctoral se centró en el diseño de una topología P2P, denominada *CoDiP2P*, basada en árbol para controlar los recursos ociosos de los nodos. No obstante, el uso de esta topología para albergar un número masivo de peers no era viable, dado el elevado coste de su mantenimiento, su congestión en los niveles superiores y la ausencia de uniformidad en cuanto a su estructura. Esto comportaba que el nodo *root* tuviese una excesiva responsabilidad de gestión dentro de la estructura, lo cual resulta intratable por el sistema si se escala.
Con objeto de aprovechar el trabajo inicial realizado sobre esta topología se reutilizó este diseño en árbol para implementar los *Mercados* del sistema *DisCoP*, dado que los árboles tienen propiedades ideales para este caso. El tamaño del Mercado de tipo $B-Tree$ se limitó a un número fijo de nodos en *DisCoP*, evitando de este modo el problema de la congestión en capas superiores del

árbol.
En la implementación de la topología de los Mercados, DisCoP distingue tres roles diferentes de peer. En la Fig. 3.5 se puede apreciar la estructura lógica de un Mercado de *DisCoP* con los roles asignados a cada peer. A continuación se describen cada uno de los roles de los Mercados, así como las partes que lo componen:

- **Área** (A_i): Es el conjunto lógico integrado por el manager del área, más los B *workers* (peers hijos) conectados en el nivel inferior. Estas áreas son los espacios lógicos que agrupan los workers, encargados de ejecutar las tareas. De este modo, el manager tiene constancia, en todo momento, de la ejecución de un trabajo dentro de su área.

- **Manager** (M_i): Este rol es el responsable de la gestión de un área de peers. Por lo tanto, cada manager tiene asignados los peers hijos conectados en el nivel inmediatamente inferior, los cuales se denominan *workers*. Este rol sirve para controlar los recursos disponibles que tienen los *workers* y al mismo tiempo suministrar información a los nodos manager ubicados en niveles superiores. Asimismo, el manager se encarga de realizar tareas de planificación de trabajos paralelos, decidiendo según el estado de los nodos, cuales son buenos candidatos para ejecutar las tareas.

- **Worker** (W_i): Este rol es asignado a los peers que son hijos de un peer manager del nivel inmediatamente superior. Los peers con este rol son los encargados de ejecutar tareas. Asimismo, cada worker debe enviar información sobre sus recursos compartidos a su nodo manager. Esta información representa el estado actual de la capacidad de sus recursos, como puede ser la potencia de procesador, porcentaje libre de la memoria principal o de disco, etc. El estado de los workers se actualiza periódicamente cada T_{Tree} segundos, mediante un algoritmo de mantenimiento que transmite la información desde los nodos hoja del árbol, hasta el nodo root del mismo. En el capítulo 4 se explica su funcionamiento.

- **Manager Replicado** (MR_i): Este rol se asigna a uno o más nodos de

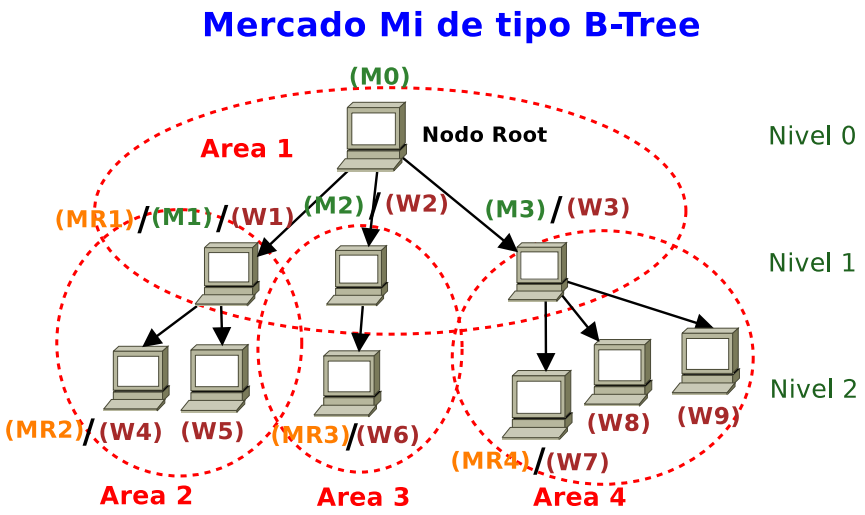

Figura 3.5: Mercado de tipo $3-Tree$ con sus roles managers, workers y managers replicados.

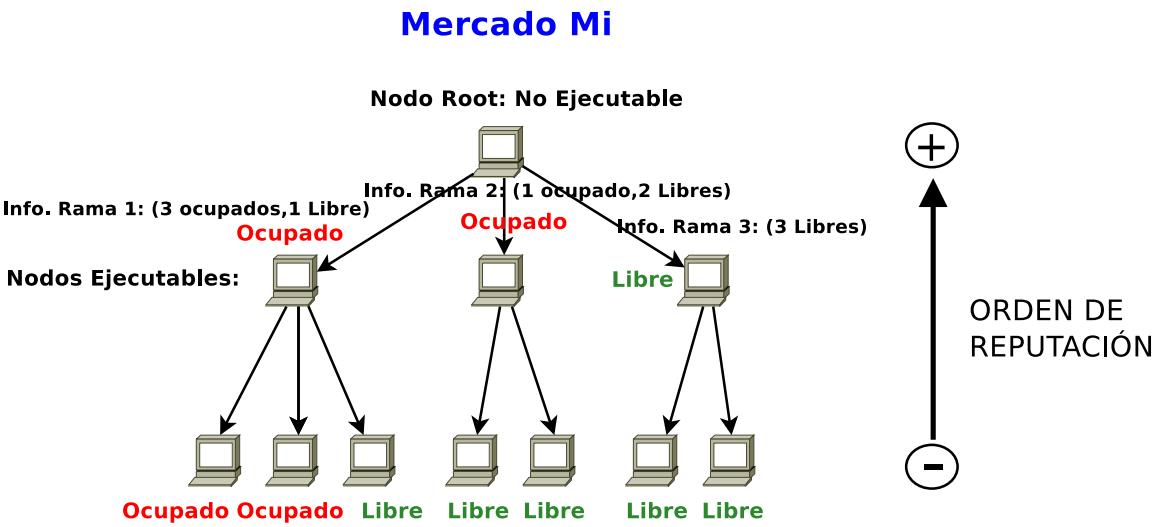

Figura 3.6: Orden ascendente de la reputación de los peers en un Mercado y estado de ejecución de los nodos.

cada área y sirve para reemplazar al manager cuando éste se desconecta de la red. Para realizar esta tarea es necesario que los managers replicados tengan información replicada de control del manager, para poder sustituirlo en caso de necesidad. Esta información se actualiza acorde con el algoritmo de mantenimiento presentado en el capítulo 4.

La búsqueda de un nodo para ejecutar una tarea en una topología de árbol tiene un coste logarítmico, $\log_{|Area|}(N_{Tree} \cdot (|Area| - 1) + 1))$. En este caso, $|Area|$ equivale al parámetro B del árbol $B-Tree$ y N_{Tree} es el número total de nodos del Mercado. Del mismo modo, el grado de enlaces ($|Area|$) por nodo es también constante, hecho que ayuda a mejorar la conectividad.
Además, otra buena ventaja del árbol es que su topología jerárquica permite acceder a los recursos computacionales de los nodos de manera eficiente. Por ejemplo, como se puede ver en la Fig. 3.6, la topología en árbol permite propagar rápidamente la información de estado desde los niveles inferiores hasta los niveles superiores del árbol, hecho que facilita en gran medida la planificación de tareas. Asimismo, partiendo del nodo *root* crecen exponencialmente, por cada nivel que se desciende en el árbol, el número de posibles caminos para escoger los nodos para la ejecución de tareas.
Otro caso óptimo, e ilustrado de nuevo en la Fig. 3.6, es el hecho que el árbol permite que las máquinas con una alta reputación estén localizadas en los niveles más altos de la jerarquía (cerca del nodo *root*), mientras que nodos con reputación más baja estén localizados en niveles bajos. La reputación se define como la antigüedad de un nodo en el sistema, es decir, el tiempo que ha permanecido conectado desde que entró a formar parte de él. De este modo, se garantiza una gran robustez del sistema, ya que los nodos menos mutables (o con menor dinamismo) estén situados en los niveles superiores del árbol. Este diseño es también de gran utilidad en el diseño de algoritmos de scheduling eficientes. Sin embargo, su diseño se deja como trabajo futuro puesto que no está en los objetivos iniciales de esta tesis.
Finalmente comentar que para el caso concreto de la arquitectura DisCoP, el Mercado de tipo *B-Tree* tiene asociado una clave identificadora H_i que representa el código Hilbert retornado por el primer nivel de la jerarquía *DisCoP*. Su objetivo principal es agrupar bajo un mismo identificador a todos los nodos que tienen las mismas características de cómputo y usarlo como un solo nodo en la red de la capa superior Bruijn.

3.3. Mejoras de diseño

En esta sección se explica dos mejoras realizadas en el diseño de la segunda capa de *DisCoP* con objeto de mejorar el rendimiento de la misma. En la sección 3.3.1 se describe la virtualización de nodos y en la 3.3.2 se presenta otra mejora que consiste en añadir enlaces Hilbert en el grafo Bruijn.

3.3.1. Virtualización de nodos

El grafo Bruijn es óptimo para realizar búsquedas con coste logarítmico, dado que presenta una congestión balanceada y maximiza la tolerancia a fallos, siempre y cuando el grafo esté completo, es decir, contenga todos los nodos físicos. En una red P2P es muy común que la red no esté completa del todo debido a su alto dinamismo, pero principalmente en nuestro caso, dado que los mercados se van creando a medida que van entrando peers ofertando nuevos recursos. En el caso del grafo Bruijn, cuando no está completo, surgen diferentes problemas, como son el aislamiento de grupos de peers o la falta de determinados enlaces que puede provocar que la búsqueda de un mercado específico suponga un número de saltos muy elevado. Con objeto de mejorar estas deficiencias, en este apartado se explica una solución basada en la virtualización de nodos.

Por virtualización de nodos se entiende que un nodo físico Bruijn puede tener asignado, aparte de su propia clave, un rango de claves de peers no existentes (nodos virtuales). De este modo, cada nodo físico H_i guarda su propia clave y todas las claves relacionadas con los nodos virtuales existentes entre el nodo H_i y su nodo físico predecesor más próximo H_{i-e}, siguiendo el grafo Bruijn. Además, por cada clave virtual asignada, el nodo físico tiene que gestionar todos los enlaces con los nodos vecinos asociados a la clave virtual. En la Fig. 3.7 se ilustra un ejemplo de un grafo Bruijn(2,4) no completo con cuatro nodos físicos, que contienen nodos virtuales. Por ejemplo, el nodo físico H_{11} contiene los nodos virtuales H_8, H_9 y H_{10}. Asimismo, en la Fig. 3.8 se ilustra el esquema de claves que tendría que mantener un nodo físico para gestionar sus nodos virtuales asociados. Dentro de un nodo físico H_i, por cada clave virtual que tiene asignada H_{i-e}, se crea una nueva subclase de nodo que

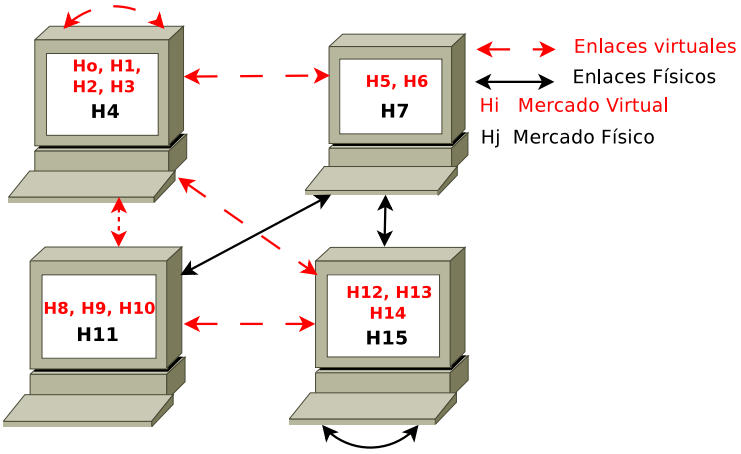

Figura 3.7: Grafo Bruijn(2,4) no completo, con nodos y enlaces virtuales.

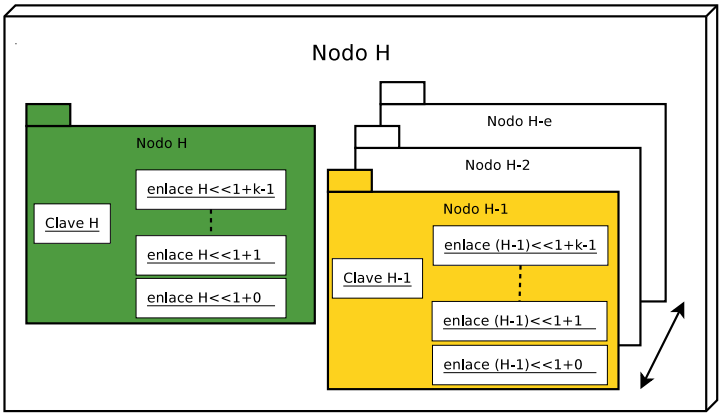

Figura 3.8: Esquema de un nodo físico con sus nodos virtuales asociados.

contenga la información necesaria para suplantar el nodo físico ausente, así como sus enlaces físicos.

Hay que destacar que el número total de enlaces físicos y virtuales que tiene que gestionar un nodo físico no es muy elevado. En la sección 3.5.2 se analizan los enlaces físicos y virtuales que tiene que gestionar un nodo.

La ventaja principal del uso de la virtualización es que el número de saltos en la búsqueda de recursos es como mucho el diámetro del grafo (número de dígitos de la clave Bruijn), y que es $\log_k(N_{Bruijn})$.

3.3.2. Enlaces Hilbert

Tal como se ha comentado en el apartado 3.2.1, la función *Hilbert SFC* tiende a asignar claves próximas a mercados con recursos similares. Para preservar esta localidad, en el grafo Bruijn hay que añadir un enlace extra por cada nodo del grafo. Este enlace, llamado *Hilbert*, simplemente conecta un nodo H_i con su nodo sucesor H_{i+1}, siguiendo la curva Hilbert y así sucesivamente para todos los nodos del grafo ($H_{i+1} \rightarrow H_{i+2}$, ..., $\rightarrow H_{i+n}$). De este modo, la plataforma preserva la localidad, intrínseca en la curva Hilbert, para realizar búsquedas aproximadas y en rango, hecho que supondrá disminuir el número de saltos para realizar dichas búsquedas.
Con objeto de determinar el rendimiento de los enlaces Hilbert sobre la topología de Bruijn, en el apartado 3.6 se explica detalladamente el concepto de *Localidad* de un grafo y se realiza un análisis experimental sobre el grado de localidad para diferentes dimensiones de grafos Bruijn, con y sin enlaces Hilbert, comparando su localidad con otras topologías, como es el caso de Chord.
La Fig. 3.9 ilustra como queda la topología Bruijn(2,3) añadiendo los enlaces Hilbert mencionados.

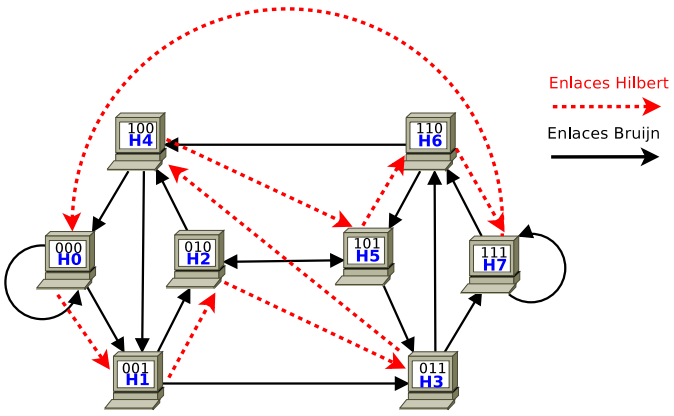

Figura 3.9: Grafo Bruijn(2,3) con enlaces Hilbert.

3.4. Clasificación de la arquitectura DisCoP

Según las clasificaciones realizadas en los capítulos 1 y 2 del presente documento, la arquitectura *DisCoP* se puede catalogar de la siguiente manera:

- Es una red de tipo *DHT completamente modificada*, ya que utiliza otro tipo de función de mapeo, diferente al típico DHT con función Hash uniforme aleatoria. Su función de mapeo se basa en el utilización de una curva Hilbert, de modo que crea una sola clave identificativa a partir de múltiples valores que identifican un nodo.

- Es una *red estructurada*, dado que la topología se construye partiendo de unas reglas para decidir cuales son sus vecinos y no se construye a partir de conexiones aleatorias.

- Es una *red híbrida*, donde un conjunto de nodos enlazados entre sí actúan como *superpeers* y agrupan cada uno de ellos, en un nivel inferior, a un grupo de peers según sus capacidades computacionales.

- Es una topología con capacidad para realizar *búsquedas multi-atributo*, en *rangos* y *aproximadas*.

- Es una arquitectura *escalable*, *tolerante a fallos* y *robusta*.

3.5. Análisis de rendimiento

En este apartado se explican las primeras pruebas experimentales de simulación realizadas sobre la plataforma para evaluar su rendimiento a nivel de arquitectura comparándolo con el de otras topologías P2P clásicas de la literatura, como es el caso de Chord [SMK+01] y Baton [JOV05]. Estas pruebas consisten en evaluar el número de enlaces físicos y virtuales que tiene que gestionar el grafo Bruijn para varios tamaños de red. Además, en la sección 3.5.3 se evalúa la tolerancia a fallos de *DisCoP*.

3.5.1. Entorno experimental

Para realizar el análisis experimental se han simulado todos los componentes de la arquitectura DisCoP. Para ello se ha utilizado el simulador GridSim [BM02]. GridSim está diseñado sobre el simulador de eventos discretos *SimJava* [MH98] y permite modelar mecanismos de planificación y comunicación en entornos Grid, empleando el lenguaje de programación multiplataforma *Java*. Aunque en un principio el simulador GridSim está pensado para entornos Grid, también es aplicable para simular entornos P2P, y en nuestro caso particular, ha sido de gran ayuda para modelar el comportamiento de la arquitectura DisCoP. Por ejemplo, entidades tales como peers, han sido implementadas mediante *threads*, de modo que a cada *thread* se le ha asignado un recurso Grid compuesto de una sola máquina. Además, todas las entidades están conectadas por enlaces de red y *routers*, donde se puede especificar su ancho de banda, su latencia y su política de *routing*.

A continuación se describe el entorno experimental utilizado:

- El sistema de simulación se ha implementado mediante un ordenador Macintosh (Apple) con biprocesador Intel Xeon, con un total de 8 núcleos a 2.8 GHz y una memoria principal de 4 Gbytes.
- El sistema operativo empleado ha sido Ubuntu Server de 64 bits.
- Las simulaciones experimentales se han realizado partiendo de los siguientes parámetros:
  - La red de interconexión tiene una latencia de 100 milisegundos, un ancho de banda de 1 Mbps y una MTU de 1.500 bytes.
  - El número de nodos simulados varía aleatoriamente entre 100 y 33.554.432 nodos.
  - El protocolo de comunicación de los nodos era en modo no conectado, equivalente a un *UDP*[2].

[2]UDP (User Datagram Protocol): Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera

- La entra y salida de peers se realiza de forma uniforme según el tipo de prueba o mediante el uso de funciones de distribución como Poisson y Normal.

3.5.2. Número de enlaces físicos

La Fig. 3.10 muestra la relación del número medio de enlaces virtuales por nodo con respecto al número de enlaces físicos, cuando el grafo Bruijn no está completo. A partir de un grafo Bruijn(2,16), se ha incrementado el número de nodos físicos desde 1.500 nodos hasta alcanzar el grafo completo, $N_{Bruijn} = 2^{16}$ =65.536 nodos. Como se puede apreciar, cuando existen pocos nodos físicos, el número de enlaces virtuales es relativamente grande y va disminuyendo a medida que el número de nodos físicos aumenta, hasta llegar a 0 enlaces virtuales. No obstante, el número de enlaces físicos, que simboliza el coste real de las comunicaciones para un valor grande de enlaces virtuales, es siempre pequeño. En el caso concreto del ejemplo mostrado se obtiene una media de 3 enlaces físicos/nodo, incluyendo el enlace Hilbert, incluso cuando el número de nodos físicos es muy pequeño. Este factor es muy bueno ya que reduce el coste de las comunicaciones al iniciar la construcción del grafo Bruijn. Este comportamiento es debido a que existen muchos enlaces virtuales, que pertenecen al mismo enlace físico, porque el nodo destino es el mismo, pero simbolizado con otro nodo virtual.
La Fig. 3.11 muestra la media de enlaces físicos por nodo para tres topologías diferentes, Bruijn, Chord y Baton, cuando el sistema es escalado gradualmente. En el caso de Bruijn se obtiene un número más bajo de enlaces que Chord y Baton. Asimismo se puede observar que los enlaces Bruijn se incrementan aumentando el grado k. Cabe destacar que Bruijn está muy optimizado para las búsquedas, ya que con un número menor de enlaces físicos que Chord y Baton, mejora incluso el número de saltos, tal como se mostrará en los capítulos posteriores.

A continuación se calcula analíticamente el número medio de enlaces físicos que puede gestionar un nodo físico, cuando un grafo Bruijn no está completo y considerando que todos los nodos virtuales están uniformemente distribuidos

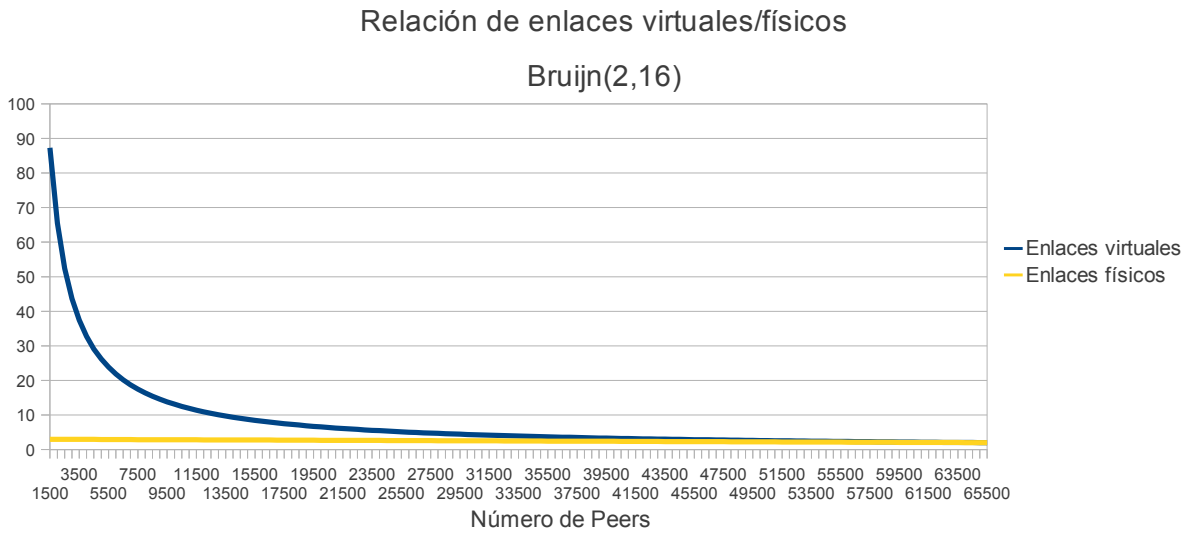

Figura 3.10: Gráfica de la relación de enlaces físicos y virtuales de la red Bruijn.

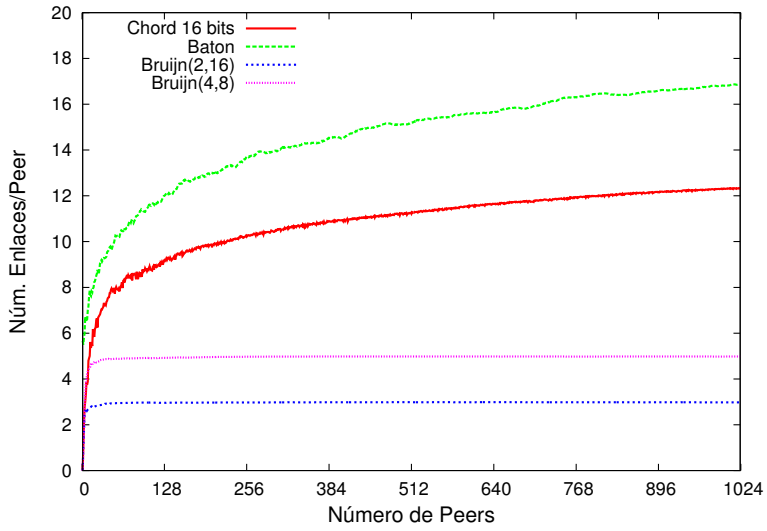

Figura 3.11: Gráfica del número de enlaces físicos para Bruijn, Chord y Baton.

en el grafo. Partiendo de la anterior premisa, el número de claves gestionadas por cada nodo físico Bruijn (N_{Claves}) se muestra en la siguiente ecuación 3.1:

$$N_{Claves} = \frac{N_{Bruijn}}{N_{Fisicos}}, \tag{3.1}$$

siendo N_{Bruijn} el número de nodos totales cuando el grafo está completo y $N_{Fisicos}$ el número de nodos físicos del grafo incompleto. A modo de ejemplo, la Fig. 3.12, muestra las claves gestionadas por dos nodos físicos, H_3 y H_6, sobre un Bruijn(2, d) incompleto. Tal como se puede observar, cada nodo físico gestiona $N_{Claves} = 3$ claves. Para el caso del nodo H_3, éste apunta a las claves H_2, H_3, H_4, H_5, H_6 yH_7, que están gestionadas entre 3 nodos físicos, H_3, H_6 y H_9. En este simple ejemplo se puede observar como el número de enlaces totales, físicos y virtuales, gestionados por cada nodo físico es el siguiente:

$$N_{enlaces\_totales} = k \cdot N_{Claves} \tag{3.2}$$

donde k es el grado del grafo Bruijn.

Por lo tanto, si cada nodo representa N_{Claves}, el número medio de enlaces físicos se obtiene mediante la ecuación 3.3:

$$N_{enlaces\_fisicos} = \frac{N_{enlaces\_totales}}{N_{Claves}} = \frac{k \cdot N_{Claves}}{N_{Claves}} \simeq k, \tag{3.3}$$

donde k es el grado del grafo Bruijn. No obstante, dado que nuestra topología mantiene en cada nodo un enlace Hilbert adicional, el resultado será $N_{enlaces\_fisicos} \simeq k + 1$.

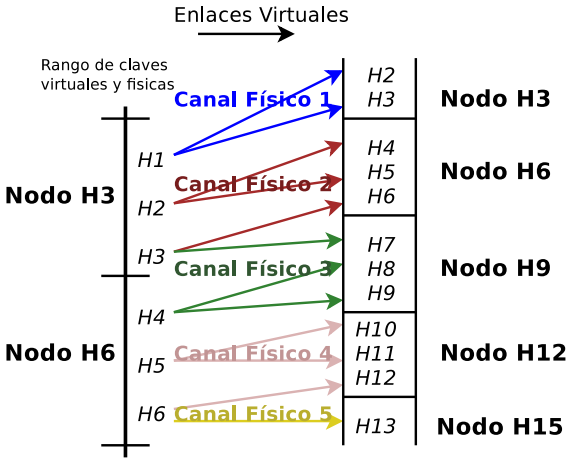

Figura 3.12: Distribución de los enlaces virtuales por nodo en el grafo Bruijn.

Este análisis demuestra que la virtualización del grafo Bruijn no provoca costes de comunicación muy elevados, ya que el coste extra de comunicaciones

de los nodos virtuales se reparte entre todos los nodos físicos.

3.5.3. Tolerancia a fallos en *DisCoP*

Con objeto de analizar la tolerancia a fallos de *DisCoP*, en esta sección se ha analizado la probabilidad de fallos de nuestro sistema cuando se desconectan uno o varios peers a la vez. Según los estudios descritos en [EH85], el número mínimo de Mercados que han de caer simultáneamente para que el grafo Bruijn quede desconectado es $2k-2$, donde k es el grado del grafo. Por lo tanto, la probabilidad de caída del sistema ($P_{caida\_sistema}$) se puede modelizar mediante la ecuación 3.4:

$$P_{caida\_sistema} = (P_{caida\_Mercado})^{2k-2}, \tag{3.4}$$

donde $P_{caida\_Mercado}$ es la probabilidad de desconexión de un Mercado en el nivel Bruijn de *DisCoP*.

Es importante destacar que la desconexión de un Mercado M_H depende del periodo de mantenimiento T_{Tree}. T_{Tree}, usado por el algoritmo de mantenimiento y descrito en el capítulo posterior, es el tiempo que necesita un manager de una área para refrescar la información de estado de sus workers. Esto significa que durante este tiempo, T_{Tree}, los managers de cada área no tienen constancia ni de los cambios de estado en su área, ni de las desconexiones de los peers del subárbol que tiene asociado.
La probabilidad de que un Mercado se desconecte del sistema ($P_{caida\_Mercado}$) es equivalente a la probabilidad de caída del área superior del mercado, controlada por el *root* peer. Para que se produzca la desconexión de un área es necesario que su manager y todos sus managers replicados RM se desconecten simultáneamente del sistema y por lo tanto no exista ningún peer que pueda sustituir al manager caído. De este modo, la probabilidad de caída de un mercado se puede formular de acuerdo con la siguiente ecuación:

$$P_{caida\_mercado} = P_{salida\_Peer}(T_{Tree})^{1+Num\_RM}, \tag{3.5}$$

donde Num_RM es el número máximo de RM por área y $P_{salida\_Peer}(T_{Tree})$ es la probabilidad de que un peer se desconecte del sistema de forma voluntaria o debido a razones técnicas (eléctricas, problemas de red, etc.), durante el

periodo T_{Tree}. Hay que destacar que $P_{salida\_Peer}(T_{Tree})$ depende de la conducta de los usuarios o errores técnicos, que según [DHA03], se modela mediante una función de Poisson. Además, los entornos P2P típicos [GSS06] esperan que los peers estén on-line ($P_{online}(T) = 1 - P_{salida\_Peer}(T)$) durante un periodo de tiempo T, del orden de minutos u horas, y por tanto mucho más grande que el periodo T_{Tree}. La mayoría de autores escogen una probabilidad $P_{online}(T)$ en el rango [0,5, $\cdots$,0,9].
Partiendo de la ecuación 3.5, la probabilidad de caída del sistema (ecuación 3.4), puede reescribirse como:

$$P_{caida\_sistema} = (P_{salida\_Peer}(T_{Tree})^{1+Num\_RM})^{2k-2}. \tag{3.6}$$

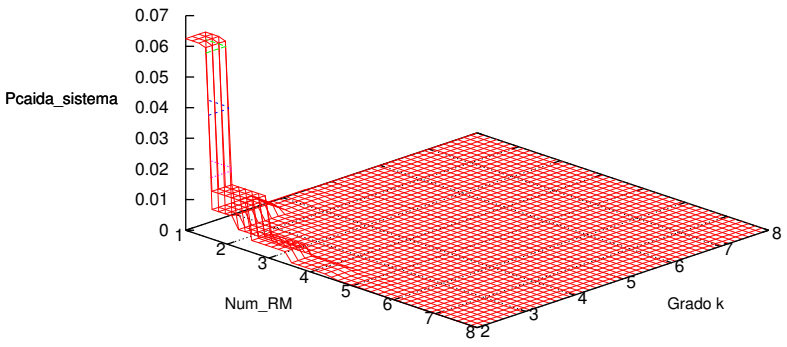

Figura 3.13: Probabilidad de caída del sistema $P_{caida\_sistema}$ en función de RM y k.

La Fig. 3.13 muestra la probabilidad de caída del sistema para una $P_{salida\_Peer}(T) = 0{,}5$, variando el número de RM y el valor de k. Los resultados obtenidos revelan que para valores de Num_RM y k mayores que 2, a medida que se incrementan estos parámetros, la probabilidad desciende rápi-

damente tendiendo a estabilizarse a un valor cercano a cero. Se puede concluir que para obtener una buena fiabilidad o robusteza en el sistema $DisCoP$, los valores óptimos de Num_RM debe de estar comprendido entre el rango 2 y 4. Valores más grandes de Num_RM implicaría una congestión de información de estado en cada área, ya que tener más de 4 RM por área supondría disponer de muchos managers replicados y, en consecuencia, una sobrecarga elevada en el envío de mensajes entre cada manager y sus replicados.

Finalmente, con objeto de medir el número máximo de niveles de un árbol se ha calculado la probabilidad de caída o desconexión de cada nivel del árbol $B-Tree$. La probabilidad de caída de un nivel viene determinada por la probabilidad de caída de una de las áreas del nivel actual, más las probabilidades de caída de las áreas que están en los niveles superiores de la misma rama del árbol. Formalmente, esta probabilidad viene dada por la siguiente ecuación 3.7:

$$P_{caida\_nivel}(h, T_{Tree}) = \binom{h}{1} \cdot P(A, T_{Tree}) - \binom{h}{2} \cdot P(A, T_{Tree})^2 ... (-1)^{(h-1)} \cdot \binom{h}{h} \cdot P(A, T_{Tree})^h = \sum_{i=1}^{h} (-1)^{(i-1)} \cdot \binom{h}{i} \cdot P(A, T_{Tree})^i, \tag{3.7}$$

siendo h el nivel elegido y $P(A, T_{Tree})$ la probabilidad de desconexión de un área que se calcula acorde con la siguiente ecuación 3.8:

$$P(A, T_{Tree}) = P_{salida\_Peer}(T_{Tree})^{1+Num\_RM}, \tag{3.8}$$

Cabe señalar que la fórmula 3.8 se ha deducido a partir de la ecuación 3.5. Para ello se ha considerado que un área A cualquiera se desconecta del árbol cuando su manager y sus workers se desconectan al mismo tiempo del árbol.

Partiendo de la ecuación 3.7, la Fig. 3.14 muestra como evoluciona la probabilidad de desconexión de varios niveles consecutivos de un mercado ($P_{caida\_nivel}(h, T_{Tree})$), empezando desde el primer nivel ($h = 1$) hasta alcan-

zar el décimo nivel ($h = 10$) y por cada uno variando el número de managers replicados Num_RM.

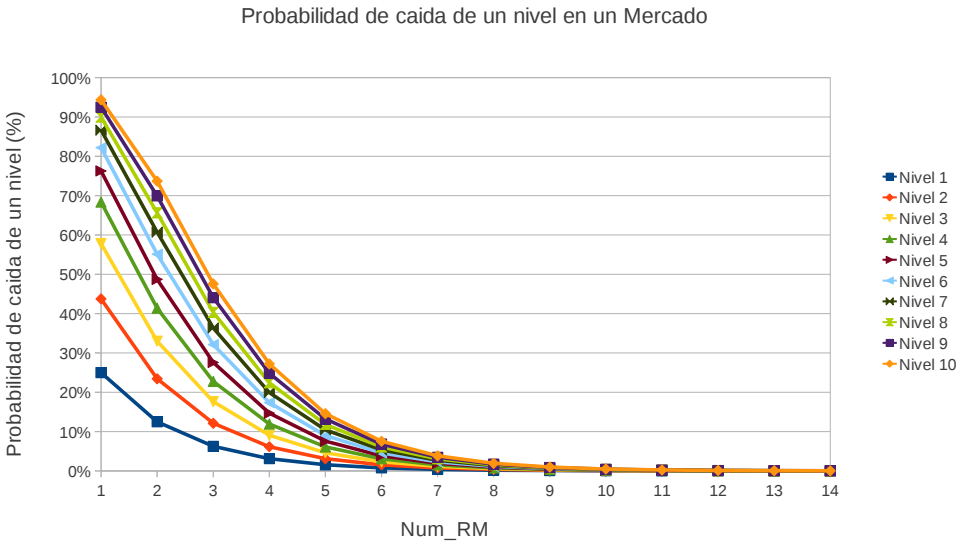

Figura 3.14: Gráfica de la probabilidad de desconexión de un nivel ($P_{caida\_nivel}(h, T_{Tree})$) en un mercado M_i.

Tal como se ha calculado anteriormente, el número de managers replicados (Num_RM) óptimo con respecto a la tolerancia a fallos es 4. En este caso, asumiendo que el valor máximo de probabilidad de desconexión de un nivel $P_{caida\_nivel}(h, T_{Tree})$ de un mercado no puede superar el 20 % para mantener la robustez en la topología $B - Tree$, y acorde con los resultados mostrados en la Fig. 3.14, el número máximo de niveles que debe tener un mercado es igual a 7.

3.6. Localidad

Una de las propiedades más deseadas de un entorno P2P es la *Localidad*. La Localidad es una métrica para medir el grado de proximidad en un *overlay* de los recursos multi-atributo similares. Un sistema con alta localidad involucrará más eficiencia en la búsqueda de recursos computacionales, aproximados o por rangos, a través del *overlay* o red. En esta sección se propone un método para

	$X_{Memoria} \rightarrow V_{Memoria}(GB)$			
$X_{CPU} \rightarrow V_{CPU}(GHz)$	0→(0-2]	1→ (2-4]	2→ (4-6]	3→(6-8]
0→(0-1]	$M_0 = \{0, 0\}$	$M_1 = \{0, 1\}$	$M_2 = \{0, 2\}$	$M_3 = \{0, 3\}$
1→(1-2]	$M_4 = \{1, 0\}$	$M_5 = \{1, 1\}$	$M_6 = \{1, 2\}$	$M_7 = \{1, 3\}$
2→(2-3]	$M_8 = \{2, 0\}$	$M_9 = \{2, 1\}$	$M_{10} = \{2, 2\}$	$M_{11} = \{2, 3\}$
3→(3-4]	$M_{12} = \{3, 0\}$	$M_{13} = \{3, 1\}$	$M_{14} = \{3, 2\}$	$M_{15} = \{3, 3\}$

Tabla 3.2: Ejemplo de asignación de atributos a Mercados (M_i).

medir la *Localidad* de una arquitectura P2P. Este método será utilizado para medir y analizar la localidad de nuestra propuesta, DisCoP, comparándola con la localidad de dos overlays diferentes: un overlay de un solo nivel, como es Hilbert SFC, y un segundo overlay de dos niveles, compuesto por Hilbert SFC más Chord. Además, se analizará la mejora de la localidad que conlleva la aplicación de los enlaces Hilbert (descritos previamente en la sección 3.3.2).

3.6.1. Métricas de Localidad

Partiendo de un overlay $\mathcal{O}$, cada mercado M_i del overlay lo identificamos por unas coordenadas n-dimensionales $(X_1, ..., X_i, ..., X_n)$, donde n es el número de atributos y $X_i \in \mathbb{Z}^+$ es la coordenada del $i - esimo$ atributo en el rango $[0, X_i^{max}]$. En este contexto, hay que destacar que la coordenada X_i tiene asociado un rango de valores del atributo $i-esimo$, V_i^j, donde $0 < V_i^j \leq V_i^{max}$, siendo V_i^{max} el valor máximo que puede adquirir el atributo $i - esimo$. Por ejemplo, para el caso $k = 2$ atributos, que podrían representar por ejemplo CPU y memoria, con un rango de valores de CPU y memoria de $(0, 4GHz]$ y $(0, 8GB]$, respectivamente, entonces V_{CPU}^{max} y $V_{Memoria}^{max}$ serían 4 GHz y 8 GB respectivamente. De acuerdo con estos rangos, la Tabla 3.2 muestra un ejemplo de distribución de mercados, junto con sus coordenadas 2-dimensionales y el rango de valores de CPU y memoria asociados a cada una de las coordenadas.

Teniendo en cuenta la notación descrita anteriormente, a continuación se explica el método para obtener las *métricas de Localidad* a partir de un overlay $\mathcal{O}$:

1. Calcular para cada mercado M_i, su conjunto de *Mercados Similares* SM_i.
   Cada mercado M_i tiene asociado un conjunto de Mercados Similares, de-

Mercado similar *previo* para el $i-esimo$ atributo (X_i)
$\begin{cases}(X_1,..,X_i-1,..,X_n) & if\ X_i>0\\(X_1,..,0,..,X_n) & if\ X_i=0\end{cases}$

Mercado similar *próximo* para el $i-esimo$ atributo (X_i)
$\begin{cases}(X_1,..,X_i+1,..,X_n) & if\ X_i<V_i^{max}\\(X_1,..,V_i^{max},..,X_n) & if\ X_i=V_i^{max}\end{cases}$

Tabla 3.3: Obtención de los mercados similares *previos* y *próximos* para el $i-esimo$ atributo (X_i) del mercado $(X_1,..,X_i,..,X_n)$.

nominado SM_i, de modo que dos mercados son *similares* cuando sus coordenadas difieren exactamente en una unidad en un solo atributo. Por lo tanto, para cada mercado M_i se obtienen dos mercados similares denominados (*previo* y *próximo*) por cada uno de los atributos, exceptuando los caso donde el atributo tiene un valor extremo, V^{max} o 0, que solo tendrán un único mercado similar, previo o próximo respectivamente. Esta casuística está ilustrada en la Tabla 3.3.

2. Obtener para cada mercado M_i, la *Distancia de Mercados Similares* (SMD_i), definida como:

$$SMD_i = \sum_{k=0}^{|SM_i|} \frac{d_{\mathcal{O}}(M_i, M_k)}{|SM_i|}, \tag{3.9}$$

donde $|SM_i|$ es el tamaño del conjunto SM_i y $d_{\mathcal{O}}(M_i, M_k)$ es la distancia, en número de saltos del overlay $\mathcal{O}$, entre el mercado M_i y su mercado similar $M_k \in SM_i$. Hay que destacar que esta métrica depende totalmente del overlay $\mathcal{O}$. De este modo, cuando el overlay está compuesto por dos niveles, como es el caso de *DisCoP*, el cálculo implicará dos pasos diferentes. El primer paso convertirá las coordenadas multi-dimensionales de los mercados M_i y M_k en coordenadas unidimensionales, H_i y H_k, respectivamente, utilizando para el caso de DisCoP, la función de Hilbert. Después, se calculará la distancia entre H_i y H_k, $d_{\mathcal{O}}(H_i, H_k)$, de acuerdo con la topología del segundo overlay. Este proceso está ilustrado en la

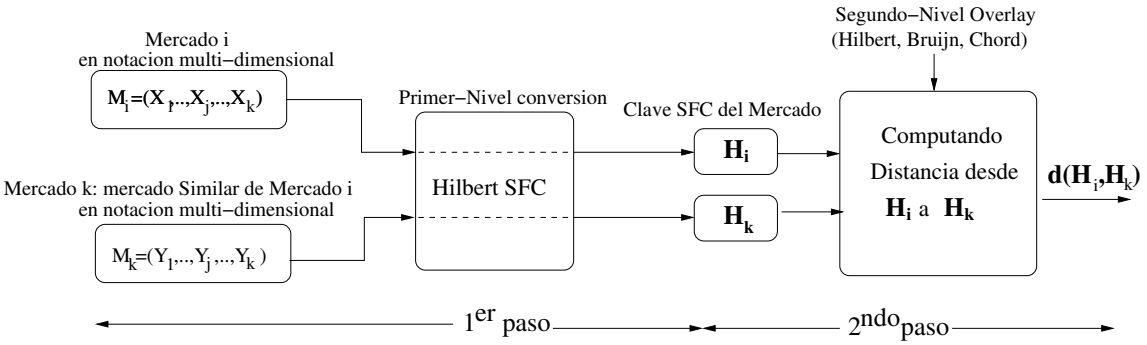

Figura 3.15: Proceso de cálculo de Localidad para un overlay de dos niveles.

Fig. 3.15.

3. Calcular la *Distancia Media entre Mercados* ($MMD_{\mathcal{O}}$) definida como la media de todas las Distancias de Mercados Similares (SMD_i) a partir de un overlay $\mathcal{O}$ concreto. Es una métrica para medir la localidad en función de la distancia absoluta entre mercados similares de un overlay. De este modo, la *Localidad* se incrementa cuando la distancia media del Mercado es más baja. Formalmente:

$$Distancia\ Media\ Mercado_{\mathcal{O}} = \sum_{\mathrm{i}} \frac{SMD_i}{N_{\mathcal{O}}}, \tag{3.10}$$

donde $N_{\mathcal{O}}$ es el número de mercados de un overlay $\mathcal{O}$ y $1 \leq i \leq N_{\mathcal{O}}$.

4. Calcular la *Localidad Relativa* ($RL_{\mathcal{O}}$) de un overlay, definida como:

$$Localidad\ Relativa_{\mathcal{O}} = \frac{MMD_{\mathcal{O}}}{D_{\mathcal{O}}}, \tag{3.11}$$

donde $D_{\mathcal{O}}$ es la máxima distancia entre cualquier par de mercados de un overlay $\mathcal{O}$, distancia conocida también como el diámetro del overlay. $RL_{\mathcal{O}}$ no está normalizada porqué esta métrica varía en el rango $\left[\frac{1}{D_{\mathcal{O}}}, 1\right]$.

5. Calcular la *Localidad Normalizada* ($NL_{\mathcal{O}}$) de un overlay $\mathcal{O}$, definida como:

$$Localidad\,Normalizada_{\mathcal{O}} = 1 - \frac{RL_{\mathcal{O}} - \frac{1}{D_{\mathcal{O}}}}{1 - \frac{1}{D_{\mathcal{O}}}}. \tag{3.12}$$

Cabe destacar que los valores de la métrica $NL_{\mathcal{O}}$ están normalizados en el rango $[0, 1]$. De este modo:

- $NL_{\mathcal{O}} = 0$ significa que el grado de localidad de un overlay $\mathcal{O}$ es nulo. En este caso, todos los elementos estarán a una distancia entre sí igual a $D_{\mathcal{O}}$.
- $NL_{\mathcal{O}} = 1$ significa que el grado de localidad es máximo. Esto significa que la distancia entre un mercado M_i y todos sus mercados similares es igual a 1.

$MMD_{\mathcal{O}}$ mide la localidad en términos absolutos, mientras que $NL_{\mathcal{O}}$ puede servir mejor para comparar la localidad entre diferentes overlays o combinaciones de ellos.

3.6.2. Caso de estudio

En esta sección se muestra, mediante un ejemplo simple, la aplicación del proceso de análisis de la localidad de *DisCoP* en relación con un overlay de un solo nivel, como es la curva SFC de Hilbert, y también otra topología de dos niveles, integradas por Hilbert y Chord. Además se analiza el efecto en la Localidad en *DisCoP* cuando se añaden los enlaces extra de Hilbert. Las Fig. 3.16 y 3.17 muestran un ejemplo de los overlays analizados, con 4 nodos y 2 enlaces por nodo. Hay que destacar que los nodos están representados por la tupla X_1X_2/H_i, donde X_1X_2 es la notación para 2-dimensiones y H_i indica el orden en el espacio Hilbert de una dimensión.

La Tabla 3.4 contiene el conjunto de Mercados Similares SM_i, asociado a cada mercado M_i en la notación 2-dimensional y la Distancia de Mercados Similares (SMD_i) para la topología Hilbert. A partir de los datos de la Tabla 3.4, las métricas de Localidad para el caso de Hilbert: Distancia Media de Mercado ($MMD_{Hilbert}$), Localidad Relativa ($RL_{Hilbert}$) y Localidad Normalizada ($NL_{Hilbert}$), son calculadas y mostradas en la Tabla 3.5. De estos resul-

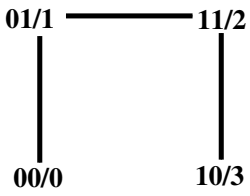

Figura 3.16: Overlay de 1-nivel: topología Hilbert.

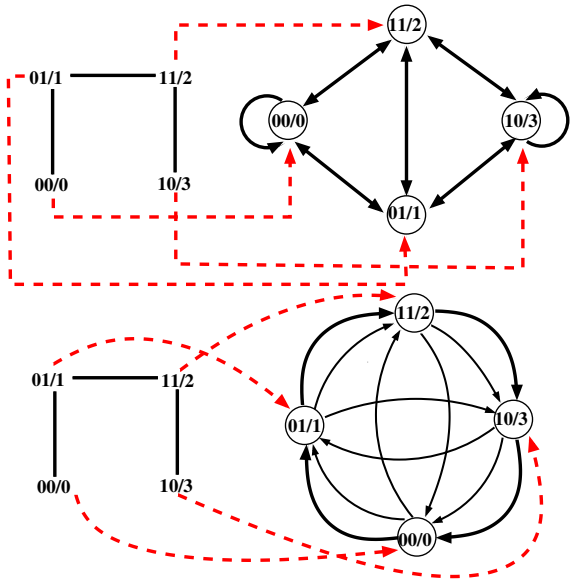

Figura 3.17: Overlay de 2-niveles: (arriba) Hilbert y Bruijn, (abajo) Hilbert y Chord.

tados, se puede concluir que Hilbert tiene una localidad relativamente buena, $NL_{Hilbert} = 0{,}75$, dado que $NL = 1$ es el valor máximo para esta métrica. Hay que decir que este resultado corrobora la elección de *Hilbert SFC* para el primer nivel de la arquitectura *DisCoP*, de acuerdo con su alta localidad respecto de otras curvas.

Después, el mismo procedimiento se repite para el caso de un overlay de dos niveles, como es el caso de la combinación de $Hilbert + Bruijn$ (HB) o $Hilbert + Chord$ (HC). La Tabla 3.6 muestra los valores de las métricas de localidad $MMD_{HB/HC}$, $RL_{HB/HC}$ y $NL_{HB/HC}$ para los overlays HB y HC. En este ejemplo, se puede ver como el overlay HB (NL_{HB} =0.75) presenta una mayor localidad que para HC (NL_{HC} =0.5), hecho que indica una mejor interacción entre Hilbert y Bruijn. De este modo, comparando las métricas

M_i	SM_i	SMD_i
00	10 y 01	$\frac{3+1}{2} = 2$
01	00 y 11	$\frac{1+1}{1} = 1$
10	11 y 00	$\frac{1+3}{2} = 2$
11	01 y 10	$\frac{1+1}{2} = 1$

Tabla 3.4: Distancias de mercados similares para la topología Hilbert (SMD_i).

Métrica	Valor
$MMD_{Hilbert}$	$\frac{6}{4}$ =1.5
$RL_{Hilbert}$	$\frac{1,5}{3}$ =0.5
$NL_{Hilbert}$	$1 - \frac{0,5-\frac{1}{3}}{\left(1-\frac{1}{3}\right)}$ =0.75

Tabla 3.5: $MMD_{Hilbert}$, $RL_{Hilbert}$ y $NL_{Hilbert}$.

HB con las de Hilbert, mostradas en la Tabla 3.5, se puede ver como las dos topologías obtienen la misma métrica NL, aunque el overlay HB obtiene un MMD menor como consecuencia de una localidad absoluta mejor.

Métrica	$Hilbert + Bruijn$ (HB)	$Hilbert + Chord$ (HC)
$MMD_{HB/HC}$	1.25	1.5
$RL_{HB/HC}$	0.625	0.75
$NL_{HB/HC}$	0.75	0.5

Tabla 3.6: $MMD_{HB/HC}$, $RL_{HB/HC}$ y $NL_{HB/HC}$ para los overlays HB y HC de dos niveles.

Partiendo de la localidad alta que caracteriza *Hilbert SFC*, resulta interesante analizar la localidad de los overlays de dos niveles, HB y HC, descritos anteriormente, cuando se añaden los enlaces Hilbert (ver sección 3.3.2). Esto quiere decir que esta topología de dos niveles implementa un camino doble entre cualquier par de mercados del segundo nivel. De este modo, cuando se calculan las métricas de localidad, la distancia en saltos entre dos nodos arbitrarios se escoge, según sea el camino más corto, siguiendo los enlaces Hilbert o bien recorriendo los enlaces propios del overlay. Hay que recordar que la topología HB con enlaces Hilbert corresponde a la arquitectura *DisCoP* completa. La Tabla 3.7 muestra las métricas de localidad *DisCoP* en

Métrica	HB con enlaces Hilbert (DisCoP)	HC con enlaces Hilbert
$MMD_{DisCoP/HC}$	1.25	1.25
$RL_{DisCoP/HC}$	0.625	0.625
$NL_{DisCoP/HC}$	0.75	0.75

Tabla 3.7: Métricas $MMD_{DisCoP/HB}$, $RL_{DisCoP/HB}$ y $NL_{DisCoP/HB}$ para los overlays $DisCoP$ y HC.

relación con las obtenidas con la topología HC con enlaces Hilbert. De la comparación de las Tablas 3.6 y 3.7 se puede ver como la topología $DisCoP$ no mejora su rendimiento con el uso de los enlaces Hilbert, mientras que HC mejora sus prestaciones sustancialmente. Estos resultados revelan la necesidad de analizar con detalle el comportamiento de las diferentes arquitecturas escalando el tamaño de los overlays. Este estudio se ha realizado en la próxima sección 3.6.3.

3.6.3. Resultados de localidad

Esta sección analiza la *localidad* de $DisCoP$, en relación al resto de overlays descritos en la sección anterior, pero escalando el sistema a miles de nodos. El escalado del sistema consiste en incrementar del número de atributos, de 2 hasta 5, y los bits por atributo, entre 1 y 5. Por lo tanto, el número máximo de mercados será N =33.554.432= $(2^5)^5$. El rendimiento de la localidad de cada overlay se obtiene mediante las métricas MMD y NL.

La Fig. 3.18 muestra los resultados obtenidos con las métricas $MMD_{Hilbert}$ (arriba) y $NL_{Hilbert}$ (abajo) para el caso del overlay Hilbert de un solo nivel. Tal como se puede ver en la Fig. 3.18, la métrica $MMD_{Hilbert}$ muestra unas deficiencias muy grandes en términos de localidad. Sin embargo, estos valores extremos son picos aislados que enmascaran la buena conducta de la localidad de Hilbert SFC en términos medios. Este hecho queda perfectamente reflejado en la métrica $NL_{Hilbert}$. La disparidad entre las dos métricas se puede entender si se tiene en cuenta que NL es relativa al diámetro de la topología ($D_{Hilbert}$) y ésta es $D_{Hilbert} = (2^5)^5 - 1$. El comportamiento de ambas métricas revela que la magnitud de la localidad está mejor reflejada en MMD, aunque esta métrica

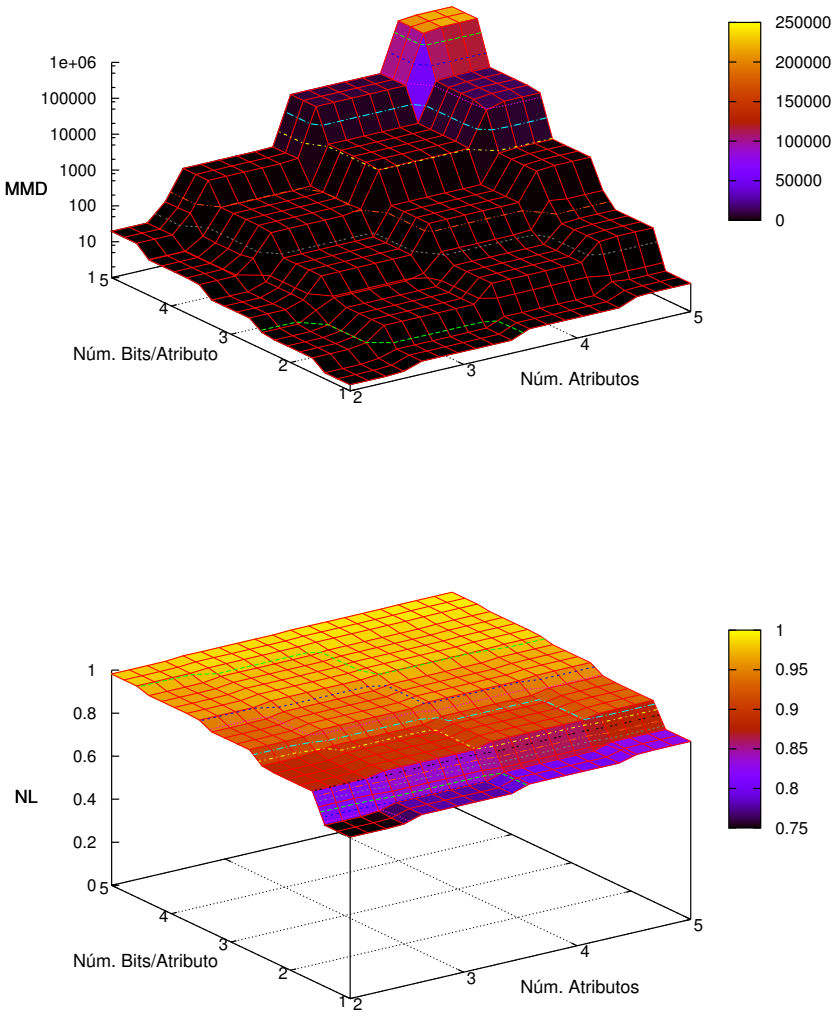

Figura 3.18: Localidad de Hilbert: (arriba) $MMD_{Hilbert}$ y (abajo) $NL_{Hilbert}$.

es demasiado sensible a los casos extremos, hecho que no pasa con la métrica NL. Estos resultados corroboran la necesidad de los dos tipos de métricas.

La Fig. 3.19 (arriba) y (abajo) muestra los resultados MMD para el overlay de dos niveles HB (Hilbert + Bruijn), sin enlaces extra Hilbert (arriba) y el mismo overlay HB pero con enlaces extra, que sería equivalente a la arquitec-

tura *DisCoP* completa (abajo). La comparación entre las dos Fig. muestra que *DisCoP* obtiene, en media, una mejora del 36.7 %, debido al uso de los enlaces extra Hilbert. Por el contrario, la comparación de MMD_{DisCoP}, de la Fig. 3.19 (abajo), con el $MMD_{Hilbert}$ de la Fig. 3.18, muestra que al añadir el grafo Bruijn en el segundo nivel del overlay, se recortan totalmente los picos extremos obtenidos anteriormente con Hilbert, $MMD_{Hilbert} \simeq 250.000$, provocado por la topología unidireccional Hilbert.

La Fig. 3.20 (arriba) y (abajo) muestra los resultados obtenidos para el overlay de dos niveles HC (Hilbert+Chord), sin y con enlaces extra, respectivamente y bajo las mismas condiciones descritas anteriormente. En este caso, la aplicación de los enlaces extra provoca una mejora en el rendimiento de la localidad, con un valor medio del 45.71 %.

Finalmente, la Tabla 3.8 muestra la ganancia media para la métrica NL, entre todos los overlays de dos niveles descritos arriba. De este modo, cada overlay de la fila se compara con el de la columna. En general, se puede ver que la topología Hilbert+Bruijn (HB) tiene siempre mejor localidad que la combinación Hilbert+Chord (HC). Esto corrobora que los mercados con multi-atributos similares están siempre mejor mapeados en Bruijn que en Chord. Además, la ganancia de localidad obtenida en Bruijn es con grado $k = 2$, el cual es mucho menor que el de Chord, que tiene $log_2(N_{HC})$ enlaces por nodo. En este sentido, es importante remarcar que la localidad Bruijn aún mejoraría mucho más añadiendo más enlaces por nodo en Bruijn (grado de Bruijn $k > 2$). Del mismo modo, la aplicación de los enlaces extra en las dos topologías produce una ganancia de localidad notable. Aunque esta mejora se nota ligeramente más en el caso del overlay HC, hecho que significa que los enlaces extra contrarrestan la baja localidad del overlay HC original.

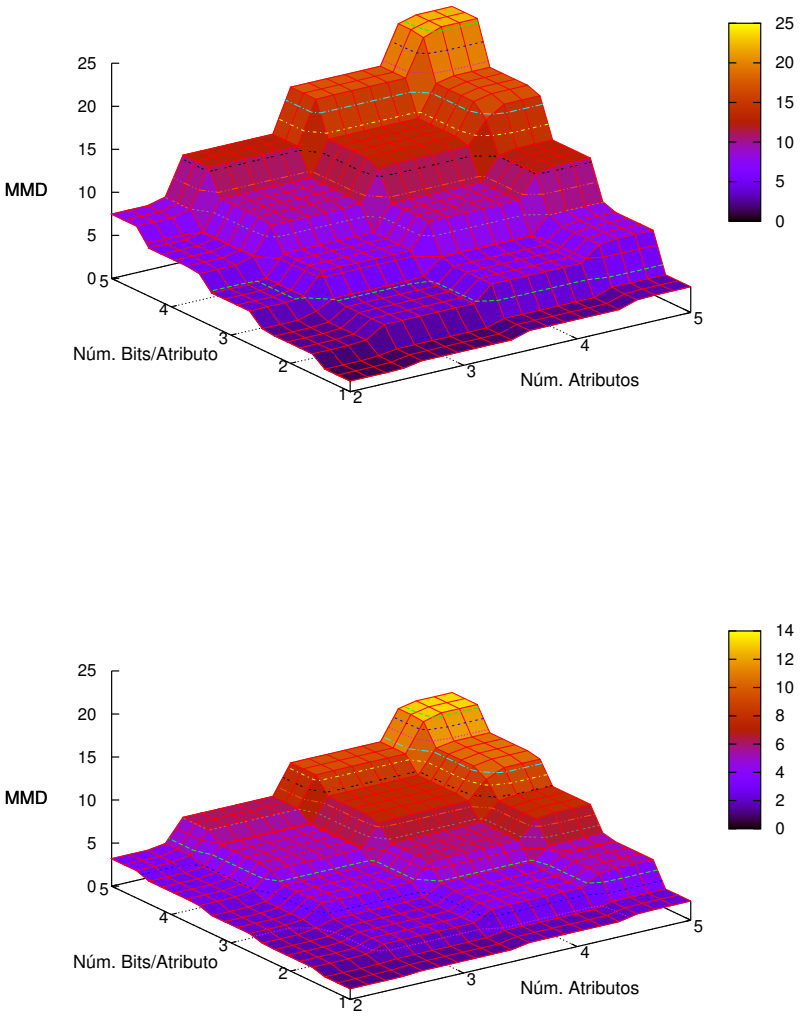

Figura 3.19: $MMD_{HB/DisCoP}$ Localidad: (arriba) topología HB y (abajo) la topología DisCoP (HB con el enlace extra).

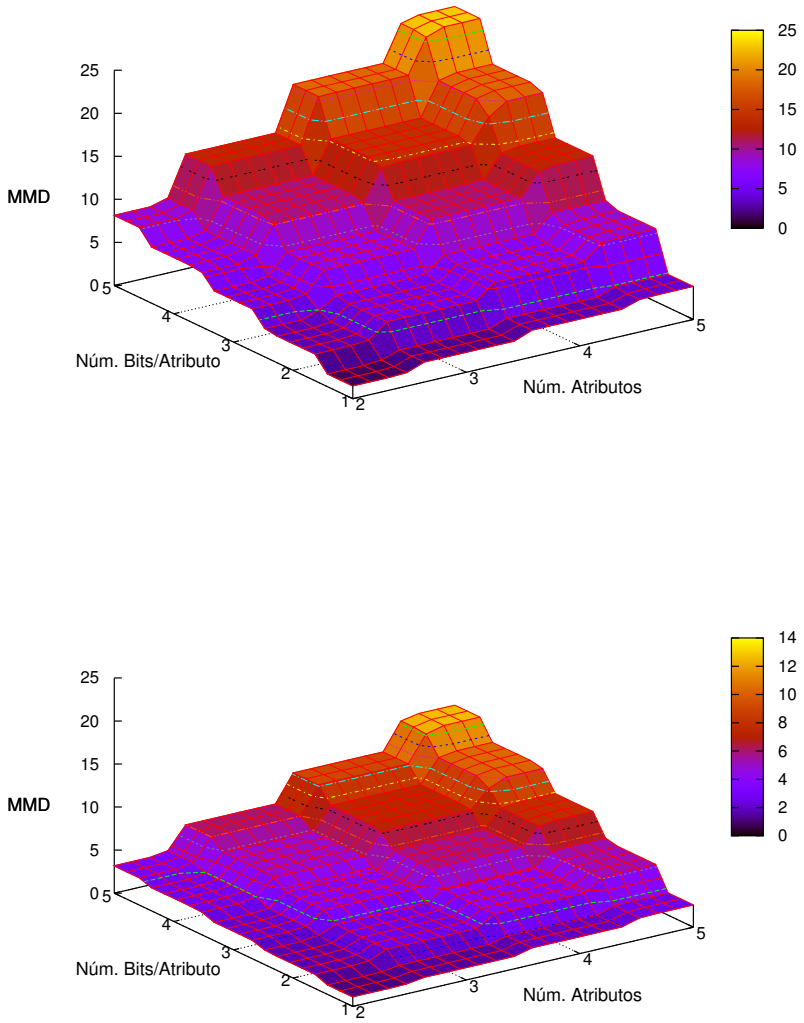

Figura 3.20: MMD_{HC} Localidad: (arriba) topología HC y (abajo) topología HC con enlaces extra.

Fila vs Columna	Ganancia (en %) de la Localidad Normalizada (NL)			
	HB	HB con enlaces extra	HC	HC con enlaces extra
HB		-64,2 %	14,2 %	-62,9 %
HB con enlaces extra	36,7 %		46,3 %	0,98 %
HC	-17,5 %	-91,15 %		-89,0 %
HC con enlaces extra	35,8 %	-1,3 %	45,7 %	

Tabla 3.8: Ganancia de la Localidad Normalizada entre overlays de dos niveles.

Capítulo 4

Funcionalidades del sistema DisCoP

Una vez descrito el diseño de la arquitectura de DisCoP, en este capítulo se procede a explicar las funcionalidades y servicios principales del sistema, adaptados y optimizados a la arquitectura diseñada previamente.
Los principales mecanismos del sistema son el algoritmo de inserción de peers, el algoritmo de mantenimiento de la conectividad del sistema, el algoritmo de salida o desconexión de los peers y finalmente el algoritmo de búsqueda de recursos. A medida que se van presentado los algoritmos, también se evalúa su rendimiento, obtenido de un extenso análisis de los resultados experimentales.

4.1. Mecanismo de Inserción de Peers

En este apartado se explica el mecanismo empleado para insertar un nuevo peer en el sistema DisCoP. El mecanismo de inserción está compuesto de dos partes: la primera parte corresponde a la inserción del peer en el nivel Bruijn, comentado en la sección 4.1.1 y la segunda, comentado en la sección 4.1.2, corresponde a la inserción en el nivel de Mercados (B-Tree).

4.1.1. Inserción en Bruijn

Una vez que un peer $Peer_h$ obtiene la clave identificadora H_i asociada con sus recursos computacionales mediante la función de Hilbert, éste ya puede ser insertado en la red Bruijn. El Alg. 1 sirve para insertar un nodo en el nivel Bruijn. Para poder insertarlo es necesario que previamente se conozca al menos un nodo de la red, al cual llamaremos $Peer_j$. A continuación, $Peer_h$ realiza una operación de *bootstrap* que consiste en copiar la tabla de routing del nodo $Peer_j$. Una vez copiada, el nodo $Peer_h$ utiliza la tabla para localizar el mercado que representa su clave H_i, $Mercado_H$. Si existe este mercado, entonces pasa a la fase inserción de $Peer_h$ dentro del tercer nivel, en la topología de árbol $B-Tree$. Para ello se aplica el algoritmo de inserción de peers dentro de los mercados descrito en la sección 4.1.2.
Si no existe el mercado buscado, entonces realiza peticiones de búsqueda de mercados vecinos. Primero busca los que se encuentran en su lado derecho, representado por el conjunto $\{\forall i \; \epsilon \; 0 \leq i < k \mid Mercado_{\{H<<1+i\}}\}$, donde k es el grado de Bruijn. Con los nodos encontrados actualiza su tabla de routing con sus respectivas direcciones físicas de red. Después, $Peer_h$ procede a encontrar los nodos vecinos del lado izquierdo, es decir aquellos que tienen el $Mercado_H$ virtual en sus tablas de *routing*. Estos están representados por el conjunto $\{\forall i \; \epsilon \; 0 \leq i < k \mid Mercado_{\{l+(H>>1)\}}\}$.
Finalmente $Peer_h$ notifica su nuevo estado de insertado a su sucesor Hilbert, $Mercado_{H+1}$, indicándole que es su predecesor y también que elimine el nodo virtual $Mercado_H$, el cual simbolizaba su inexistencia. De este modo, $Peer_h$ constituye el nodo root del nuevo $Mercado_H$.

El algoritmo de inserción en el nivel de Bruijn puede tener según el caso dos costes computacionales diferentes: en el caso de que exista el mercado que representa la clave H_i, la inserción solo requiere una búsqueda, y su coste es $\theta(\log_k(N_{Bruijn}))$, donde k es el grado y N_{Bruijn} la dimensión de la red. En el segundo caso, se requieren k operaciones de búsqueda de los vecinos derechos, k para los vecinos izquierdos y una operación para crear el enlace Hilbert sucesor. De este modo, el coste final es $\theta((2 \cdot k + 1) \log_k(N_{Bruijn}))$.

Algoritmo 1 Algoritmo de inserción en la red Bruijn

procedure $Peer_h$.Insercion_Mercado_en_Bruijn()

Require: $Peer_j$: Nodo *root* cualquiera de la red Bruijn

$Peer_h$ conecta a $Peer_j$
{Operación Bootstrap}
$Peer_h$ copia **tabla de routing** de $Peer_j$ temporalmente
$Peer_h$ busca el $Mercado_H$
{Examina si existe su mercado}
if ($\exists$ $Mercado_H$) **then**
  $Peer_h$ se inserta en $Mercado_H$
else
  for $i := 0$ hasta $i < k$ paso 1 **do**
    $Peer_h$ busca $Mercado_{\{H<<1+i\}}$
    $Peer_h \longrightarrow Mercado_{\{H<<1+i\}}$
  end for
  for $i := 0$ hasta $i < k$ paso 1 **do**
    $Peer_h$ busca a $Mercado_{\{i+(H>>1)\}}$
    $Mercado_{\{i+(H>>1)\}} \longrightarrow Peer_h$
  end for
  $Peer_h \longrightarrow_{Hilbert} Mercado_{H+1}$
  $Mercado_{H+1}$ elimina nodo virtual $Mercado_H$
  $Peer_h$ pasa a ser $Mercado_H$
end if

end procedure

4.1.2. Inserción en los Mercados

Si existe el $Mercado_H$, el nuevo nodo se inserta en el árbol $B-Tree$ del $Mercado_H$. La inserción en el árbol se realiza de forma balanceada, minimizando de esta forma su altura. Este hecho redunda en los pasos de inserción en posteriores entradas y salidas de peers, de forma que la inserción será más eficiente y por lo tanto menos costosa. La inserción de $Peer_h$ se realiza siguiendo los pasos del siguiente algoritmo:

1. En el primer paso se comprueba que el área A_i de el manager M_i no esté llena. En caso afirmativo, se inserta el nuevo peer en dicha área, pasando a ser un *worker*.

2. El segundo paso sucede cuando el área A_i está llena y existe algún worker W_j, dentro de esta área, que no es manager de un área inferior. En este caso, dicho worker W_j se convierte en un manager M_j de una nueva área A_j y el $Peer_h$ se inserta en el área A_j.

3. En el tercer paso, si el área A_i está llena y existen uno o más managers con workers en un área inferior A_j, entonces se comprueba el área inferior A_j que no esté llena y con menos peers. En caso afirmativo, $Peer_h$ se inserta en el área encontrada A_j. Este caso asegura que tenga en cuenta las áreas no llenas y se aprovechen para insertar un nuevo peer antes de crear una nueva.

4. El cuarto paso sucede cuando todas las áreas del nivel inferior al área A_i están llenas. En este caso, el *manager* M_i inserta $Peer_h$ en la rama menos poblada. De un modo recursivo, $Peer_h$ se inserta en la rama encontrada siguiendo el primer paso descrito en este algoritmo. De este modo, esta heurística asegura el balanceo del árbol.

5. Finalmente, en el caso de que el mercado solicitado esté completamente lleno (ver sección 3.5.3), $Peer_h$ será insertado en una cola de espera.

El coste total de inserción de un peer en *DisCoP* está compuesto de dos partes. La primera parte corresponde al coste de búsqueda del mercado en el

grafo Bruijn y la segunda consiste en el coste de inserción del nuevo peer en el mercado (árbol B-Tree). Tal como se ha comentado en la sección 4.1.1, el coste de búsqueda de un mercado es $\theta(log_k(N_{Bruijn}))$. El segundo coste viene determinado por el número de consultas realizadas a los managers de niveles inferiores siguiendo la jerarquía de árbol desde el nodo *root* M_i a un área no completada o recién creada. El coste es $\theta(log_{|Area|}(N_{Tree} \cdot (|Area| - 1) + 1)$, donde $|Area|$ es la capacidad del área y N_{Tree} es el número de peers del mercado. Por lo tanto, el coste total corresponde a $\theta(log_k(N_{Bruijn}) + \log_{|Area|}(N_{Tree} \cdot (|Area| - 1) + 1))$.

4.1.2.1. Análisis experimental

Para analizar y verificar el rendimiento de los algoritmos de la arquitectura DisCoP, se ha utilizado el mismo entorno de simulación descrito en el apartado 3.5.1.

Esta sección muestra el coste temporal de las principales funcionalidades de DisCoP presentadas en este capítulo. Teniendo en cuenta que el dinamismo de nuestro sistema recae principalmente en el nivel de Mercados, únicamente se mostrarán los resultados relacionados de este nivel. Todas las pruebas se han realizado para un Bruijn(2,10) completo. Del mismo modo, se ha limitado el tamaño máximo de un mercado a 1.024 peers.
En primer lugar, se ha medido el coste de inserción suponiendo que el grafo Bruijn está completo. La Fig. 4.1 muestra el tiempo de inserción de un peer en función del índice de entrada al mercado y el tamaño de las áreas. Estos resultados han sido obtenidos bajo una latencia de red de 100ms y un ancho de banda de 1 Mbps. En la misma Fig. 4.1 se puede ver que para áreas pequeñas, el tiempo de inserción de cada peer es bastante alto. Esto es debido a que la jerarquía del árbol crece más rápido en este caso, y como consecuencia, se tienen que crear constantemente nuevas áreas. El salto en diagonal que aparece en la gráfica corresponde a la latencia extra ocasionada por el nuevo peer en el momento en que ha encontrado todas las áreas llenas y el manager tiene que crear otra para insertarlo.

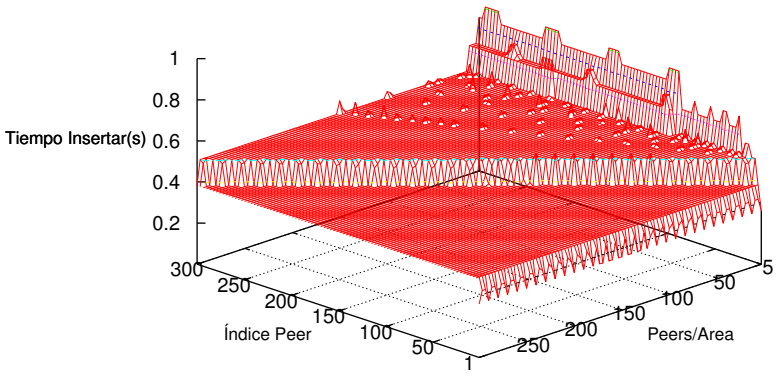

Figura 4.1: Coste del tiempo de inserción de un peer en un Mercado.

4.2. Mantenimiento de la conectividad de los Peers

El objetivo del algoritmo de mantenimiento es tener actualizados, en cada peer, los enlaces a sus vecinos e informar sobre el estado de los recursos computacionales compartidos. El algoritmo de mantenimiento también se realiza en los dos niveles del sistema, en el nivel Bruijn, explicado en la sección 4.2.1 y en el de Mercados, explicado en la sección 4.2.2.

4.2.1. Mantenimiento en la red Bruijn

El mantenimiento en el nivel Bruijn requiere que cada nodo (*root*), el cual representa un mercado del grafo $Bruijn(k, d)$, envíe periódicamente, cada T_{Bruijn} segundos, un mensaje de control $Market\_Request$ a todos sus mercados vecinos. Los vecinos de $Mercado_H$ son el conjunto $\{Mercado_{\{H<<1+i\}} \mid \forall i \, \epsilon \, 0 \leq i < k\}$ más el mercado sucesor $Mercado_{H+1}$ siguiendo el enlace extra Hilbert. Una vez recibido el mensaje, los mercados vecinos responden enviando otro

mensaje con el texto *Market_ Alive*, notificando de esta forma que aún existe. Cuando un mercado no recibe el mensaje de control en T_{Bruijn} segundos y éste a su vez es su mercado predecesor $Mercado_{H-1}$, entonces se ejecuta el procedimiento de *Salida de Mercados de Bruijn*, explicado en la sección 4.3.1. El Alg. 2 (Algoritmo de Mantenimiento Bruijn) describe formalmente el procedimiento de mantenimiento del nivel Bruijn.

Algoritmo 2 Algoritmo de mantenimiento Bruijn

```
procedure Mantenimiento_Nivel_Bruijn()
  for (t := 0 ; t < ∞ ; t++) do
    for all Mercado_H conectado a Bruijn(k, d) do
      for all Mercado_J|Mercado_J es vecino de Mercado_H do
        Mercado_H envía mensaje Market_Request a Mercado_J;
        Mercado_H recibe Market_Alive desde Mercado_J;
      end for
      if (∃Mercado_J no ha respondido ∧ Mercado_J = Mercado_{H-1}) then
        Mercado_H.Gestionar_Salida_de_Mercados_de_Bruijn(Mercado_J);
      end if
      wait(T_Bruijn);
    end for
  end for
end procedure
```

4.2.2. Mantenimiento en un Mercado

Con el fin de preservar la topología $B - Tree$ de cada mercado, conservando su conectividad, cada manager M_i ejecuta en cada periodo T_{Tree}, el Alg. 3 (Algoritmo de Mantenimiento de un Mercado). El principal objetivo del algoritmo es actualizar la información del estado de todos los nodos del mercado en los niveles superiores a partir del área gobernada por M_i hasta el nodo *root* del árbol. Se trata de obtener el estado de los recursos computacionales gestionados por cada manager. El intercambio de mensajes se realiza entre cada M_i y sus workers W_i de la misma área A_i en cada periodo T_{Tree}. M_i envía diferentes tipos de mensaje dependiendo del rol del peer receptor: *worker* o *manager replicado*. Además, una vez recibido el mensaje de M_i, los workers responden enviando información sobre la capacidad actual de sus re-

cursos computacionales, en caso que el peer sea un simple *worker*, o el estado de los recursos computacionales disponibles de todo el subárbol inferior, si el peer es también manager de una área inferior. La información de los recursos computacionales que se envía en el algoritmo puede ser el número de máquinas que existen con el estado de ejecución libre o ocupado, cuales tienen un 20 %, 40 %, 60 %, 80 % y 100 % de los recursos libres respecto su potencia máxima, etc.

De este modo, en cada periodo T_{Tree}, se va incrementando el nivel de actualización de la información y por consiguiente, el coste de actualizar la información desde un nivel inferior hacia el nodo *root* es de $\theta((h-1)\cdot T_{Tree})$, donde h es el número actual de niveles del árbol. Hay que destacar que $h = \log_{|Area|}(N_{Tree}\cdot(|Area|-1)+1)-1$. En la próxima sección, 4.2.2.1 (Análisis experimental), se explica como obtener los valores óptimos del periodo de mantenimiento T_{Tree} y el tamaño del área ($|Area|$).

Algoritmo 3 Algoritmo de mantenimiento de un Mercado

```
procedure Mantenimiento_Nivel_Mercado()
  for (t := 0 ; t < ∞ ; t++) do
    for all W_i conectado a la área A_i do
      if (W_i = RM_i) then
        M_i envía mensaje Manager_Alive + Información_Replicada a W_i;
      else
        M_i envía mensaje Manager_Alive a W_i;
      end if
      M_i recibe Peer_Alive + Recursos_Disponibles desde W_i;
    end for
    wait(T_Tree);
  end for
end procedure
```

4.2.2.1. Análisis experimental

Esta sección analiza el tamaño óptimo de un área en función de los parámetros que afectan en la conectividad de los peers: el periodo T_{Tree} del algoritmo de mantenimiento, la latencia y el ancho de banda de la red. Tal y como se describe en el algoritmo de mantenimiento de los mercados en la sección 4.2.2, el

manager de cada área envía periódicamente mensajes a sus respectivos workers y viceversa. Debido a ello, el tamaño de un área depende directamente del tiempo necesario en enviar y recibir estos mensajes de mantenimiento. Si el tiempo en enviar y recibir es más grande que el periodo de mantenimiento T_{Tree}, es señal de que los managers no puede atender a tantos workers durante el periodo marcado y por lo tanto se ha alcanzado el límite máximo de peers que pueda contener un área.

Teniendo en cuenta estas condiciones, el tamaño del área depende directamente de la longitud del mensaje, el ancho de banda, la latencia de red y el periodo T_{Tree}. En la Fig. 4.2 se muestra el número máximo de peers insertados en un área variando las condiciones de la red, la latencia y el ancho de banda. En cambio, en la Fig. 4.3 se muestra el mismo resultado pero variando la latencia de red y el periodo de mantenimiento T_{Tree}.

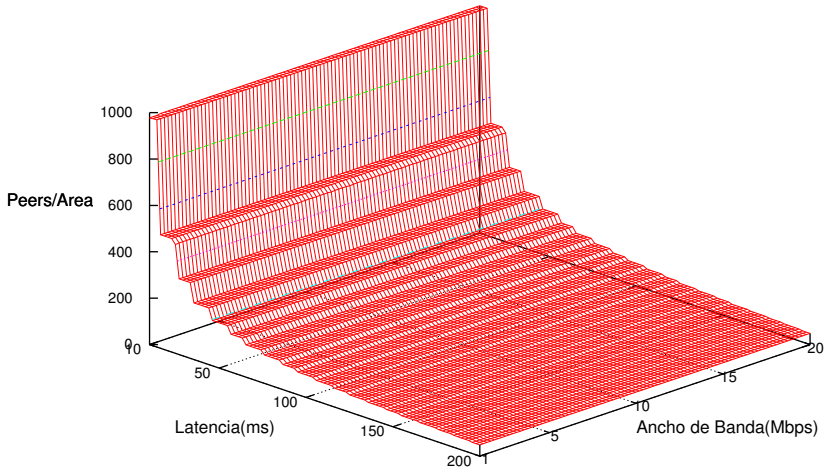

Figura 4.2: Tamaño de un área en función de la latencia y ancho de banda de la red.

La Fig. 4.2 muestra como la latencia en el límite de un área es un factor más influyente porque generalmente, los mensajes enviados por el algoritmo de mantenimiento son cortos y, como consecuencia de ello, el tiempo de recepción

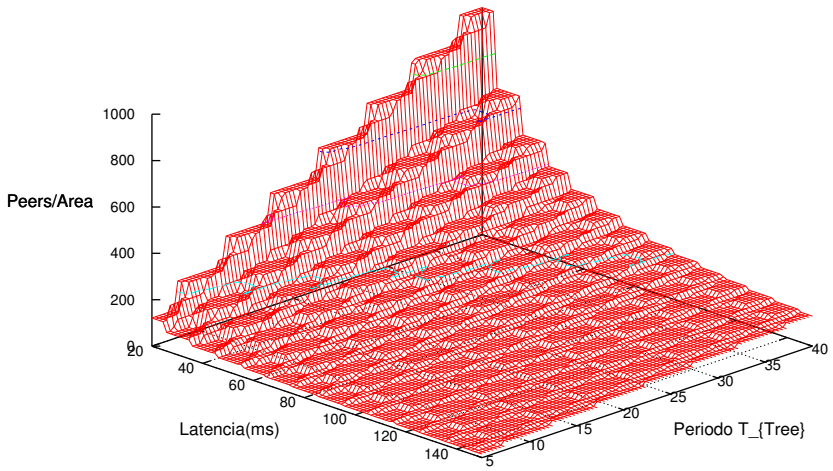

Figura 4.3: Tamaño de un área en función de la latencia y el periodo de mantenimiento T_{Tree}

de un mensaje es mucho más prolongado que su coste de transmisión por la red.

Del mismo modo, la Fig. 4.3 ilustra como el tamaño máximo de un área está directamente relacionado con el periodo de mantenimiento T_{Tree}. Además, la mala elección del periodo T_{Tree} puede tener consecuencias muy negativas. Si su valor es elevado, el sistema puede estar altamente desactualizado. En cambio, si es bajo, el coste de mantener todo el sistema actualizado puede ser demasiado excesivo. Además, si el periodo T_{Tree} es demasiado bajo, el tamaño del área tiene que ser a su vez muy reducido, por tanto, el número de niveles de la jerarquía del árbol $B - Tree$ deberá aumentar de forma inversamente proporcional.

Acorde con los resultados anteriores, el tamaño del área depende de la media de la latencia de conexión a Internet, que en nuestro entorno experimental es $\approx 100ms$, del coste de optimizar las funcionalidades de los mercados de DisCoP, del tamaño de los mensajes de mantenimiento y además hay que permitir un margen de error para evitar la saturación de la red. Basándonos en los re-

sultados de la Fig. 4.3 se puede comprobar como el valor óptimo del tamaño de una área está comprendido entre 20 y 30 peers y un periodo de mantenimiento T_{Tree} entorno a 20 segundos.

Acorde con estos resultados y teniendo en cuenta los resultados experimentales obtenidos en la sección 3.5.3, con respecto al número máximo de niveles que debe tener un mercado de tipo árbol $B-Tree$, los parámetros extremos de un mercado son un tamaño de área de 20 peers y 7 niveles. Partiendo de estos valores, el número total de peers que puede albergar un mercado es $N_{Tree} = \frac{|Area|^{(h+1)}-1}{|Area|-1} = \frac{20^8-1}{19}$ =1.347.368.421 nodos.

4.3. Mecanismo de salida de Peers

Esta sección explica como se controla la gestión de la salida de peers de *DisCoP* cuando éstos, voluntaria o involuntariamente se desconectan del sistema.
El mecanismo está implementado teniendo en cuenta la topología de dos niveles de DisCoP.

4.3.1. Salida de Mercados en Bruijn

Cuando un mercado falla, alguno de sus mercados vecinos tienen que reparar las conexiones perdidas y además crear un mercado o nodo virtual que represente su ausencia. Aunque la probabilidad de que un mercado falle es extremadamente baja, por razones de fiabilidad de la topología $B-Tree$ (ver sección 3.5.3), es necesario un mecanismo para mantener la conectividad en el grafo Bruijn. El método que presentamos está implementado en el Alg. 4 (Algoritmo de salida de Mercados en Bruijn). El proceso es el inverso al algoritmo de inserción de mercados (Alg. 4).

El mecanismo es el siguiente: cuando un $Mercado_H$ falla, su sucesor $Mercado_{H+1}$ detecta su fallida mediante el algoritmo de mantenimiento. A continuación, éste virtualiza el $Mercado_H$ y restablece todos los enlaces rotos provocados en la salida de $Mercado_H$, tal y como se muestra en el Alg. 4.

Siguiendo el Alg. 4, el $Mercado_{H+1}$ notifica a su mercado contiguo

$Mercado_{H-1}$ que tiene que actualizar el enlace Hilbert apuntando hacia él. Después, el $Mercado_{H+1}$ busca y restablece los k enlaces Bruijn izquierdos del $Mercado_H$ hacia los mercados $Mercado_{\{i+(H>>1)\}}$, donde i es el índice del enlace y el símbolo $>>$ representa la operación de desplazamiento de bits hacia la derecha. Finalmente, también se restablecen los k enlaces Bruijn derechos ($Mercado_{\{H<<1+i\}}$) entre el mercado $Mercado_H$ y $Mercado_{H+1}$. En este caso, el símbolo $<<$ representa la operación de desplazamiento de bits hacia la izquierda.

El coste total de este algoritmo es $\theta((2{\cdot}k{+}1){\cdot}$ *Coste de Búsqueda*). Teniendo en cuenta que el *Coste de Búsqueda* en el grafo Bruijn es $\log_k(N_{Bruijn})$, el coste total del Alg. 4 es $\theta((2 \cdot k + 1) \cdot \log_k(N_{Bruijn}))$.

Cabe señalar que este proceso debe repetirse por cada mercado virtualizado que estuviera bajo la responsabilidad del mercado fallido.

Algoritmo 4 Algoritmo de salida de Mercados en Bruijn

procedure Gestionar_Salida_de_Mercados_en_Bruijn()
Require: $Mercado_H$ falla y $Mercado_{H+1}$ detecta la fallida
  $Mercado_{H+1}$ reemplaza $Mercado_H$
  {Actualiza el enlace extra}
  $Mercado_{H+1}$ notifica a $Mercado_{H-1}$ que el es su sucesor
  $Mercado_{H-1} \longrightarrow_{Hilbert} Mercado_{H+1}$
  {Actualiza los enlaces Bruijn derechos}
  for $i := 0$ a $i < k$ paso 1 **do**
    $Mercado_{H+1} \longrightarrow Mercado_{\{H<<1+i\}}$
  end for
  {Actualiza los enlaces Bruijn izquierdos}
  for $i := 0$ a $i < k$ paso 1 **do**
    $Mercado_{\{i+(H>>1)\}} \longrightarrow Mercado_{H+1}$
  end for
end procedure

4.3.2. Salida de Peers de los Mercados

Cuando un peer P_i se desconecta de un mercado, voluntaria o involuntariamente, el manager del peer P_i detecta que se ha producido una desconexión del peer en cuestión a partir del algoritmo de mantenimiento del sistema, ex-

plicado en la sección 4.2.2. Cuando esto pasa, el manager comprueba si el peer P_i es un nodo *worker* (W_i) o un *manager* (M_i) de una área inferior A_i:

- En caso de ser un *manager* M_i de un área A_i, éste avisa a su manager replicado más antiguo (RM_i) de su área A_i que será el nuevo manager M_i. Además RM_i avisa a todos los workers de A_i que sera el nuevo manager de su área. Posteriormente, si RM_i es manager M_j de una área inferior A_j ejecutará de nuevo el Alg. 5.
- En caso de ser un *worker*, W_i, no es necesario realizar ningún tipo de reestructuración del árbol.

Por motivos de fiabilidad, el algoritmo selecciona el manager replicado más antiguo, es decir el que lleva más tiempo conectado en su área. Por extensión, los nodos situados en niveles superiores del árbol son más antiguos. De este modo, el sistema entiende que los peers que hace más tiempo que están conectados en el sistema son más fiables, y por lo tanto deben estar en niveles superiores del árbol. Los peers más jóvenes en cambio se consideran más dinámicos y por este motivo deben estar situados en niveles más bajos. De esta forma se controla el gran dinamismo y mutabilidad de los nodos de un sistema P2P, garantizando de esta forma un mayor grado de robustez de la plataforma.

Acorde con esto, el coste del Alg. 5 viene determinado por el número de peers afectados por la reestructuración de la topología en la desconexión de un peer del sistema. El peor caso sucede cuando un manager replicado RM_i seleccionado para reemplazar la salida de un manager es también un manager de un nivel inferior. El coste del algoritmo es $\theta(\log_{|Area|}(N_{Tree} \cdot (|Area| - 1) + 1) - 1)$.

4.3.2.1. Análisis experimental

En este apartado se explica la experimentación realizada para evaluar el mecanismo de salida de peers, bajo las mismas condiciones experimentales descritas en la sección 3.5.1. La Fig. 4.4 muestra el tiempo empleado en reestructurar y balancear el mercado cuando un peer con responsabilidad, como es el caso de un manager, falla. La gráfica muestra, una vez creado todo el sistema,

Algoritmo 5 Algoritmo de gestión de la salida de Peers en el Mercado

procedure M_i.Gestionar_Salida_Peers_en_Mercado()
Require: RM_i,A_i
  M_i selecciona el $RM_i \in A_i$ más antiguo
  if $|A_i - RM_i| \neq 0$ **then**
    RM_i pasa a ser manager M_i de A_i
    RM_i notifica $\forall workers \in A_i$ que será el manager M_i
    if RM_i es también manager M_j de un área inferior A_j **then**
      M_j.Gestionar_Salida_Peers_en_Mercado()
    end if
  else
    RM_i pasa a ser *worker* de un área A_i superior
  end if
end procedure

el tiempo empleado en la reestructuración de un árbol del tipo $B-Tree$ en función del índice del peer fallido. El índice en este caso corresponde al orden en que ha sido insertado en el mercado el peer que ha fallado. Se puede ver como el mayor tiempo de reestructuración se corresponde con un manager situado en los niveles superiores del árbol. Por otro lado, cuando un worker falla, el tiempo de reestructuración es despreciable. En la gráfica se puede observar como las áreas pequeñas causan más picos. Esto es debido a que en proporción, contienen muchos más managers.

4.4. Mecanismos de búsqueda de recursos

En esta sección se explican los principales mecanismos de búsqueda de recursos de cómputo de *DisCoP*. Los mecanismos de búsqueda se emplean para localizar recursos con unas características o potencias computacionales concretas. Para ello es necesario localizar la dirección física de red de los mercados en donde se encuentran estos recursos. Hay que destacar que la dirección de red de los mercados es la misma que la dirección del nodo *root* de la topología $B-Tree$.

En la sección 4.4.1 se describen distintos mecanismos de búsqueda de un nodo destino (Mercado) a partir de cualquier otro nodo, basado en el desplazamien-

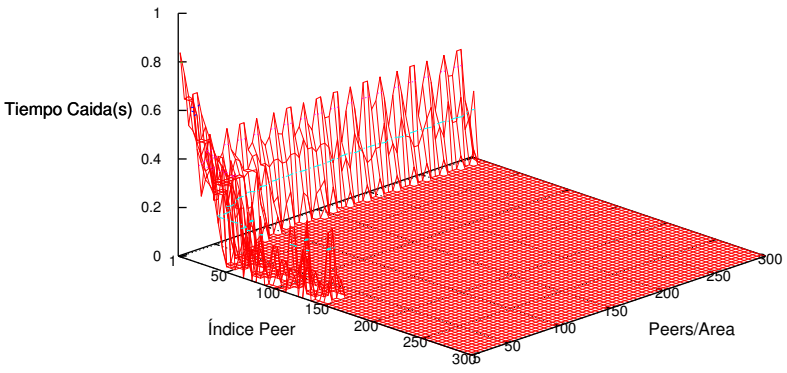

Figura 4.4: Coste de salida de un peer en un Mercado.

to de dígitos sobre la clave Bruijn.

A continuación, en la sección 4.4.2 se describen los algoritmos de búsqueda exacta, empleados por *DisCoP*, para localizar un único Mercado con unos valores de atributo concretos.

El principal mecanismo de búsqueda implementada en *DisCoP* soporta búsquedas complejas del tipo multi-atributo y en rango. Tal como se describió en el capítulo 1, las búsquedas multi-atributo son capaces de localizar datos o recursos especificando en la consulta más de una atributo. Además, si a estas consultas se les añade un rango con dos límites por atributo (búsqueda en rango), entonces la búsqueda es aún más compleja. En la sección 4.4.3 se describe el principal algoritmo de búsqueda compleja de *DisCoP*.

Finalmente, en la sección 4.4.4, se propone un segundo algoritmo que soporta búsquedas aproximadas. Este algoritmo puede acoplarse al algoritmo principal de búsquedas según las necesidades. Cabe recordar que las búsquedas aproximadas sirven para localizar, con pocos saltos, recursos con valores similares al recurso solicitado, cuando éste no existe.

4.4.1. Conceptos previos de búsqueda en Bruijn

En la explicación de la arquitectura principal de DisCoP en el capítulo 3, se describió que el nivel donde se producen el intercambio principal de mensajes en la búsqueda de recursos, planificación, etc. es en la red Bruijn. Existen una serie de mecanismos de búsqueda exacta, pre-diseñados para este grafo, que localizan el nodo destino a partir de un nodo origen con pocos saltos. Todos estos mecanismos de búsqueda se basan en desplazar los dígitos de la clave origen S hacia la izquierda (L), o hacia la derecha (R) o una combinación de ambos (L_1RL_2, R_1LR_2, LR y RL) hasta obtener la clave destino D. Estos mecanismos están explicados en [LSA96, NNL04, NMVL05].
Estos mecanismos de desplazamientos de dígitos se traducen en la red Bruijn dirigiendo el mensaje a transmitir hacia al nodo vecino derecho o izquierdo, según sea la dirección del desplazamiento (L o R), al que tiene como índice de enlace el nuevo dígito a insertar para generar la siguiente clave intermedia con el fin de llegar al destino. En la Fig. 4.5 se muestra un ejemplo de cómo se dirige un mensaje M hacia una clave destino $D = 102021$ partiendo de una clave origen $S = 120102$ para un Bruijn(3, 6).
Por otro lado, en la Fig. 4.6 se muestran los movimientos que producen los diferentes mecanismos de desplazamiento para formar la clave destino D. En la misma Figura, el símbolo l indica los dígitos a desplazar a la izquierda y el símbolo r los dígitos hacia la derecha. Las zonas grises oscuras de las claves representan la subcadena de dígitos comunes de ambas claves, S y D, que reducen el camino en la obtención de D.

El primer tipo de desplazamientos, R_1LR_2 y L_1RL_2, se utilizan en pocos casos. El primero R_1LR_2 consiste en hacer primero un desplazamiento de $r1$ dígitos a la derecha, después l dígitos a la izquierda y finalmente $r2$ dígitos a la derecha. En cambio el otro, L_1RL_2 realiza primero un desplazamiento de $l1$ dígitos a la izquierda, después r dígitos a la derecha y finalmente otra vez $l2$ movimientos a la izquierda.
A continuación, para los siguientes mecanismos LR y RL, existen dos tipos de desplazamientos diferentes: para LR, se realiza un desplazamiento de l dígitos a la izquierda y después r dígitos a la derecha. Si $r \geq l$, se trata de un $LR1$, sino es un $LR2$. Estos dos tipos de desplazamientos no se eligen a priori, sino

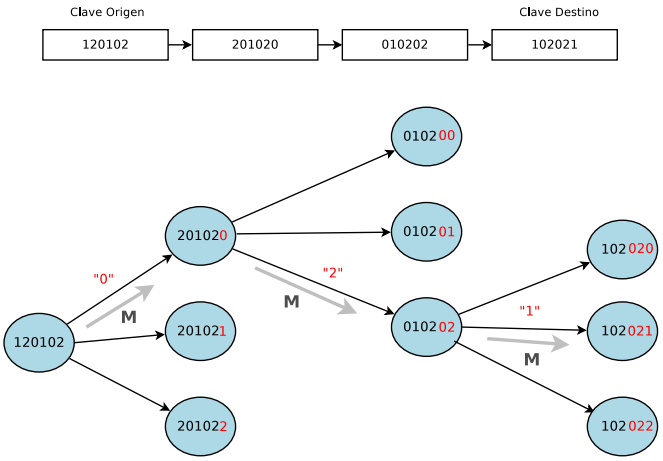

Figura 4.5: Ejemplo de routing dentro de la red Bruijn.

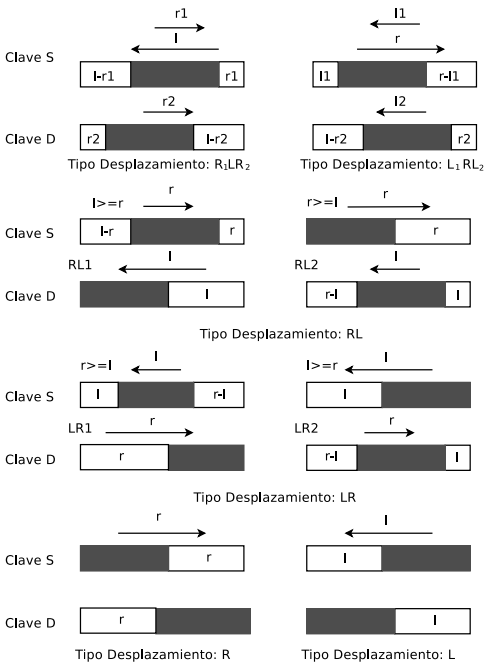

Figura 4.6: Tipos de desplazamientos aplicables al grafo Bruijn [NNL04, LSA96]

que vienen dados según los valores de la clave origen y destino.
Del mismo modo, pero a la inversa, sucede para el caso RL. Primero desplaza r dígitos hacia la derecha y después l dígitos hacia la izquierda. En este caso, si $l \geq r$, es de tipo $RL1$, sino es de tipo $RL2$. Esta distinción se emplea en el algoritmo *Fault Free Shortest Path FFSP*, comentado en la siguiente sección 4.4.2, donde solo utiliza los tipos $LR2$ y $RL2$.
Una primera propuesta de búsqueda sería que a partir de una clave origen S y una clave destino D, antes de realizar una búsqueda de un recurso, se calculase previamente la longitud del camino de todos los mecanismos de desplazamiento de dígitos, y escoger el más corto. De este modo se encontraría una ruta candidata al recorrido más corto para llegar al nodo D. El algoritmo *Fault Free Shortest Path* (*FFSP*) comentando en el siguiente apartado 4.4.2 aplica esta técnica.
Para mostrar un ejemplo del mecanismo de desplazamiento de dígitos de las claves Bruijn, en la Tabla 4.1 se muestra la obtención de los caminos desde un nodo $S = 012011$ hacia $D = 022010$ para un Bruijn(3, 5). Primero, es necesario localizar los dígitos coincidentes entre la clave origen S y la clave destino D para así evitar saltos innecesarios para cada mecanismo. En el ejemplo, estos dígitos se reflejan en la Tabla 4.1, resaltados en negrita. La Fig. 4.6 muestra para todos los mecanismos, las zonas sombreadas de color gris que representan los dígitos que tienen que tener en común ambas claves y donde deben estar ubicadas (parte izquierda, centro, derecha), para así reducir el número de pasos para encontrar la clave D. Finalmente, en el ejemplo de la tabla 4.1, después de mostrar el desarrollo de las rutas por cada mecanismo, vemos como el resultado final del camino más corto lo obtienen las rutas de R y RL con 5 saltos.

4.4.2. Algoritmos de búsquedas exactas en DisCoP

Para la localización de recursos computacionales en DisCoP, se han diseñado tres algoritmos de búsqueda exacta que emplean un grafo Bruijn no dirigido para localizar los recursos. Estos utilizan los mecanismos predefinidos en la sección anterior 4.4.1. Cabe destacar que la búsqueda exacta equivale

Nodo Origen S: 012011 Nodo Destino D: 022010 Bruijn(3,5)		Saltos
R_1LR_2	$01\mathbf{201}1 \rightarrow_{R_1} x01\mathbf{201} \rightarrow_L 01\mathbf{201}0 \rightarrow_L 1\mathbf{201}0x \rightarrow_L \mathbf{201}0xx \rightarrow_{R_2} 2\mathbf{201}0x \rightarrow_{R_2} 02\mathbf{201}0$	6
L_1RL_2	$01\mathbf{201}1 \rightarrow_{L_1} 1\mathbf{201}1x \rightarrow_{L_1} \mathbf{201}1xx \rightarrow_R 2\mathbf{201}1x \rightarrow_R 02\mathbf{201}1 \rightarrow_R x02\mathbf{201} \rightarrow_{L_2} 02\mathbf{201}0$	6
RL	$\mathbf{01}2011 \rightarrow_R \mathbf{201}201 \rightarrow_R 2\mathbf{201}20 \rightarrow_R 02\mathbf{201}2 \rightarrow_R x02\mathbf{201} \rightarrow_L 02\mathbf{201}0$	5
LR	$012011 \rightarrow_L 12011x \rightarrow_L 12011xx \rightarrow_L 011xxx \rightarrow_L 11xxxx \rightarrow_L 1xxxxx \rightarrow_L xxxxxx \rightarrow_R 0xxxxx \rightarrow_R 10xxxx \rightarrow_R 010xxx \rightarrow_R 2010xxx \rightarrow_R 22010x \rightarrow_R 022010$	12
L	$012011 \rightarrow_L 120110 \rightarrow_L 201102 \rightarrow_L 011022 \rightarrow_L 110220 \rightarrow_L 102201 \rightarrow_L 022010$	6
R	$\mathbf{0}12011 \rightarrow_R 1\mathbf{0}1201 \rightarrow_R 01\mathbf{0}120 \rightarrow_R 201\mathbf{0}12 \rightarrow_R 2201\mathbf{0}1 \rightarrow_R 02201\mathbf{0}$	5

Tabla 4.1: Tabla de cálculo de los caminos con diferentes mecanismos de desplazamiento de dígitos de la clave Bruijn entre un nodo origen S y el nodo final D.

conceptualmente a búsqueda simple.
Además, estos tres algoritmos de búsqueda exacta pueden acoplarse al algoritmo de búsqueda compleja de DisCoP comentado en la siguiente sección 4.4.3. En la Fig. 4.7 se ilustra el diagrama de flujo del mecanismo de búsqueda exacta de DisCoP, compuesto por tres fases:

- La **fase 1** equivale a la codificación de los valores de los atributos del recurso que se desea localizar mediante la función de mapeo Hilbert.
- La **fase 2** realiza el cálculo del camino óptimo para encontrar el Mercado con los recursos deseados (D), según el algoritmo elegido, *Left + Right*, *Left + Right + Hilbert* o *FFSP2*, descrito a continuación.
- Finalmente, en la **fase 3**, una vez obtenido el camino teórico, se envía el mensaje de solicitud de búsqueda a través de la red Bruijn, siguiendo nodo por nodo el camino calculado en la **fase 2** hasta encontrar el Mercado D.

A continuación se explica detalladamente los tres algoritmos de búsqueda exacta implementados en DisCoP:

- **Algoritmo de búsqueda exacta L+R** ($Left + Right$): Este mecanismo se basa en calcular cual es el camino más corto partiendo del los

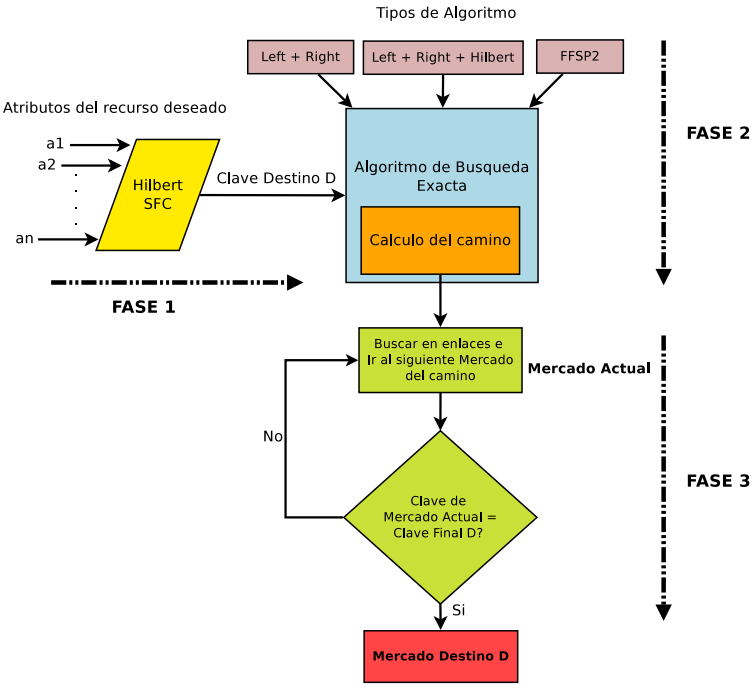

Figura 4.7: Diagrama de flujo de los mecanismos de búsqueda simple de DisCoP

mecanismos de desplazamiento L y R antes de transmitir un mensaje hacia el Mercado destino. Una vez encontrado el camino más corto se elije como ruta para transmitir el mensaje.

- **Algoritmo de búsqueda exacta L+R+H** ($Left + Right + Hilbert$): Este algoritmo emplea tres mecanismos de búsqueda que son $Left$, $Right$ y $Hilbert$, y escoge el camino más corto entre los tres. Hay que destacar que el camino Hilbert se basa en ir de nodo sucesor en sucesor hasta llegar al destino. Por lo tanto, para calcular la distancia entre un nodo S y D se aplica ($D_{Hilbert} = |D - S| \ mod \ N$), donde N es el número máximo de mercados del sistema.
  Tal como se vio en el capítulo 3, añadir este enlace Hilbert al grafo Bruijn ayuda a conservar la localidad en sistemas multi-atributo. No obstante, al aplicar este algoritmo a las búsquedas complejas multi-atributo y rango ayuda mucho a reducir el coste total de la búsqueda comparado con el anterior algoritmo. Este hecho se corrobora en el análisis experimental

de las búsquedas complejas, realizado en el apartado 4.4.5.2.

- **Algoritmo de búsqueda exacta *FFSP* (Fault Free Shortest Path)**: Ciertos autores [NNL04, NMVL05] investigaron la manera de encontrar el camino de búsqueda más corto con la restricción añadida de que se produjera fallos de nodos en los grafos Bruijn con un grado k más grande o igual a 2. Estos autores propusieron dos algoritmos de *routing* llamados *Fault Free Shortest Path (FFSP)*, el *FFSP1* y el *FFSP2*. Los resultados de simulación mostraron que el *FFSP2* es un candidato apropiado para las redes reales con un valor k elevado y un enorme número de nodos, mientras que el *FFSP1* es una buena elección para las redes con alta tolerancia a fallos, con un bajo valor de k y para un número de nodos pequeño o mediano.
  La complejidad temporal de *FFSP2* en la resolución de la ruta óptima es $\theta(2^{\frac{d}{2}+1} \cdot k)$, en comparación con $\theta((2 \cdot k)^{\frac{d}{2}+1})$ de *FFSP1* (en un $Bruijn(k, d)$). Teniendo en cuenta que la probabilidad de fallo de un Mercado es muy baja, analizado en la sección 3.5.3, y que la complejidad del tiempo de resolución de los *FFSP* es muy alta, se escogió el algoritmo *FFSP2* como candidato para el enrutamiento de mensajes. Los Alg. 6 y 7 se corresponden con una posible implementación de *FFSP2*.
  El algoritmo FFSP2 emplea algunos de los diferentes mecanismos de desplazamiento de dígitos de la clave Bruijn (L, R, LR, RL), teniendo en cuenta una serie de teoremas y corolarios descritos en [NNL04, NMVL05]. Su método consiste en calcular desde el nodo origen al destino todos los vecinos de los nodos que se encuentra por el camino. Los nodos vecinos están guardados y ordenados según un índice de profundidad de exploración en dos arrays A (nodo origen) y B (nodo destino). Además, para cada nodo vecino calculado, guarda la clave del nodo anterior desde donde llegó, obteniendo posteriormente el camino final.
  En cada iteración, después de calcular los nodos vecinos, el algoritmo elimina los nodos redundantes que no sirven para encontrar el camino más corto. Para realizar esta tarea es necesario encontrar un tipo de nodos llamados *Elementos Dominantes*. Según la Definición 2 de [NNL04], un *Elemento Dominante* es un nodo que si se encuentra en el camino en-

Algoritmo 6 Nivel 1 del Algoritmo FFSP2

function Algoritmo_busqueda_FFSP2(N_{origen},$N_{destino}$)
Require: A,B: Array[][]
A[0]:={N_{origen}};
B[0]:={$N_{destino}$};
i:=j:=1;
salir:=**false**;
while (¬*salir*) **do**
new A[i+1]:=proceso_SPD(A[i]);
$i := i + 1$;
if ($u \in$ A[i] es vecino de $v \in$ B[j]) **then**
salir:=**true**;
end if
new B[j+1]:=proceso_SPD(B[j]);
$j := j + 1$;
end while
{Calcula el camino a partir del nodo colisionado u y las marcas desde los nodos orígenes donde se ha creado}
Camino = Calcula_Camino(A[],B[],u);
end function

tre el nodo origen y el destino, asegura que éste es el camino más corto. Para encontrar un Elemento Dominante, los mismos autores, emplean el Teorema 2 según el cual, dado un conjunto de nodos T que difieren en 1 dígito en la posición más a la izquierda, entonces el Elemento Dominante de éste es el nodo $k \in T$ que tiene el camino más corto hacia el destino empleando la búsqueda $RL2$ y R. Por otro lado, si existe un conjunto T de nodos que difieren en el dígito de la posición más a la derecha, entonces se aplica $LR2$ y L para encontrar el Elemento Dominante. Una vez localizados los Elementos Dominantes del nivel actual, $A[i]$ y $B[j]$, el algoritmo continúa la siguiente iteración, explorando un nuevo nivel de vecinos y elementos dominantes a partir de los anteriores.

Finalmente, este algoritmo termina cuando un nodo $u \in A[i]$ es vecino de $B[j]$. A partir de aquí, se obtiene el camino óptimo, trazando el recorrido partiendo del nodo u y de los arrays $A[i]$ y $B[i]$, siguiendo tanto desde A como desde B, la ruta por todos los nodos desde donde se llegó al nodo u.

Algoritmo 7 Nivel 2 del Algoritmo FFSP2

function proceso_SPD(B_j) : B_{j+1}
Require: N, $N_{dominantes}$: Array[];
{Calcula los nodos vecinos y marca en una variable el nodo de donde procede}
N=calcular_nodos_vecinos(B_j);
$N_{dominantes}$ = { };
while ($\exists k \in T \mid T \subseteq N$ de claves que difieren en 1 solo dígito en la posición más a la izquierda o la derecha) **do**
if (T difieren 1 dígito a la posición más a la izquierda) **then**
$N_{dominantes}$+={$k \in T$| long_camino_RL2(k, $N_{destino}$) $\wedge$ long_camino_R(k, $N_{destino}$)) es el más corto};
end if
if (T difieren 1 dígito a la posición más a la derecha) **then**
$N_{dominantes}$+={$k \in T$| long_camino_LR2(k, $N_{destino}$) $\wedge$ long_camino_L(k, $N_{destino}$)) es el más corto};
end if
end while
Eliminar_Duplicados($N_{dominantes}$);
Eliminar_Nodos_Fallidos($N_{dominantes}$);
return $N_{dominantes}$;
end function

4.4.3. Algoritmo de búsqueda compleja

En esta sección se presenta el algoritmo principal de búsqueda compleja de DisCoP. El algoritmo de búsqueda compleja tiene la habilidad de realizar peticiones de búsqueda multi-atributo y por rango. Un ejemplo de búsqueda compleja sería la petición que haría un usuario que desea encontrar Mercados con los recursos $CPU = \{2-4\}$ GHz, $Memoria = \{2-8\}$ Gbytes y $red = \{3-6\}$ Mbps. Para poder satisfacer esta consulta, el algoritmo descompone la consulta compleja en consultas simples optimizando el camino mediante 4 fases diferentes.
Para ver el proceso general del algoritmo, en la Fig. 4.8 se muestra el diagrama de flujo de la búsqueda compleja ilustrando todos los componentes implicados para satisfacer de manera óptima este tipo de consultas.

En los Alg. 8, 9 y 10 se muestra el modelo algorítmico de la búsqueda compleja y sus funciones implicadas para el desarrollo del mismo, de manera formal y descendiente. Para entender mejor el algoritmo, a continuación se

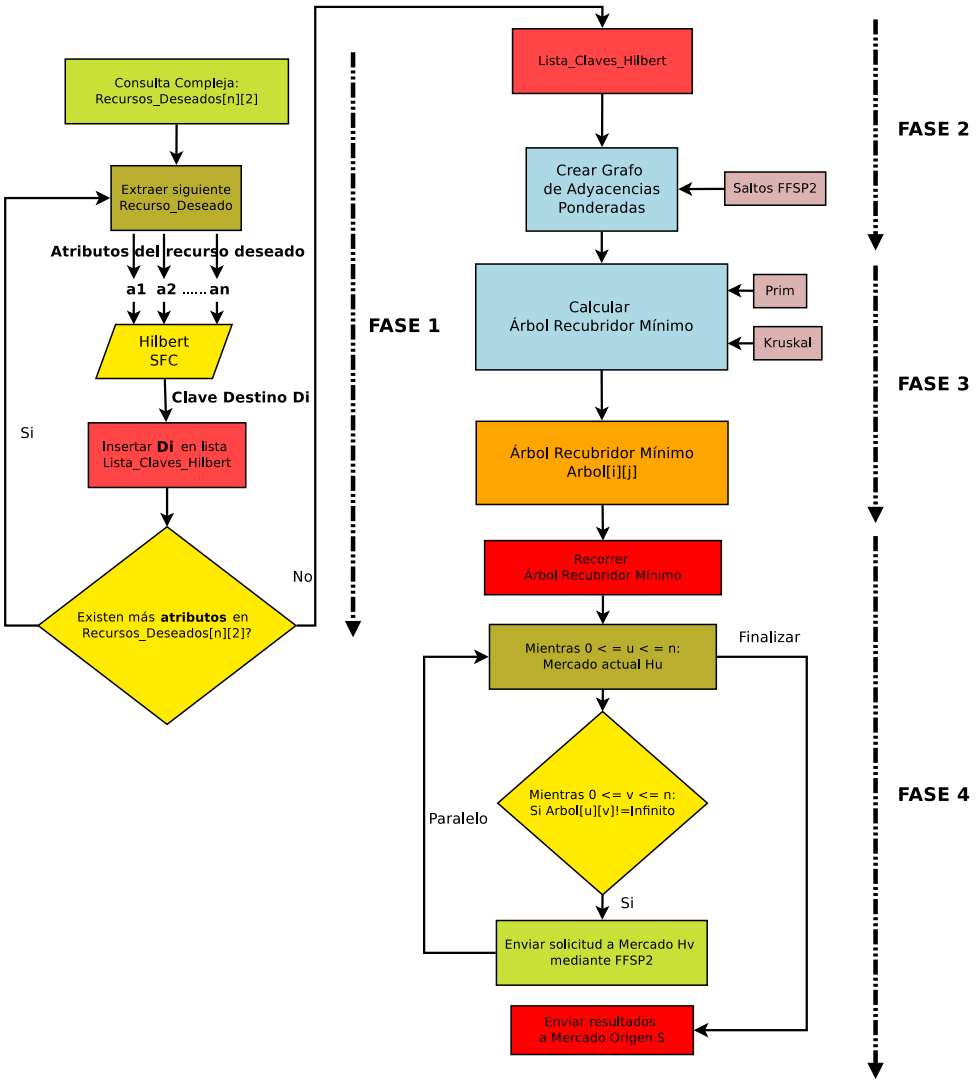

Figura 4.8: Diagrama de flujo del algoritmo de búsquedas complejas.

explica detalladamente las cuatro fases del cual está compuesto:

- **Primera Fase**: En esta fase, el algoritmo realiza el proceso de *descomponer la consulta compleja* en un subconjunto de claves Hilbert que satisfacen los valores de la consulta. Para ello, por cada tupla de atributos $< a_0, a_1, ..., a_n >$ que satisfagan la consulta se transforma, mediante la función Hilbert, en una clave H_i y a continuación se guarda en una lista de claves llamada *Lista_ Claves_ Hilbert*.

- **Segunda Fase**: Esta fase consiste en crear un grafo de adyacencias mediante una array de dos dimensiones $G_{ady}[u][v]$ utilizando sus índices u y v como los vértices del grafo para ser utilizado para encontrar el recorrido óptimo en la siguiente fase. Además los índices u y v representan también el índice de una clave H_i dentro de la lista *Lista_ Claves_ Hilbert*. El propósito de esta fase es llenar el grafo de adyacencias $G_{ady}[u][v]$ con los costes de las aristas igual al número de saltos necesarios para llegar a través de Bruijn desde una clave H_i (con índice u) hasta H_j (con índice v) utilizando el algoritmo de búsqueda simple *FFSP2* (comentado en la sección 4.4.2). De este modo, se obtiene un subgrafo conexo, completo y ponderado de todas las distancias entre los mercados que pertenecen al subconjunto de *Lista_ Claves_ Hilbert* que satisfacen la restricción de la consulta compleja.

- **Tercera Fase:** A continuación, el algoritmo de búsquedas complejas prosigue aplicando sobre la matriz de adyacencias $G_{ady}[u][v]$ unos algoritmos populares de recorrido de grafos llamados *Prim*[Pri] y *Kruskal*[Kru]. Ambos son algoritmos que buscan y construyen el *Árbol Recubridor Mínimo* [Arb], explorando el grafo inicial, a través de sus vértices para el algoritmo de Prim y las aristas para el de Kruskal. Un árbol recubridor mínimo o de expandido mínimo es un subgrafo que tiene forma de árbol, es decir, sin ciclos, que contiene todos los vértices del grafo inicial y además es mínimo cuando recubre todos los vértices del grafo general con las aristas de menor peso.
  El problema de encontrar el camino más corto pasando por todos los

vértices se llama *el problema del viajante* [pro]. Este problema es *NP-completo* y se resuelve combinando todos los caminos posibles hasta encontrar el de coste más bajo. Por lo tanto, el algoritmo tiene un coste del orden $\theta(n!)$, donde n es el número de vértices que hay que recorrer.
Aunque los algoritmos *Prim* y *Kruskal* no encuentran el camino óptimo, si que encuentran un Árbol Recubridor Mínimo que encaja muy bien con nuestros propósitos. Además, estos algoritmos tienen un coste muy bajo. *Prim* tiene un coste $\theta(n^3)$, donde n es el número de vértices y *Kruskal* tiene un coste $\theta(m \cdot \log(m))$, donde m es el número de aristas del grafo. Además, Prim calcula el árbol mediante la exploración de vértices y Kruskal lo realiza mediante la exploración de aristas. En el Alg. 9 se muestra el algoritmo de *Prim*. Según el análisis experimental de la sección 4.4.5.2, se ha escogido *Prim* como algoritmo para calcular el árbol recubridor mínimo porque obtiene mejores resultados sobre Bruijn respecto Kruskal.
Tal como se puede ver en el Alg. 9, el algoritmo *Prim* está compuesto por un bucle principal, donde a cada iteración va construyendo el árbol recubridor mínimo añadiendo un vértice nuevo. Para su construcción, primero extrae de una *Cola* de prioridades el vértice u con menor distancia al N_{origen} o al subárbol expandido mínimo. N_{origen} es el nodo origen que realiza la petición de búsqueda compleja y el array $distancias[]$ sirve para guardar las distancias actuales mínimas de cada vértice respecto a la instancia actual del subárbol recubridor mínimo.
El algoritmo, una vez obtenido el vértice u de la *Cola* que tiene la distancia más corta al árbol, obtiene sus vértices adyacentes v y comprueba sus distancias respecto a u. El vértice v que tenga menor distancia será el siguiente vértice que se añadirá al árbol recubridor mínimo. De este modo, el algoritmo prosigue iterativamente realizando la misma labor en cada vértice hasta extraer todos los vértices de la *Cola*, obteniendo de esta forma el Árbol Recubridor Mínimo final.

- **Cuarta Fase:** Una vez calculado el *Árbol Recubridor Mínimo*, el algoritmo de búsqueda compleja, envía la petición de búsqueda compleja a través de la red Bruijn siguiendo la trayectoria marcada por el árbol re-

cubridor mínimo. Esta fase está descrita formalmente en el Alg. 10. Al obtener en la fase anterior un subgrafo en forma de árbol sin ciclos, es muy fácil paralelizar las peticiones de búsqueda. Por cada mercado encontrado, si la petición de búsqueda tiene que continuar el envío transmitiendo a más de un destinatario, se puede realizar paralelamente por caminos diferentes, para así reducir la latencia final de búsqueda.
Además, cada ruta entre dos mercados se recorre mediante el algoritmo de búsqueda exacta *FFSP2*. Finalmente, una vez encontrados todos los mercados que satisfacen la consulta, se devuelve al nodo solicitante N_{origen} la dirección de red de todos ellos.

El coste total del algoritmo de búsqueda compleja para calcular la ruta óptima antes de transmitir la petición es $n^3 + n^2 \cdot (2^{\frac{d}{2}+1} \cdot k) + n$, de orden $\theta(n^3)$, donde n es el número de claves Hilbert contenidas dentro de la consulta compleja. Hay que decir que el componente n^3 del coste total pertenece al coste del algoritmo de Prim para encontrar el árbol (fase 3), $n^2 \cdot (2^{\frac{d}{2}+1} \cdot k)$ pertenece al coste de crear el grafo de adyacencias ponderadas empleando el algoritmo FFSP2 (fase 2) y finalmente n es el coste de convertir todas las consultas simples en la clave Hilbert (fase 1).
El coste máximo de búsqueda, en saltos, es $n \cdot \log_k(N_{Bruijn})$, donde n es el número de mercados a buscar y $\log_k(N_{Bruijn})$ es la distancia máxima entre dos mercados.

En la sección 4.4.5.2 se describe los resultados experimentales obtenidos en el análisis de este algoritmo.

4.4.4. Algoritmo de búsquedas aproximadas

Una vez implementado el algoritmo de búsquedas complejas, el overlay DisCoP tiene que tener capacidad de realizar búsquedas aproximadas para cuando no exista ningún mercado con los recursos deseados.
Los resultados aproximados incrementan la efectividad de las búsquedas solicitadas por los usuarios. Las búsquedas aproximadas de recursos en un entorno P2P de computación tienen mucho sentido. Por ejemplo, una búsqueda aproximada de un nodo con $CPU = 2GHz$, en el caso de que este no existiese,

Algoritmo 8 Fase 1 y 2: Algoritmo de búsqueda compleja

function N_{origen}.Algoritmo_busqueda_Compleja($Recursos\_Deseados$[][]: Array[0..$n-1$][2])

Require: Lista_Claves_Hilbert: Lista; G_{ady}: Array[n][n];

Ensure: Lista_Mercados: Lista

{Se crea una lista de las claves Hilbert que pertenecen al subconjunto de recursos deseados.}

for all $R_{deseado} \epsilon$ {$Recursos\_Deseados$[0..$n-1$][2]} **do**

$Key_{destino}$:= HilbertSFC($R_{deseado}$);

Lista_Claves_Hilbert.insertar($Key_{destino}$);

end for

new Array G_{ady}[|Lista_Claves|][|Lista_Claves|] := {};

{Se construye un subgrafo completo conexo con los vértices que son las claves a buscar y las aristas ponderadas con el número de saltos utilizando FFSP2 en el grafo Bruijn general}

for $u = 0$ hasta $u <$ |Lista_Claves_Hilbert| paso 1 **do**

H_i := Lista_Claves_Hilbert.obtener(u);

for $v = 0$ hasta $v <$ |Lista_Claves_Hilbert| paso 1 **do**

H_j := Lista_Claves_Hilbert.obtener(v);

$G_{ady}[u][v]$:= Calcular_Numero_Saltos_FFSP2(H_i, H_j);

end for

end for

{Se calcula el árbol recubridor mínimo mediante el Alg. Prim}

arbol := Calcular_arbol_optimo_Prim(G_{ady}, N_{origen}, |Lista_Claves_Hilbert|);

{Se localiza todos los mercados deseados siguiendo el árbol recubridor.}

Lista_Mercados:=Ejecutar_Recorrido_Arbol(arbol, Lista_Claves_Hilbert);

return Lista_Mercados;

end function

Algoritmo 9 Fase 3: Algoritmo de encontrar el Árbol Recubridor Mínimo mediante *Prim*

function N_{origen}.Calcular_arbol_optimo_Prim(G_{ady}, N_{origen}, n)
Require: *distancia*[]: Array[0..$n-1$];
Ensure: *arbol*: Array[0..$n-1$][0..$n-1$]
  {Añade a una cola de prioridad todos los vértices y su prioridad que es la distancia respecto al nodo origen}
  for $v = 0$ hasta $v < cn$ paso 1 **do**
    distancia[v] = ∞;
    $Cola$.añadir(<v,distancia[v]>);
  end for
  {Inicializa la variable "arbol" todo a infinito}
  for $u = 0$ hasta $u < n$ paso 1 **do**
    for $v = 0$ hasta $v < n$ paso 1 **do**
      arbol[u][v] := ∞;
    end for
  end for
  distancia[N_{origen}]:=0;
  padre := *null*;
  {Extrae de forma ordenada todos los vértices con distancia mínima respecto al origen y marca la adyacencia seleccionada en "arbol"}
  while $|Cola| \neq 0$ **do**
    u := $Cola$.extraer_minimo();
    if (padre $\neq$ *null*) **then**
      arbol[padre][u] := G_{ady}[padre][u];
      arbol[u][padre] := G_{ady}[padre][u];
    end if
    padre := u;
    for all (v | v es adyacente de u) **do**
      if ($v \in Cola \ \wedge$ (distancia[v] > G_{ady}[u][v])) **then**
        distancia[v] := G_{ady}[u][v];
        $Cola$.actualizar(<v,distancia[v]>);
      end if
    end for
  end while
  return arbol;
end function

Algoritmo 10 Fase 4: Recorrido del Árbol Recubridor Mínimo a través del grafo Bruijn

```
function N_k.Ejecutar_Recorrido_Arbol(arbol, Lista_Claves_Hilbert)
Require: k : Índice del nodo N_k;
Ensure: Lista_Mercados: Lista
  Lista_Mercados.añadir(N_k);
  for v = 0 hasta v < |Lista_Claves_Hilbert| paso 1 do
    if (arbol[k][v] ≠ ∞) then
      parallel fork
        H_v.Lista_Claves_Hilbert.obtener(v);
        N_v:=N_k.Enviar_consulta_con_FFSP2(H_v);
        Lista_Mercados := Lista_Mercados+N_v.Ejecutar_Recorrido_Arbol(arbol,
      Lista_Claves_Hilbert);
      end fork
    end if
  end for
  return Lista_Mercados;
end function
```

podría ser retornada con otros resultados como CPU =1.5 o 2.5GHz, si estos existiesen.

En DisCoP se ha diseñado un mecanismo de búsqueda aproximada que puede acoplarse a los algoritmos descritos anteriormente. En el Alg. 11 se muestra el algoritmo de búsqueda aproximada empleando el algoritmo de búsqueda exacta *FFSP2*.

El procedimiento consiste en utilizar el algoritmo de búsqueda exacta *FFSP2*, para buscar el mercado deseado. En caso de encontrarse el mercado solicitado, simplemente se devuelve el resultado. Por otro lado, si el algoritmo no encuentra el resultado exacto entonces busca el mercado más similar aplicando la virtualización de nodos Bruijn y el enlace extra Hilbert. De este modo, cuando se encuentra un mercado virtual dentro de uno real, el mercado actual redirige la búsqueda a través de los enlaces extra Hilbert en ambos sentidos (enlaces sucesores y predecesores de Hilbert) hasta encontrar dos mercados reales. Cuando se obtienen las claves H_{pred} y H_{succ} de los mercados reales en el Alg. 11, $Mercado_{pred}$ y $Mercado_{succ}$, se transforma sus claves mediante la función inversa de Hilbert ($Hilbert^{-1}$) a los valores de los atributos originales. A partir de aquí, se calcula la distancia euclidiana de éstos, como si fuesen

puntos multi-dimensionales, respecto con los valores de los atributos deseados. El mercado que tenga la distancia euclidiana más pequeña respecto al recurso original será el que tenga los atributos más similares al solicitado y por lo tanto será el resultado devuelto por el algoritmo de búsqueda aproximada.
El coste de este algoritmo es $\theta(\log_k(N_{Bruijn}) + c)$, donde c es la variable que indica el número de saltos extra que tiene que hacer el algoritmo para encontrar los recursos aproximados.

Algoritmo 11 Algoritmo de búsquedas aproximadas

function $Peer_i$.Algoritmo_busquedas_aproximadas()
Require: $atrib\_deseados, atrib_{pred}, atrib_{succ} : Array[0..n-1]$; $dist_{pred}$; $dist_{succ}$;
Ensure: $Mercado$
  $clave\_destino :=$ Hilbert($Recursos\_Deseados$)
  $Mercado :=$ Algoritmo_de_busqueda_FFSP2($clave\_destino$)
  $Mercado_{pred} := Mercado_{succ} := Mercado$
  while $(Mercado_{pred} = virtual)$ **do**
    $Mercado_{pred} := Mercado_{pred}.predecesor$;
  end while
  while $(Mercado_{succ} = virtual)$ **do**
    $Mercado_{succ} := Mercado_{succ}.successor$;
  end while
  $atrib_{pred}[0..n-1] :=$ Hilbert$^{-1}(H_{pred})$;
  $atrib_{succ}[0..n-1] :=$ Hilbert$^{-1}(H_{succ})$;
  $dist_{pred} := \sqrt{\sum_{i=0}^{n}(atrib\_deseados[i] - atrib_{pred}[i])^2}$;
  $dist_{succ} := \sqrt{\sum_{i=0}^{n}(atrib\_deseados[i] - atrib_{succ}[i])^2}$;
  if $dist_{succ} \leq dist_{pred}$ **then**
    $Mercado := Mercado_{succ}$;
  else
    $Mercado := Mercado_{pred}$;
  end if
  return $Mercado$;
end function

4.4.5. Rendimento experimental de las búsquedas

En este apartado se realiza una experimentación mediante simulación para evaluar el rendimiento de las búsquedas simples y complejas en el nivel de Bruijn de DisCoP. El rendimiento de la búsqueda se mide utilizando el número de saltos necesarios para enviar un mensaje de un nodo origen hasta su destino.

Además se comparan los resultados obtenidos con otras dos topologías, *Chord* para el caso de búsquedas exactas y *Baton* para el caso de búsquedas complejas. Estas topologías se encuentran descritas en los capítulos 1 y 2.
En la sección 4.4.5.1 se explica la experimentación realizada con búsquedas exactas y en la sección 4.4.5.2 los resultados obtenidos en la evaluación de las búsquedas complejas.

4.4.5.1. Búsquedas exactas

Esta sección está compuesta por dos partes. La primera parte realiza una comparativa del rendimiento de búsqueda dentro de la red Bruijn(2,10) entre 5 mecanismos de búsqueda exacta, de los cuales los dos primeros (L, R) son mecanismos de desplazamiento de dígitos previamente implementados sobre Bruijn y los otros tres (L+R, L+R+H, *FFSP2*) son mecanismos diseñados sobre el overlay DisCoP. Por otro lado, la segunda parte consiste en una comparativa del mejor algoritmo de búsqueda exacta (*FFSP2*), con el mecanismo de búsqueda de la topología Chord.

En la Fig. 4.9 se muestra la latencia media en número de saltos de los cinco algoritmos de búsqueda exacta sobre la red Bruijn variando el número de peers insertados simultáneamente en el sistema. Tal y como puede observarse, todos los mecanismos aumentan el número de saltos al escalar la red. Esto corrobora la buena implementación de los algoritmos de simulación. En el caso de R y L, son los algoritmos con peores resultados debido a que su búsqueda solo se realiza en una dirección. Los resultados muestran que L en media se comporta un $2,09\,\%$ mejor que R. Si se observa con atención L+R y L+R+H se puede ver como en general obtienen una latencia de búsqueda más baja debido a que tienen múltiples opciones para decidir el camino más corto. Además L+R+H es algo mejor que L+R, un $1,10\,\%$ de media, debido a la influencia del enlace Hilbert. Esta diferencia es mínima en el caso de búsquedas exactas porque la *localidad* no tiene mucha incidencia en los resultados. El algoritmo *FFSP2* obtiene un rendimiento muy superior respecto a los anteriores debido a su efectividad en encontrar el camino más corto entre dos nodos. Tal y como se refleja también en la Figura, *FFSP2* es un $21,84\,\%$ más eficiente que L+R+H y un $29,84\,\%$ más que L.

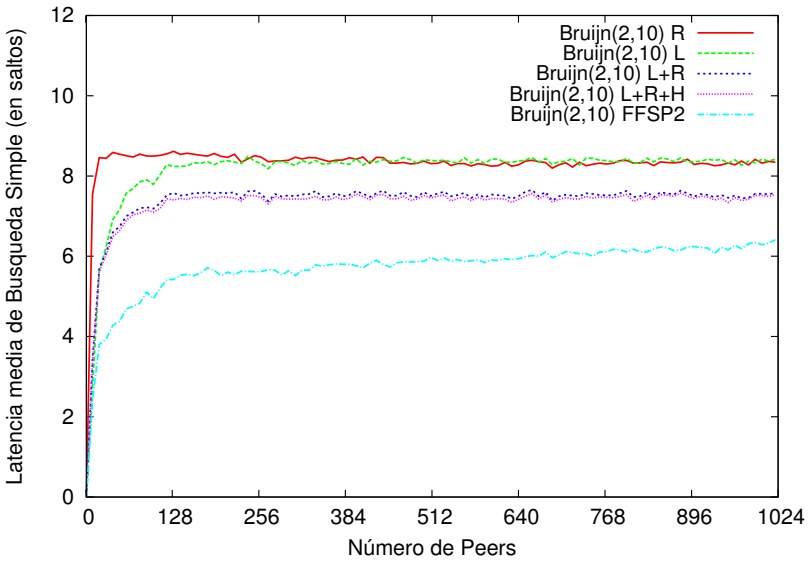

Figura 4.9: Gráfica del número de saltos de Bruijn con 5 algoritmos de búsqueda simple.

El mejor algoritmo de búsqueda exacta de Bruijn, que es *FFSP2*, se ha comparado con el mecanismo de búsqueda de Chord con 1.024 peers. En la Fig. 4.10 se muestran los resultados obtenidos. La línea roja, que representa la media de saltos de la topología Chord, crece logarítmicamente $\log_2(N)$. No obstante, el número de saltos es algo menor que el diámetro máximo $\simeq \frac{\log_2(N)}{2}$, porque el resultado es la media y no la distancia máxima. Asimismo, se puede ver como contrapartida que Chord supera a Bruijn(2,10) en un $10,51\,\%$, en el caso de *FFSP2* y en un $29,91\,\%$ en *L+R+H*. Esto es debido a que Chord tiene, en general, mucha más información de los peers de la red en su tabla de routing, con un máximo de 160 entradas, hecho que ayuda a rebajar bastante la latencia media de búsqueda. Sin embargo, si se crea un Bruijn(4, 5), es decir un Bruijn de grado $k = 4$, entonces cambian las tornas. En este caso, la latencia de *FFSP2* y *L+R+H* es mucho menor que Chord, un $20,42\,\%$ y un $17,55\,\%$ respectivamente.

4.4.5.2. Búsquedas complejas

En este apartado se analiza el rendimiento de las búsquedas complejas basadas en multi-atributo y en rangos. Para realizar la experimentación, se

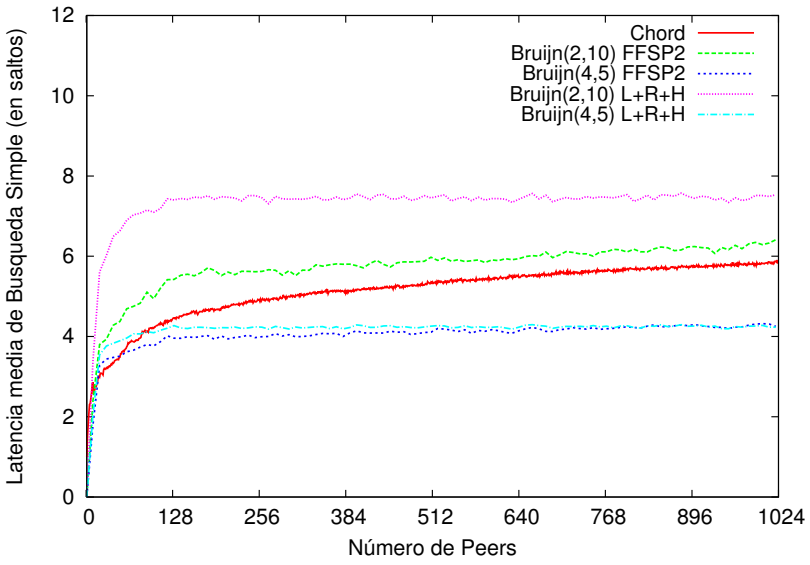

Figura 4.10: Gráfica del número de saltos de Bruijn vs. Chord en búsquedas exactas.

ha definido para cada búsqueda efectuada, una consulta $2-tupla$ de tipo $([a_f - a_0], [b_f - b_0])$ de números aleatorios con una variabilidad del 25 % por cada atributo con respecto el número máximo de valores (32) por atributo. Se han realizado 5.000 búsquedas complejas en cada algoritmo, donde el dominio de cada $2-tupla$ definida tenía un promedio de 64 tuplas simples.

Además, igual que en el apartado anterior, estas pruebas también están compuesta de dos partes. La primera parte realiza una comparativa entre el rendimiento de las búsquedas complejas, empleando como algoritmo de ruta simple *FFSP2* en un grafo Bruijn(2, 10) pero probando los dos algoritmos de obtención del *Árbol Recubridor Mínimo*, *Prim* y *Kruskal*, comentados en la sección 4.4.3. El objetivo de esta primera parte de la experimentación consiste en determinar el algoritmo de búsqueda del árbol más apropiado para nuestro propósito. En la segunda parte se realiza otra prueba, esta vez comparando el rendimiento del algoritmo de búsqueda compleja utilizando el algoritmo de búsqueda del árbol recubridor mínimo, Prim o Kruskal, que mejor se adapta a la topología de Bruijn con respecto a otra red P2P, como es *Baton*.

La Fig. 4.11 muestra los primeros resultados obtenidos empleando los dos algoritmos, *Prim* y *Kruskal*, midiendo la latencia media en saltos, cuando la

búsqueda se hace en serie o en paralelo. Tal como se muestra, para el caso de búsquedas complejas en serie, los dos tienen un rendimiento muy similar, con una diferencia de tan solo $0,37\,\%$ de *Prim* respecto *Kruskal*. En cambio, el rendimiento en saltos en paralelo es bastante diferente, un $11,98\,\%$ mejor Prim que Kruskal. Este hecho es debido a que ambos buscan el Árbol Recubridor Mínimo de distintas maneras, uno por vértices y el otro por aristas, respectivamente. En el caso de la búsqueda compleja en serie, el coste en saltos de los dos recorridos son muy similares, porque encuentran el árbol de recorrido de mínimo peso.
No obstante, el rendimiento en las búsquedas paralelas no es el mismo porque en el subgrafo inicial de la *fase 2*, donde se calcula el recorrido óptimo, existen muchos caminos ambiguos porque es un grafo no dirigido, conexo y completo. A causa de esto, y de la manera de construir el árbol de los algoritmos Prim y Kruskal, para encontrar el recorrido mínimo, los árboles son diferentes topológicamente. Como resultado final, el árbol de *Prim* contiene menos niveles de profundidad y más hijos por nodo, lo que optimiza significativamente el paralelismo de las búsquedas. Por lo tanto, *Prim* es el algoritmo que mejor se adapta al grafo Bruijn para la creación del *Árbol Recubridor Mínimo* para las búsquedas complejas de *DisCoP*.

Una vez visto el mejor algoritmo para encontrar el árbol mínimo (*Prim*), la Fig. 4.12 muestra el rendimiento obtenido en las búsquedas complejas con respecto a la topología de *Baton*. Además, para el caso de Bruijn, aparte del mecanismo de búsqueda exacta *FFSP2*, se emplean otros mecanismos como *L+R* y *L+R+H*.
De este modo, examinando los resultados, el peor caso que se obtiene y además con bastante diferencia con respecto a los demás, es Baton. Aunque esté optimizado para búsquedas basadas en rango, Baton no lo está para búsquedas multi-atributo. De modo que su algoritmo de búsqueda tiene que hacer muchas más peticiones para encontrar todos los nodos. Para el caso de Bruijn, el mejor algoritmo de búsqueda compleja en serie es para el que emplea *FFSP2* y *L+R+H*, donde existe poca diferencia, $(-7,6\,\%)$. Esto es debido a que el enlace Hilbert optimiza muy bien las búsquedas complejas porque conserva la propiedad de la *localidad*, utilizando en este caso el enlace

extra Hilbert para ir de un mercado a otro con recursos similares con gran eficiencia. Si comparamos *L+R+H* con *L+R* en serie se puede ver como mejora un 75 % más su rendimiento.

No obstante, los mejores resultados de rendimiento para la búsqueda compleja se producen en los casos que se aplica paralelismo. En la Figura se puede ver como todos los algoritmos de búsqueda en paralelo, exceptuando *L+R*, se adaptan al número de nodos actual de la red, obteniendo una latencia media de 18 saltos, un valor mínimo de 6 y un máximo de 33 aproximadamente. En cambio, para el caso del algoritmo *L+R*, el resultado se mantiene constante para cualquier número de peers, con una latencia media de 18 saltos. Esto es debido a que el *Árbol Recubridor Mínimo* construido por el algoritmo de *Prim* acostumbra a estar más balanceado cuando no existe el enlace Hilbert, porque solo utiliza los enlaces de Bruijn que no conservan localidad y por lo tanto la mayor parte de los enlaces del subgrafo de adyacencias construido en la fase inicial, tiene muchas aristas con pesos grandes y muy pocas con pesos pequeños. Debido a esto, el árbol de Prim tiene muchas más ramas por nodo y menos niveles de profundidad, lo que aumenta el rendimiento paralelo, pero sin embargo la distancia entre mercados es mucho más grande.

Por último, se puede concluir que los resultados de los algoritmos de búsqueda compleja en serie con *FFSP2* y *L+R+H* son muy similares y tienen un rendimiento muy superior a Baton, con un 76 % aproximadamente. Además, aplicar el enlace Hilbert sobre las búsquedas complejas es muy importante porque incrementa el rendimiento de las búsquedas hasta un 75 %.

4.4.5.3. Búsquedas aproximadas

La Fig. 4.13 muestra la efectividad de las búsquedas aproximadas multi-atributo del algoritmo de búsquedas aproximadas. En ella se muestra la distancia euclidiana entre el mercado deseado y el mercado encontrado en relación al porcentaje de mercados físicos ($Num.\ Mercados/N_{Bruijn}) \times 100$) existentes en la red Bruijn. Estos valores se obtuvieron con un $Bruijn(2, 16)$ y una curva Hilbert de 4 atributos y 4 bits/atributo. En resumen, se puede ver como las distancias de los resultados aproximados disminuyen siguiendo una tendencia

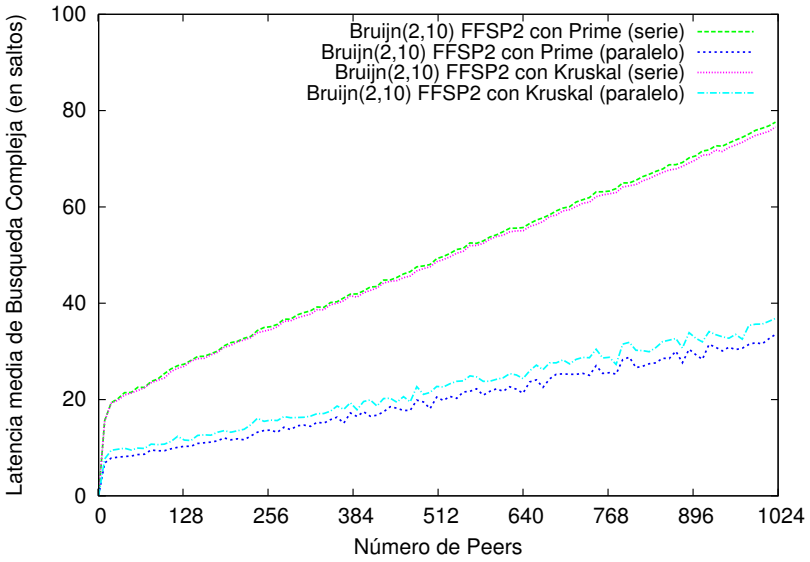

Figura 4.11: Gráfica del número de saltos de Bruijn con los algoritmos *Prim* y *Kruskal* con peticiones de búsqueda complejas.

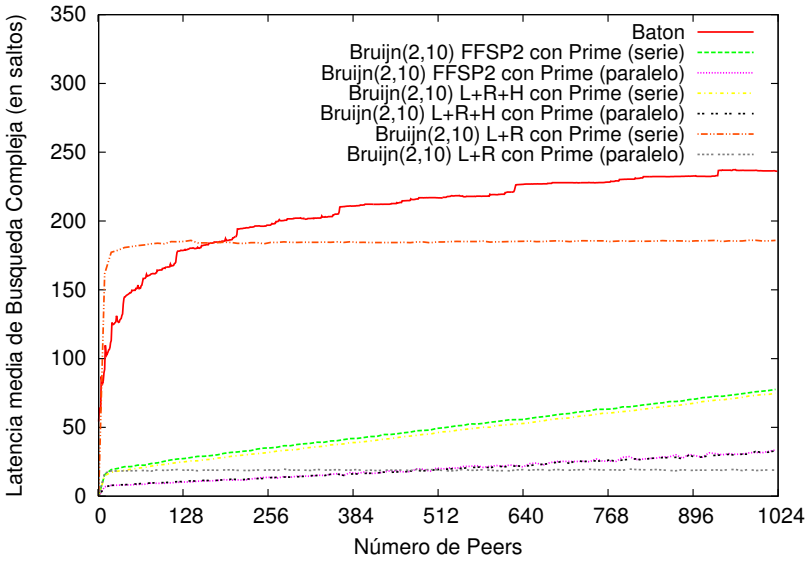

Figura 4.12: Gráfica del número de saltos de *Bruijn* vs. *Baton* con peticiones de búsqueda complejas.

exponencial cuando el porcentaje de creación aumenta. Asimismo, esta conducta demuestra como el método de búsqueda aproximada es capaz de aprovechar la propiedad de *clustering* de Hilbert eficientemente.

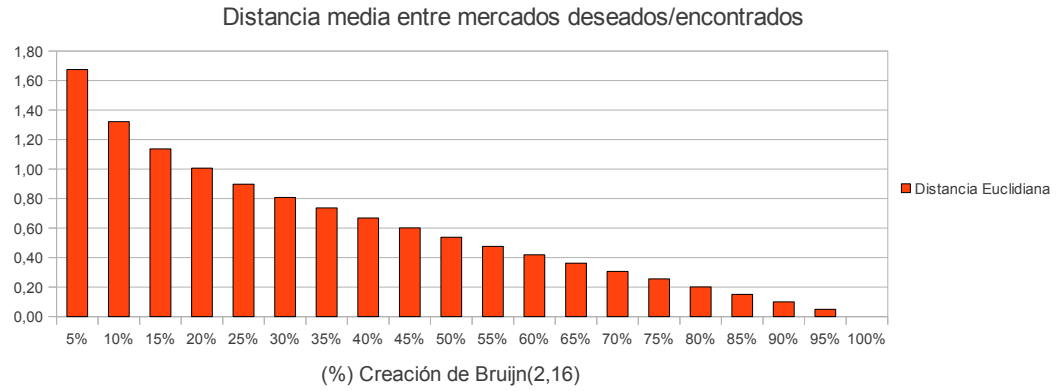

Figura 4.13: Distancia entre mercados solicitados y mercados encontrados en función del % de creación de Bruijn.

4.4.6. Congestión de la red

La congestión de redes es el fenómeno producido cuando una red o parte de ella tiene más tráfico del que puede transmitir. En este análisis se pretende medir como se distribuyen los mensajes enviados por nodos, aplicando los algoritmos de búsqueda anteriormente descritos, a través de los enlaces de la topología. De esta manera se pretende medir si la congestión está balanceada a lo largo de la topología y si es propensa a que haya saturaciones de nodos o cuellos de botella que pueden provocar una bajada del rendimiento general de la red.

La Fig. 4.14 muestra la distribución de la congestión obtenida por cada mercado en una red de Bruijn(2,10) completa y un Chord con una clave de 10 bits, en el caso de búsquedas exactas. Por otro lado, en la Fig. 4.15 se muestra la comparativa de búsquedas complejas entre Bruijn y Baton. Hay que destacar que en ambos casos se han realizado 10.000 búsquedas. En el caso

de las búsquedas complejas se han utilizado los mismos parámetros que en la sección 4.4.5.2, con consultas de tipo $2 - tupla$ y una variabilidad del 25 %. En este caso, la congestión se ha medido como el ratio entre el número de mensajes que atraviesan un nodo, o mercado en el caso concreto de la arquitectura DisCoP, y el número de mensajes totales enviados en el sistema. En ambos casos, la congestión es siempre menor del 1 %. En general, se puede ver como la congestión en las búsquedas simples está mejor balanceada en Bruijn, con muy poca variación respecto la media 0,098 %, ocurriendo lo contrario en el caso de Chord. Análogamente, en el caso de las búsquedas complejas, también se puede apreciar como la congestión en Bruijn está más dispersa y balanceada entre todos los nodos, al contrario que Baton. La causa de la pobre distribución de la congestión de Baton es debido, entre otras cosas, a la topología de árbol binario balanceado y su mecanismo de routing empleado.

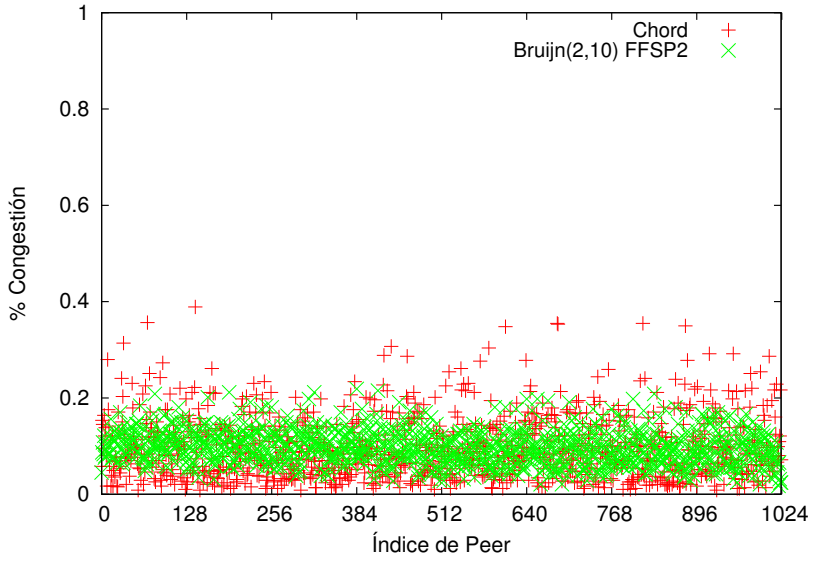

Figura 4.14: Congestión de las búsquedas simples de Bruijn vs. Chord.

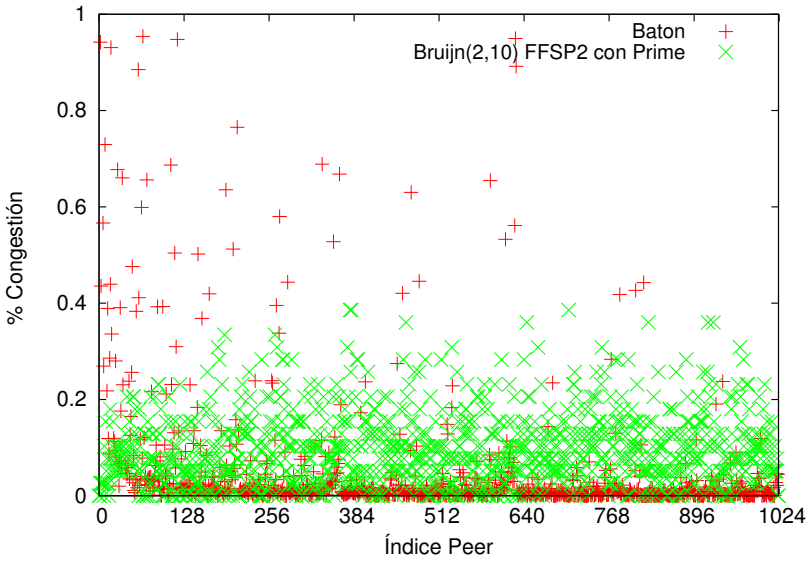

Figura 4.15: Congestión de las búsquedas complejas de Bruijn vs. Baton.

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo se describen las conclusiones obtenidas en el desarrollo del presente trabajo, junto con las principales aportaciones y publicaciones realizadas. Asimismo se explican las líneas abiertas que serán parte del desarrollo futuro de la investigación, y que alguno de sus puntos ya se está llevando a cabo.

5.1. Conclusiones

Las características de cómputo y conectividad de los sistemas informáticos han llevado en los últimos años al surgimiento de un conjunto de aplicaciones que operan bajo el modelo de cómputo distribuido, denominado Peer-to-Peer (P2P). La computación P2P, entorno donde se enmarca nuestro trabajo, representa una alternativa emergente de bajo coste para proporcionar acceso a recursos de cómputo distribuido, de forma escalable y tolerante a fallos. Las arquitecturas P2P se aprovechan de la infrautilización de los ordenadores, integrándolos en una plataforma de compartición de recursos entre iguales y utilizando los ciclos de procesador y otros recursos ociosos de miles de ordenadores conectados a través de Internet, para proporcionar altas prestaciones a aplicaciones paralelas/distribuidas. Debido a que los sistemas P2P involucran la participación de los nodos de una red sin administración central, se deben considerar las siguientes características en dichos entornos: a) conectividad de red variable siguiendo cualquier topología, puesto que los nodos pueden conec-

tarse y desconectarse de ella de forma arbitraria, b) ubicación dinámica de los servicios y recursos, c) auto-organización, donde cada nodo regula su grado de participación en la red y d) entorno altamente dinámico y heterogéneo.

Las primeras aplicaciones P2P encontraron su espacio dentro de los sistemas de compartición de ficheros por Internet. En estos sistemas se utiliza la capacidad de almacenamiento excedente y el ancho de banda de red disponible para buscar y descargar todo tipo de archivos y contenidos. Recientemente, la comunidad científica ha mostrado su interés en utilizar el paradigma P2P para la computación de altas prestaciones. Sin embargo, estos primeros trabajos han estado básicamente orientados al desarrollo de frameworks, con funcionalidades limitadas, que permitiesen la ejecución de aplicaciones distribuidas muy concretas, en este tipo de entornos.

El presente trabajo pretende abordar el diseño de una plataforma P2P de cómputo distribuido de propósito general, que aproveche de forma eficiente la potencialidad de recursos existentes en Internet, proporcionando mecanismos de calidad de servicio y tolerancia a fallos. Garantizar requerimientos de calidad de servicio a las aplicaciones paralelas se ha convertido en un objetivo crítico para numerosas plataformas desarrolladas por la comunidad científica. La calidad de servicio depende de las características particulares de las aplicaciones y de la arquitectura computacional (en nuestro caso P2P). El modelo de calidad de servicio que se quiere abordar para la distribución de cómputo basada en P2P pretende garantizar: tiempo de retorno de la aplicación, coste de ejecución asumido en función de las prestaciones obtenidas y la disponibilidad del servicio cuando los usuarios lo requieran, independientemente de los fallos que se puedan producir. Cabe señalar que esto introduce importantes retos en los entornos de computación P2P, donde los recursos no están dedicados, son altamente dinámicos y heterogéneos.

Acorde con el objetivo descrito, el trabajo realizado en esta Tesis se ha estructurado en diferentes fases temporales. En primer lugar se realizó una investigación de los trabajos existentes en la literatura científica sobre redes P2P, no solo de cómputo distribuido, sino también de mecanismos de búsqueda de recursos en este tipo de entornos. En el capítulo 2 se ha comentado y clasificado la mayor parte de la información extraída de la literatura, resaltando

la situación actual de estas redes P2P orientadas al cómputo distribuido, así como las principales diferencias con respecto a nuestra propuesta.

A continuación se procedió a definir una primera arquitectura de computación P2P, denominada *CoDiP2P*, caracterizada por una topología de árbol *B-Tree*. Esta incipiente arquitectura permitió el desarrollo de las funcionalidades básicas necesarias para el desarrollo de un entorno de computación P2P. *CoDiP2P* fue implementado, mediante el framework JXTA, y evaluado con respecto a otras topologías tradicionales, como *Chord* y *Baton*. Los resultados obtenidos mostraron que la topología *B-Tree* se comportaba muy bien con un número pequeño de peers, pero que a medida que escalaba tenía serios problemas de congestión en los niveles superiores del mismo. Partiendo de este trabajo inicial, en una tercera fase se propuso una arquitectura de tres niveles, que fuese totalmente escalable y que permitiese agrupar eficientemente los nodos en función de sus recursos de computación. Este trabajo dio lugar a la definición de la arquitectura *DisCoP*, presentada en esta Tesis. *DisCoP* integra tres niveles diferentes:

1. El primer nivel está formado por una curva *Hilbert SFC*, cuya función es ordenar los nodos en mercados, acorde con sus recursos de cómputo disponibles. De todas las curvas SFC existentes en la literatura se eligió *Hilbert SFC* porque maximizaba el grado de localidad respecto al resto de curvas.

2. El segundo nivel está formado por un *grafo Bruijn* y su objetivo es conectar entre si todos los mercados de nodos, que integran el sistema, mediante un grafo altamente escalable, con un diámetro logarítmico y un grado constante de enlaces por nodo. Bruijn cumplía todos los requerimientos solicitados.

3. El tercer nivel de *DisCoP*, denominado *Mercados*, implementa los mercados mediante una topología en árbol *B-Tree*. De este modo, los nodos con características similares son agrupados bajo una misma estructura lógica (mercado) y al mismo tiempo se aprovecha la estructura de árbol diseñada en el sistema original *CoDiP2P*.

Asimismo, con objeto de adaptar el grafo Bruijn a los requerimientos de una topología P2P, se realizaron una serie de mejoras sobre el mismo: la creación de *nodos virtuales* cuando el grafo Bruijn está incompleto y la adición de *enlaces extra de Hilbert* para maximizar la localidad de Bruijn y así realizar búsquedas complejas eficientemente. Finalmente, la arquitectura *DisCoP* propuesta fue evaluada mediante simulación con respecto a otras topologías clásicas de la literatura, como son el anillo Chord y el árbol B-Tree, tanto a nivel de topología, como de tolerancia a fallos y localidad. Esta pruebas experimentales revelaron el buen rendimiento de la propuesta realizada. El diseño de la arquitectura *DisCoP*, junto con la evaluación de su rendimiento, está descrito en el capítulo 3 del presente trabajo.

La siguiente fase de trabajo consistió en la definición de las funcionalidades básicas de la arquitectura propuesta: inserción y salida de nodos, así como gestión de la estructura; para cada uno de los niveles que integran la arquitectura. Cada una de estas funcionalidades básicas fueron definidas a nivel algorítmico y evaluadas a nivel de simulación. Finalmente a partir de dichas funcionalidades, se diseñaron diferentes sistemas de búsqueda de recursos computacionales: multi-atributo, en rango y aproximadas. Por cada uno de los algoritmos de búsqueda propuestos se definieron sus costes, en función del número de nodos del sistema, y se realizaron pruebas experimentales que validaron su buen rendimiento con respecto a otros mecanismos de búsqueda clásicos, como son *Chord* o *Baton*. Esta fase del trabajo ha sido desarrollada a lo largo del capítulo 4 de esta Tesis.

El trabajo desarrollado a lo largo de esta Tesis ha dado pie a las siguientes aportaciones:

Nuevo modelo de gestión de recursos de cómputo en entornos P2P: La elevada dispersión de posibles recursos computacionales ofrecidos por un nodo: número de CPUs, porcentaje de CPU, memoria, disco, ancho de banda, etc., así como el amplio rango de valores posibles para cada uno de ellos, hace que la gestión de los recursos computacionales en un entorno P2P sea totalmente distinto a la gestión de ficheros compartidos, tradicionales de este tipo de sistemas P2P. Acorde con este reto, en esta tesis se propone un mecanismo de gestión de recursos basados en mer-

cados, de modo que cada mercado está constituido por un conjunto de nodos con recursos computacionales similares. De este modo se optimiza la búsqueda de recursos de cómputo a lo largo de la red. La descripción de este modelo, junto con su implementación en la arquitectura de tres niveles correspondiente, fue publicada en:

[CBGS10] D. Castellà, H. Blanco, F. Giné and F. Solsona. *Combining Hilbert SFC and Bruijn Graphs for Searching Computing Markets in a P2P System.* 16th International Euro-Par Conference on Parallel Processing (Euro-Par 2010), LNCS (Springer), volumen 6271, pages 471-483, 2010.

Diseño de una arquitectura de cómputo P2P de tres niveles: La gestión de recursos basada en mercados comportaba la definición de una arquitectura específica que se adaptase a sus requerimientos. Acorde con este objetivo, la arquitectura de tres niveles, *DisCoP*, fue definida y modelada. Como se comentó anteriormente, el diseño de *DisCoP* partió de un trabajo inicial basado en una topología en árbol, denominado *CoDiP2P*. Este trabajo preliminar se ha presentado en las siguientes publicaciones:

[CRB+08] D. Castellà, J. Rius, I. Barri, A. Guim, M. Orobitg, H. Blanco and F. Giné. *CoDiP2P: a Distributed Computing Architecture Oriented to P2P Environments.* XIX Jornadas de Paralelismo, 2008.

[CBR+08] D. Castellà, I. Barri, J. Rius, F. Solsona, F. Giné and F. Guirado. *CoDiP2P: a Peer-to-Peer Architecture for Sharing Computing Resources.* International Symposium on Distributed Computing and Artificial Intelligence (DCAI 2008), Advances in Soft Computing (Springer), volume 50, pages 293-303, 2008.

[CRB+09] D. Castellà, J. Rius, I. Barri, F. Giné and F. Solsona. *CoDiP2P: a New P2P Architecture for Distributed Computing.* Conference on Parallel, Distributed and Network-based Processing (PDP 2009), pages 323-329, 2009.

A partir de este trabajo inicial, realizado en una topología en árbol, se desarrolló y evaluó la arquitectura *DisCoP*, trabajo que ha sido enviado

en la siguiente publicación para su evaluación:

Aceptado D. Castellà, F. Giné and F. Solsona. *A Resilient Architecture Oriented to P2P Computing.* 10th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2011), fecha de congreso 25-27 agosto 2011.

Calidad de servicio orientada a la búsqueda de recursos computacionales en entornos P2P: En un entorno de computación P2P, con recursos no dedicados y muy dinámicos, la calidad de servicio se entiende como la capacidad de la plataforma para dar siempre un servicio óptimo, con un tiempo de respuesta rápido en la búsqueda de recursos deseados y que asegure, en la medida de lo posible, la correcta finalización de las aplicaciones lanzadas sobre la misma. En este sentido, el presente trabajo ha definido y evaluado diferentes mecanismos de búsqueda de recursos computacionales: búsquedas exactas, en rango y aproximadas, que se adapten a los estrictos requerimientos del sistema. Nuestras propuestas han sido evaluadas, con respecto al número de saltos necesarios para obtener determinados recursos, en función del tamaño del sistema. Asimismo han sido comparadas con respecto a otros algoritmos clásicos de la literatura. En todas las pruebas realizadas, el rendimiento de nuestras propuestas, con respecto al resto, ha mejorado a medida que el sistema crecía en número de mercados y nodos. Estos algoritmos han sido presentados y evaluados en la siguiente publicación:

[CBGS11] D. Castellà, H. Blanco, F. Giné and F. Solsona. *A Computing Resource Discovery Mechanism over a P2P Topology.* 9th International Conference on High Performance Computing for Computational Science (VECPAR 2010), LNCS (Springer), volume 6449, pages 366-379, 2010.

Análisis de localidad de una red de computación P2P: La búsqueda de recursos aproximados, que no tiene sentido en sistemas P2P tradicionales de compartición de ficheros, cobra todo el sentido en entornos de cómputo. Este hecho comporta que un sistema de cómputo con calidad

de servicio, como es el caso de *DisCoP*, disponga de esta funcionalidad. La búsqueda eficiente de recursos aproximados implica que los recursos similares pertenecientes a distintos mercados, estén próximos entre si. Esta característica, conocida como *localidad* de recursos, ha sido estudiada ampliamente en este trabajo, de modo que se ha propuesto un nuevo procedimiento de cálculo de la localidad, proceso que es aplicable a cualquier overlay P2P. Este procedimiento ha sido aplicado para evaluar la localidad de *DisCoP* con respecto a otros overlays, como es el caso de Hilbert aislado o bien la unión de Hilbert y Chord. Asimismo, este procedimiento ha sido aplicado para evaluar las prestaciones de los enlaces Hilbert añadidos a la arquitectura DisCoP. Este trabajo ha sido recientemente enviado a evaluar:

Aceptado D. Castellà, F. Giné and F. Solsona. *An Efficient Locality P2P Computing Architecture.* CMMSE, Junio 2011. Enviado y en revisión al Journal of Supercomputing.

5.2. Líneas abiertas

En base a la experiencia obtenida en el desarrollo de esta tesis, han ido surgiendo nuevas líneas de investigación que pueden contribuir a completar el nivel de refinamiento y la amplitud del problema abordado en un entorno de computación P2P con calidad de servicio:

- Con los años, los recursos computacionales van aumentado sus capacidades. Partiendo de una arquitectura donde la función Hilbert distribuye los nodos según sus capacidades computacionales, el entorno DisCoP ha de saber adaptarse dinámicamente a la evolución tecnológica, creando políticas de redistribución de los nodos entre mercados según sus nuevas capacidades de cómputo, el flujo de entrada de peers, etc.

- Realizar un estudio comparativo más amplio sobre el rendimiento de búsqueda, tolerancia a fallos y costes de las funcionalidades desarrolladas en esta tesis, aplicado sobre otras topologías P2P populares, como son CAN, Pastry, Tapestry, Kadmelia y Viceroy.

- Implementar completamente el sistema *DisCoP* y realizar pruebas experimentales de funcionamiento, rendimiento y de fiabilidad, en un entorno real. Algunas de estas pruebas consistirían en evaluar los puntos débiles del sistema cuando se produzcan entradas y salidas masivas de peers.

- Diseñar un algoritmo de planificación de tareas distribuidas sobre el sistema DisCoP. Esto comportaría el diseño de un protocolo de comunicación, sincronización con los nodos implicados para la ejecución, el envío de las tareas implicadas en la ejecución y la recepción de los resultados finales.

- Implementar mecanismos de replicación de cómputo para aumentar la QoS ante fallos en los nodos. La idea es extender las técnicas de replicación ampliamente utilizadas en entornos multicluster hacia sistemas P2P, atendiendo a sus características específicas.

- Realizar mecanismos que permitan almacenar resultados parciales de ejecuciones en los peers, denominado "caching" de cómputo, con el objetivo de mejorar la fiabilidad y disponibilidad del sistema, incrementando las garantías de QoS.

- Definir mecanismos y políticas de reserva de recursos. Los sistemas de reserva garantizan un cierto grado de exclusividad de los recursos de cálculo, hecho que facilita la consecución de los requerimientos de QoS.

- Desarrollar mecanismos de contabilidad e incentivación para la compartición de recursos. Los sistemas P2P se basan en incentivar la cesión y/o compartición de recursos de cómputo y es por ello necesario establecer criterios que primen los nodos con un mayor grado de participación en el entorno.

Bibliografía

[ABJM04] Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. Large-scale deployment in P2P experiments using the JXTA distributed framework. In *Euro-Par 2004. 10th International Conference on Parallel Processing*, number 3149 in LNCS, pages 1038–1047. Springer-Verlag, 2004.

[ADT04] N. A. Al-Dmour and W. J. Teahan. ParCop: A decentralized peer-to-peer computing system. *International Symposium on Parallel and Distributed Computing*, 0:162–168, 2004.

[AES97] Anurag Acharya, Guy Edjlali, and Joel H. Saltz. The utility of exploiting idle workstations for parallel computation. In *SIGMETRICS'97*, pages 225–236, 1997.

[AJN05] Gabriel Antoniu, Mathieu Jan, and David Noblet. Enabling the p2p jxta platform for high-performance networking grid infrastructures. In *Proc. of the first Intl. Conf. on High Performance Computing and Communications (HPCC '05)*, number 3726 in Lect. Notes in Comp. Science, pages 429–439, Sorrento, Italy, September 2005. Springer-Verlag.

[Arb] Árbol recubridor mínimo. `http://es.wikipedia.org/wiki/Arbol_recubridor_minimo`.

[Are] Ares galaxy. `http://sourceforge.net/projects/aresgalaxy/`.

[AS07] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, November 2007.

[Aud] Audiogalaxy. `http://www.audiogalaxy.com/`.

[AX02] A. Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. In *Second International Conference on Peer-to-Peer Computing (P2P 2002).*, pages 33–40, 2002.

[BAS04] Ashwin Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury:supporting scalable multiattribute range queries. In *Special Interest Group on Data Communication (SIGCOMM'04)*, volume 34, pages 353–366, October 2004.

[BCG05] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 651–660. ACM, 2005.

[BDET00] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *SIGMETRICS Perform. Eval. Rev.*, 28:34–43, June 2000.

[BGKS02] Daniel Brookshier, Darren Govoni, Navaneeth Krishnan, and Juan Carlos Soto. JXTA: Java P2P programing. Sams, 2002.

[Bit] Bittorrent. `http://www.bittorrent.com/`.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, 1996.

[BM02] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience (CCPE)*, 14(13):1175–1220, 2002.

[BVV03] Ranjita Bhagwan, George Varghese, and Geoffrey M. Voelker. Cone: Augmenting dhts to support distributed resource discovery. Technical report, Microsoft Research, 2003.

[CBGS10] Damia Castellà, Hector Blanco, Francesc Giné, and Francesc Solsona. Combining Hilbert SFC and Bruijn graphs for searching computing markets in a P2P system. In *Proceedings of the 16th international Euro-Par conference on Parallel processing*, EuroPar'10, pages 471–483. Springer-Verlag, 2010.

[CBGS11] Damia Castellà, Hector Blanco, Francesc Giné, and Francesc Solsona. A computing resource discovery mechanism over a P2P tree topology. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pages 366–379. Springer-Verlag, 2011.

[CBR+08] D. Castellà, I. Barri, J. Rius, F. Giné, F. Solsona, and F. Guirado. Codip2p: A peer-to-peer architecture for sharing computing resources. In *International Symposium on Distributed Computing and Artificial Intelligence, DCAI 2008*, volume 50, pages 293–303. Springer, 2008.

[CJ07] Xinuo Chen and Stephen A. Jarvis. Design and implementation of efficient range query over dht services. In *Proceedings of the 1st International Conference on Signal Processing and Communication Systems*, 2007.

[CLGS04] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, WebDB '04, pages 25–30, 2004.

[CLM+07] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: an efficient and robust P2P range index structure. In *Proceedings of the*

2007 ACM SIGMOD international conference on Management of data, SIGMOD '07, pages 223–234, 2007.

[CMM02] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving DNS using Chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[CRB+08] D. Castellà, J. Rius, I. Barri, A. Guim, M. Orobitg, H. Blanco, and F. Giné. CoDiP2P: a distributed computing architecture oriented to P2P environments. In *XIX Jornadas de Paralelismo*, 2008.

[CRB+09] D. Castellà, J. Rius, I. Barri, F. Giné, and F. Solsona. CoDiP2P: a new P2P architecture for distributed computing. In *Conference on Parallel, Distributed and Network-based Processing (PDP 2009)*, pages 323–329, 2009.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–66, 2001.

[DHA03] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[DHJ+05] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie-structured overlays. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 57–66, 2005.

[dis] Distributed.net. `http://www.distributed.net/`.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 202–215, 2001.

[eDo] edonkey2000 home page. `http://www.eDonkey2000.com/`.

[EH85] A. Esfahanian and L. Hakimi. Fault-tolerant routing in DeBruijn communication networks. *IEEE Transaction on Computers*, 34:777–788, 1985.

[Emu] Emule. `http://www.emule-project.net/home/`.

[FI03] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, pages 118–128, 2003.

[GAA02] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Conference on Innovative Data Systems Research (CIDR)*, 2002.

[Gnu] Gnutella2. `http://g2.trillinux.org/`.

[Gro] Groove. `http://www.groove.net`.

[Gru96] Grupo de trabajo P2P de internet2. `http://p2p.internet2.edu/`, 1996.

[GS04] Jun Gao and Peter Steenkiste. An adaptive protocol for efficient support of range queries in DHT-based systems. In *Proceedings of the 12th IEEE International Conference on Network Protocols*, pages 239–250, 2004.

[GSS06] Rohit Gupta, Varun Sekhri, and Arun K. Somani. CompuP2P: An architecture for internet computing using Peer-to-Peer networks. *IEEE Transactions Parallel Distributed Systems*, 17(11):1306–1320, 2006.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *International Conference on Management of Data*, pages 47–57. ACM, 1984.

[HBMS04] Oliver Heckmann, Axel Bock, Andreas Mauthe, and Ralf Steinmetz. The edonkey file-sharing network. In *Proceedings of the Workshop on Algorithms and Protocols for Efficient PeertoPeer Applications*, pages 2–6, 2004.

[HKO05] Benoit Hudzia, M. Tahar Kechadi, and A. Ottewill. TreeP: A tree-based P2P network architecture. In *IEEE International Conference on Cluster Computing*, pages 1–15, 2005.

[HLP+04] Tobias Hoßfeld, Kenji Leibnitz, Rastin Pries, Kurt Tutschku, Phuoc Tran-Gia, and Krzysztof Pawlikowski. Information diffusion in edonkey filesharing networks. *Research Report Series*, (341), September 2004.

[IRD02] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 213–222, 2002.

[Jag06] H. V. Jagadish. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, page 34, 2006.

[JdSJ10] Luciano Jose, Senger Marcio Augusto de Souza, and Dierone Cesar Foltran Jr. Towards a peer-to-peer framework for parallel and distributed computing. *Computer Architecture and High Performance Computing, Symposium on*, 0:127–134, 2010.

[jJ06] Yuh jzer Joung. KISS: A simple prefix search scheme in P2P networks. In *WebDB*, pages 56–61, 2006.

[JOV05] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Very Large Data Base Endowment (VLDB)*, pages 661–672, 2005.

[JYF07] Yuh-Jzer Joung, Li-Wei Yang, and Chien-Tse Fang. Keyword search in dht-based peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 25:46–61, 2007.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 190–201, 2000.

[KC04] S. H. Kwok and K. Y. Chan. An enhanced gnutella p2p protocol: A search perspective. In *International Conference on Advanced Information Networking and Applications*, 2004.

[KK03] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *International Conference on Peer-to-Peer Systems*, pages 98–107, 2003.

[kPC02] Wang kee Poon and Jiannong Cao. Rheeve: A plug-n-play peer-to-peer computing platform. In *International Conference on Distributed Computing Systems Workshops*, page 706, 2002.

[Kru] Algoritmo de kruskal. `http://es.wikipedia.org/wiki/Algoritmo_de_Kruskal`.

[LCP+05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[LKRG03] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In *2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 395–406. ACM Press, 2003.

[LLW+06] Dongsheng Li, Xicheng Lu, Baosheng Wang, Jinshu Su, Jiannong Cao, Keith C. C. Chan, and Hong va Leong. Delay-bounded range queries in dht-based peer-to-peer systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, pages 64–, 2006.

[Lot] Lotus notes. `http://www-142.ibm.com/software/products/es/es/notes/`.

[LPh] Lphant. `http://www.lphant.com/`.

[LSA96] Zhen Liu, Ting-Yi Sung, and Centre Sophia Antipolis. Routing and transmitting problems in de bruijn networks. *IEEE Transactions on Computers*, 45(9):1056–1062, 1996.

[MA07] J. Mishra and S. Ahuja. P2pcompute: A peer-to-peer computing system. In *International Symposium on Collaborative Technologies and Systems*, pages 169–176, 2007.

[MAK03] Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. Analysis of multi-dimensional space-filling curves. *Geoinformatica*, 7(3):179–209, 2003.

[MD5] Md5: Message-digest algorithm 5. `http://es.wikipedia.org/wiki/MD5`.

[Mes] Msn messenger. `http://www.msn.com`.

[MH98] R. McNab and F.W. Howell. Simjava. 1998.

[MIT] Massachusetts institute of technology. `http://web.mit.edu/`.

[MJFS96] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13:2001, 1996.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin / Heidelberg, 2002.

[MNR02] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, August 2002.

[Nap] Napster. `http://en.wikipedia.org/wiki/Napster_(peer-to-peer)`.

[Net] Net2phone. `http://www.net2phone.com/`.

[NMVL05] Ngoc Chi Nguyen, Nhat Minh, Dinh Vo, and Sungyoung Lee. Fault free shortest path routing on the de bruijn networks. *Lecture notes in computer science*, 3421:327–334, 2005.

[NNL04] Ngoc Chi Nguyen, Vo Dinh Minh Nhat, and Sungyoung Lee. Efficient routing and broadcasting algorithms in de bruijn networks. In *ISPA*, pages 677–687, 2004.

[NSA] National security agency. `http://www.nsa.gov/`.

[NVW+05] Kotilainen N., M. Vapa, M. Weber, J. Toyryla, and J. Vuori. P2pdisco - java distributed computing for workstations using chedar peer-to-peer middleware. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[Pee] Peeriobiz y peeriodata. `http://www.peerio.com`.

[Pri] Algoritmo de prim. `http://es.wikipedia.org/wiki/Algoritmo_de_Prim`.

[pro] El problema del viajante. `http://es.wikipedia.org/wiki/Problema_del_viajante`.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 311–320, 1997.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[RKCD01] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 30–43. Springer-Verlag, 2001.

[RWE+01] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5:40–49, September 2001.

[SC08] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE In Signal Processing Magazine*, 25(2):128–131, 2008.

[SET99] Seti@home: Search for extraterrestrial intelligence at home. `http://setiathome.ssl.berkeley.edu`, May 1999.

[SGAA04] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. *Data Engineering, International Conference on*, 0:165, 2004.

[SHA] Sha1 secure hash algorithm 1. `http://es.wikipedia.org/wiki/Sha1`.

[Sky] Skype. `http://www.skype.com/intl/es/home/`.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160. SIGCOMM 2001, August 2001.

[SP89] M.R. Samatham and D.K. Pradhan. The de bruijn multiprocessor network: A versatile parallel processing and sorting network for vlsi. *IEEE Transactions on Computers*, 38:567–581, 1989.

[SP03] Cristina Schmidt and Manish Parashar. Flexible information discovery in decentralized distributed systems. In *in Proceedings of the 12th High Performance Distributed Computing (HPDC)*, pages 226–235, 2003.

[SP08] Cristina Schmidt and Manish Parashar. Squid: Enabling search in dht-based systems. *J. Parallel Distrib. Comput.*, 68:962–975, July 2008.

[STS02] C. A. Stein, Michael J. Tucker, and Margo I. Seltzer. Building a reliable mutable file system on peer-to-peer storage. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, 2002.

[Tej06] Ramon J. Millán Tejedor. *Domine Las Redes P2P*. Creaciones Copyright, S.L., creaciones copyright, s.l. edition, 2006.

[WRC00] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, 2000.

[ZF] Niklas Zennstrom and Janus Friis. Kazaa. `http://www.kazaa.com/`.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[ZSLS06] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over dht. In *In Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, 2006.