UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

DESCUBRIMIENTO DE SERVICIOS TOLERANTE A FALLOS BASADO EN HIPERCUBOS PARA SISTEMAS DISTRIBUIDOS DE GRAN ESCALA

ANTONIA GALLARDO GÓMEZ
Ingeniera Técnico Superior en Telecomunicaciones

DIRECTOR
Dr. LUIS M. DÍAZ DE CERIO RIPALDA

Memoria presentada en cumplimiento parcial de los requisitos para el grado de Doctor en el Departamento de Arquitectura de Computadores de la Universitat Politècnica de Catalunya (UPC) – BarcelonaTech
Julio 2013

DESCUBRIMIENTO DE SERVICIOS TOLERANTE A FALLOS BASADO EN HIPERCUBOS PARA SISTEMAS DISTRIBUIDOS DE GRAN ESCALA

ANTONIA GALLARDO GOMEZ

Memoria presentada en cumplimiento parcial de los requisitos para el grado de Doctor en el Departamento de Arquitectura de Computadores de la Universitat Politècnica de Catalunya (UPC) – BarcelonaTech

Julio 2013

Dedicada a mi hijo David
y a mi madre.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE PUBLICACIONES

Se listan las publicaciones dividiéndolas en artículos en revistas y presentados en congresos y en orden cronológico inverso.

I. ARTÍCULOS EN REVISTAS

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Roc Messeguer Pallarès; Andrreu P. Isern Deyà; Kana Sanjeevan. GRID Resource Searching on the GridSim Simulator. *Lecture Notes in Computer Science*. 5544,pp. 357- 366. Springer-Verlag, 05/2009 . ISSN 0302-9743.
Fuente de impacto: SCOPUS (SJR) – Índice de impacto: 0.572

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan. HGRID: A Self-configuring Grid Resource Discovery. *Journal of Computing and Information Technology*. 16 - 4, pp. 333 - 338. University Computing Centre of Zagreb, 12/2008. ISSN 1330-1136.

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan.Self-configuring Resource Discovery on a Hypercube Grid Overlay. *Lecture Notes in Computer Science*. 5165,pp. 510 - 519.Springer-Verlag, 08/2008. ISSN 0302-9743.
Fuente de impacto: SCOPUS (SJR) – Índice de impacto: 0.493

II. ARTÍCULOS PRESENTADOS EN CONGRESOS

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Roc Messeguer Pallarès; Andreu P. Isern Deyà; Kana Sanjeevan. GRID Resource Searching on the GridSim Simulator. *9th International Conference on Computational Science* (mayo del 2009), pp. 357 - 366. ISBN 978-3-642-01969-2.
Fuente de impacto: ERA 2010 – Rango de impacto: A

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan. HGRID: Fault Tolerant, Log2N Resource Management for Grids. *5th International Conference on Networking and Services* (abril 2009), pp. 511 - 517. ISBN 978-0-7695-3586-9.

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan. Self-configuring Resource Discovery on a Hypercube Grid Overlay. *14th International Euro-Par Conference* (agosto 2008), pp. 510 - 519. ISBN 978-3-540-85450-0.
Fuente de impacto: ERA 2010 – Rango de impacto: A

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan. HGRID: A Self-configuring Grid Resource Discovery. *30th International Conference on Information Technology Interfaces* (junio 2008), pp. 833 - 838. ISBN 978-953-7138-12-7.
Fuente de impacto: ERA 2010 – Rango de impacto: C

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan. HGRID: An Adaptive Grid Resource Discovery. *2nd International Conference on Complex, Intelligent and Software Intensive Systems* (marzo 2008), pp. 411 - 416. ISBN 978-0-7695-3109-0.

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan; Luis C. E. Bona. Scalable Self-Configuring Resource Discovery for Grids, *V Brazilian Workshop on Grid Computing and Applications* (junio 2007), pp. 13 - 22. ISBN 85-766-9118-3.

Antonia Gallardo Gómez; Luis Díaz de Cerio Ripalda; Kana Sanjeevan. A Fault Tolerant, Log2N Resource Management Scheme for Grids. *XIV Jornadas de Concurrencia y Sistemas Distribuidos* (junio 2006), pp. 145 - 155.ISBN 84-689-9292-5

AGRADECIMIENTOS

En estas líneas quiero agradecer a mi director Luis, a mi compañero de despacho y amigo Sanji y a mi ponente Leandro, su ayuda y su paciencia durante todo el tiempo que ha durado esta tesis.

También quiero agradecer a Miguel, a Cristina, a Esther y a Angélica que me recordaran que tenía este trabajo pendiente (tengo tendencia a dispersarme). Muchas gracias.

No quiero olvidar tampoco a Roc ya que con él empecé un trabajo final de carrera que sin su ayuda no hubiera finalizado y al que hago referencia en este documento.

También quiero recordar a mis padres que siempre trabajaron para que yo llegara hasta aquí y que aguantaron mi mal humor cuando no lograba mis objetivos. Mi hermana también estuvo ahí.

Finalmente, agradezco a la persona más importante de mi vida, a mi hijo David, su ayuda y todas las horas que no ha podido jugar conmigo en el último año (mientras yo acababa este trabajo), y al que quiero muchísimo.

ABSTRACT

Current distributed systems that are capable of sharing resources in a distributed manner are experiencing an increase in their use and scope. This increased use in various application areas is largely due to the low cost of deployment and maintenance of a distributed environment, when compared to architectures like supercomputers. A supercomputer consists of thousands of processing units (CPUs) and thousands of gigabytes (GBs) of RAM, which work together to process complex operations and / or expensive simulations of physics, medical research, weather, etc. Supercomputers are located in a single room, where all processors and memories are linked by high speed networks and are within walking distance. This clearly differentiates them from distributed systems that are geographically distributed environments linked (typically) by the Internet.

Many research efforts today are focused on providing solutions for distributed systems. These include for example the work of European Grid Infrastructure (EGI) [1] which aims to create an e-Science infrastructure that makes use of a variety of distributed computing technologies and will strengthen Europe's position in the area of distributed computing infrastructures.

It is essential that the solutions proposed allow for the sharing of a very large number of resources. This is a complex challenge to tackle since we do not want the solution to degrade as the number of system resources increase.

It is also imperative that the solutions proposed, adapt to failures that occur both in the network and in the hardware and / or software that form the distributed system. This does not imply that the components should be universally available, but that the solution should be able to adapt to obtain the highest possible performance from all the components available at a given time. This is a more complex challenge.

This thesis proposes a Resource Discovery Service (of services or resources) for large-scale distributed systems that are able to adapt to failures that occur in the system. By large scale systems, we mean systems that may be made up of hundreds, thousands or millions of machines. Failures refer to components that are not accessible through the network, components that are not working, or are overloaded, etc. It should be mentioned that in this work we found that in a real environment of geographically distributed resources, it is essential for service discovery to be fault tolerant. In one of the evaluations of our proposed discovery service, we chose 150 machines distributed geographically throughout Europe at random, without knowing if they were in a state of failure or not. We found that 24'67% of them were unavailable because they were in a failed state or because they failed during the evaluation of our service discovery.

Examples of distributed systems where you can apply the discovery service that is presented in this thesis are Grid environments. A Grid environment [2] integrates communication networks and machines to provide a virtual platform for computing and data management to users. A user accesses the Grid and utilizes its resources (individual computers, data files, etc.). Grid environments allow users to interact with the resources in a uniform way, providing a

powerful platform. Another way to define a Grid is to compare it with the electrical grid [3]. For example, when you submit a job to the Grid environment or store data in it, it is not known exactly which resource in the Grid is used - just as when an electrical device is connected, we do not know from where the energy consumed is generated. A user of a Grid only needs an access device and a network connection while an electrical network user only needs a plug that is connected to the mains. Thus, Grid environments are transforming society, considerably increasing computing power and distributed storage by combining individual resources that by themselves have no significant power or storage capacity. An efficient resource discovery service becomes essential in such environments.

Other examples of distributed systems where the Resource Discovery Service presented in this thesis can be applied are Cloud environments [4], since such environments are large-scale distributed systems. Companies like Amazon, Google or Microsoft that own innumerable number of machines are provisioning commercial Cloud environments and offering computing (Amazon EC2), data storage (Amazon S3), web applications hosting (Google App Engine) , collaboration through online versions of office applications (Microsoft Office 365), etc.. In these environments too, service discovery is critical.

In this thesis, the proposed discovery service is based on an overlay that has a hypercube topology that interconnects nodes / intermediaries (brokers). The term overlay is used to describe a virtual network constructed at the application layer, above the level of TCP / IP. It acts as an intermediary component that mediates between service consumers (or clients) and service providers (or servers).

Service discovery can be seen as a process that consists of 4 phases: *bootstrapping*, where customers and service providers establish the first contact with the system; *service announcement*, where a service provider publishes information about the services provided; *search*, where a client seeks one or more specific services; and *selection*, the last phase of service discovery where the service to be utilized by the user is selected from those available.

This thesis focuses on the search phase. An example could be a client trying to locate clusters with 8 CPUs, 1 GB of memory with CPU usage below 50%; where the clusters must have given software installed, such ATLAS-6.0.4; and is connected to a database containing the data of a particular experiment. Once the clusters that meet the above requirements are localized, the last phase of resource discovery is the selection of the cluster that is to be used by the client.

The difficulties of this type of search lies, among others, in that resource discovery systems based on centralized topologies that are not fault tolerant degrade when the number of shared resources increase. Hence, if a server fails, the search fails and the customer cannot use the service. For these reasons, the discovery service presented is not based on a centralized topology. Furthermore, there is no component of the system that has information about the entirety of shared resources within which a client could locate a needed resource. This poses the complex challenge of ensuring that if the resource searched in a system is available, then the client must be able to locate it.

Like other authors [5], we distinguish between three aspects of fault tolerance in an overlay. An overlay is a virtual network that connects nodes. The first aspect is data replication, a

technique of having multiple copies of the same data in different nodes. The data will be lost only if the nodes fail simultaneously. The second aspect is the maintenance of the overlay. This requires the application of some technique or techniques to reconfigure the overlay, removing from it the failed nodes. Since the reconfiguration of the overlay is not immediate, the system must continue to serve the demands of their customers while reconfiguring the overlay, the third aspect has to do with the static resilience of the algorithm that runs on the overlay. Static resilience is the ability of the algorithm running on the overlay to continue serving customers before failover algorithms run.

Of the three aspects above, this paper will not focus on data replication or maintenance of the overlay. The main contribution of this thesis will focus on presenting a search algorithm with high static flexibility. In order to evaluate static flexibility, we consider static nodes identified to be in a failed state in the overlay and assume that the reconfiguration techniques for the overlay have still not been executed – i.e., the nodes in a failed state still belong to the overlay. Searches are performed on this static scenario and evaluated to see if resources (services) can be located efficiently under these conditions.

The results obtained for static flexibility are very encouraging. For example, in a distributed system consisting of about 1,000,000 nodes, if we consider that the required resources present in the overlay are uniformly distributed and that 30% of the nodes fail in a uniformly distributed manner, the algorithm presented in this thesis is able to locate about 94% of them. This is in contrast to the algorithm proposed by HaoRen et al. [6] (described later in this thesis) that locates only 4% of the available resources. Furthermore, if we apply the algorithm in HyperD [7] to the example shown in their article [8] (also described later in this thesis) and compare it to the results from applying the algorithm presented in this thesis, we are able to locate 100% of the resources present compared to 55.5% in their case.

The algorithms that have been compared with the one proposed in this thesis are algorithms for large-scale distributed systems that are structured as hypercube overlay topologies. Specific to the case of HaoRen et al., after the review of a publication, it has been necessary to assert that comparatively, the algorithm proposed in this paper is not the same as the algorithm proposed by them. Furthermore, our proposal has significantly enhanced static resilience.

It is important to emphasize that a resource discovery system formed by an overlay along with a search algorithm that has less static resilience than another, should be able to reconfigure itself faster in order to be able to be equivalently fault tolerant. The high values of static resilience that have been realized in this thesis make the costly process of immediate reconfiguration of failed nodes in a hypercube unnecessary for the correct functioning of the resource discovery process. The reason is that although the hypercube has not been reconfigured, the resource discovery system continues to serve the requests of customers efficiently.

To conclude this section, the quality of the algorithm proposed in this thesis is first evaluated by measuring whether when applied to large-scale distributed systems it significantly degrades it, and secondly by measuring its static resilience and comparing the results obtained with algorithms proposed by other authors and thirdly by evaluating in terms of search times.

Also, search times are obtained by emulation of systems formed by about a thousand nodes. As a final evaluation, we implemented a system consisting of 150 machines distributed geographically throughout Europe that were running user processes that were beyond our control. Thus we were able run experiments (searches) under real traffic conditions. The machines were chosen randomly without knowing if they were in a faulty state or not. Of the chosen machines, 24'67% of them was found to be in a failed state - because they were in error or because they failed during searches. This clearly shows that the algorithms to be used in large-scale distributed systems must be fault tolerant.

The main contributions of this thesis are:

- A decentralized architecture that has partial information of system failures, named HOverlay that can efficiently search for services in large-scale distributed systems (orders of magnitude of hundreds, thousands and millions of machines).
- Two search algorithms (prior to the final proposal), called Algorithm-vd and Algorithm-va for large-scale systems that adapt to failures that inevitably occur in any system.
- A search algorithm (the final proposal) called Algorithm-taux, also for large-scale systems that adapts to the inevitable failure of any system and one that learns from previous searches. This enables future searches to better adapt to failures with Algorithm-va and Algorithm-vd, for example when there is a network partition, i.e., when all the network connections between two groups of systems fail simultaneously. In order to evaluate Algorithm-taux and compare it to other algorithms we have implemented:
    - A simulator that can evaluate a distributed system with static resilience and distributed algorithms that run on the system based on a hypercube topology that can have up to millions of nodes. This high number of nodes has required simulations with large cost in time. Using this prototype, when the system malfunctions, Algorithm-taux adapts even to 90% more.
    - A simulator that allows us to emulate a distributed system, measuring search times and effectiveness. Using this prototype to simulate 30% of evenly distributed failed nodes and the scarcity of services sought and present in the system, Algorithm-taux was able to find at least one service that satisfied its search criteria in 100% of the simulations performed.
    - A prototype to measure search times and effectiveness in a real system that is geographically distributed and based on a hypercube topology. The prototype has been tested on a set of 150 machines distributed throughout Europe.

I. INTRODUCCIÓN

1.1 Resumen de la tesis

Actualmente los sistemas distribuidos, capaces de compartir de forma distribuida recursos están experimentando un incremento en su uso y en sus ámbitos de aplicación. Por ejemplo, ampliando su aplicación a redes wireless de sensores para obtener datos de los sensores y compartir los recursos computacionales y de almacenamiento de éstos, como un único sensor virtual. Este aumento de uso y de ámbitos de aplicación es, en buena medida, debido al bajo coste que supone el despliegue y mantenimiento de un entorno distribuido, si se compara con arquitecturas como los supercomputadores. Un supercomputador es un superordenador formado por miles de unidades de procesamiento (CPUs) y miles de gigabytes (GBs) de memoria RAM, que trabajan conjuntamente para procesar operaciones complejas y/o simulaciones costosas de procesos físicos, investigaciones médicas, fenómenos meteorológicos, etc. Los supercomputadores están ubicados en una única sala, donde todos los procesadores y memorias están unidos mediante redes de alta velocidad y de poca distancia, lo que les diferencia claramente de los sistemas distribuidos que son entornos distribuidos geográficamente unidos por Internet (típicamente).

Así, múltiples esfuerzos de investigación se centran hoy en día en aportar soluciones para sistemas distribuidos. Cabe citar por ejemplo el trabajo realizado por European Grid Infraestructure (EGI) [1] cuyo objetivo es crear una infraestructura e-Science que haga uso de una gran variedad de tecnologías de computación distribuida que permita fortalecer la posición europea en el área de las infraestructuras computacionales distribuidas.

Es esencial que las soluciones aportadas permitan compartir un número muy alto de recursos. Este es un reto complejo de abordar ya que plantea el problema de que la solución no se degrade a medida que el número de recursos del sistema aumente.

También es imprescindible que las soluciones aportadas se adapten a los fallos que se producen tanto en la red como en los componentes de hardware y/o software que forman el sistema distribuido. Esto no implica que los componentes deban estar universalmente disponibles, sino que la solución debe adaptarse para obtener el rendimiento más alto posible de los componentes disponibles en cada momento. Este es un reto aún más complejo.

El trabajo de esta tesis propone un servicio de descubrimiento de servicios (o recursos) para sistemas distribuidos de gran escala, capaz de adaptarse a los fallos que se producen en el sistema. Por *gran escala*, se entiende sistemas formados por cientos, miles o millones de máquinas y por *fallos*, se entiende componentes que no son accesibles a través de la red, componentes que no están funcionando, que están sobrecargados, etc. Cabe comentar que en este trabajo se ha constatado que en un entorno real distribuido geográficamente es imprescindible que el servicio de descubrimiento se adapte a los fallos que se producen en el sistema. En una de las evaluaciones del servicio de descubrimiento propuesto escogiendo 150 máquinas distribuidas geográficamente a lo largo de Europa de forma aleatoria sin conocer si se encontraban en estado de fallo o no, un 24,67% de ellas fallaron, bien porque ya estaban en fallo o bien porque fallaron durante la evaluación del servicio de descubrimiento.

Ejemplos de sistemas distribuidos donde puede aplicarse el servicio de descubrimiento presentado en este trabajo son los entornos de Grid. Un entorno de Grid [2] integra redes de comunicación y máquinas para proporcionar una plataforma virtual para la computación y la gestión de datos a los usuarios. Un usuario accede al Grid y utiliza recursos (ordenadores individuales, archivos de datos, etc.). El entorno de Grid permite a los usuarios interaccionar con los recursos de una forma uniforme, proporcionando una plataforma potente. Otra forma de definir un Grid es compararlo con la red eléctrica [3] ya que, por ejemplo, cuando se envía una tarea al entorno de Grid o se almacenan datos no se conoce exactamente qué recurso se ha utilizado, al igual que cuando se conecta un aparato eléctrico no se sabe de dónde procede ni dónde se ha generado la energía que consume. Un usuario de un Grid sólo necesita un dispositivo de acceso al Grid y una conexión de red, de la misma manera que un usuario de la red eléctrica sólo necesita un enchufe y que el enchufe esté conectado a la red eléctrica. Así, los entornos de Grid están transformando la sociedad, ya que aumentan de forma significativa la potencia de cálculo y de almacenamiento distribuido, combinando recursos individuales que, por sí mismos, no tienen una potencia ni una capacidad de almacenamiento significativa. El servicio de descubrimiento de recursos es fundamental en esos entornos.

Otros ejemplos de sistemas distribuidos donde también puede aplicase el sistema de descubrimiento presentado en este trabajo son los entornos de Cloud [4], ya que dichos entornos son sistemas distribuidos de gran escala. Empresas como Amazon, Google o Microsoft propietarias de cientos o miles de máquinas están apostando comercialmente por estos entornos ofreciendo cómputo (EC2 de Amazon), almacenamiento de datos (S3 de Amazon), ejecutar aplicaciones web en su infraestructura (Google App Engine de Google), colaboración a través versiones online de aplicaciones ofimáticas (Microsoft Office 365 de Microsoft), etc. También en estos entornos el servicio de descubrimiento es fundamental.

En este trabajo el servicio de descubrimiento propuesto está basado en una red overlay con la topología de un hipercubo que interconecta nodos/intermediarios (*brokers*). El término overlay se usa como red virtual construida en la capa de aplicación, por encima del nivel de TCP/IP, y se entiende un intermediario como un componente que media entre consumidores de servicios (o clientes) y proveedores de servicios (o servidores).

Podemos ver el descubrimiento de servicios como un proceso que consta de 4 fases: *bootstrapping*, donde los clientes y los proveedores de servicios establecen el primer contacto con el sistema; *anuncio de servicios*, donde un proveedor de servicios publica información sobre los servicios que proporciona; *búsqueda*, donde un cliente trata de encontrar uno o varios servicios determinados; y *selección*, la última fase del descubrimiento de servicios, donde se selecciona entre los servicios obtenidos aquel que utilizará el cliente.

La presente tesis se centra en la fase de búsqueda. Un ejemplo de búsqueda puede ser localizar desde un cliente clusters con 8 CPUs, 1 GB de memoria y un uso de CPUs inferior al 50%; donde los clusters deben tener instalado un software determinado, por ejemplo ATLAS-6.0.4 y estar conectados a una base de datos que contiene los datos de un cierto experimento. Con los clusters localizados que satisfagan los requerimientos anteriores, en la última fase del descubrimiento de recursos se selecciona el cluster que utilizará el cliente.

Las dificultades de este tipo de búsquedas radica, entre otras, en que los sistemas de descubrimiento basados en topologías centralizadas se degradan al aumentar el número de recursos compartidos y en que no se adaptan a los fallos, ya que si falla el servidor fallan las búsquedas y el cliente finalmente no puede utilizar el servicio. Por estos motivos el servicio de descubrimiento que se presenta no está basado en una topología centralizada y por tanto no hay ningún componente del sistema con información de todos los recursos compartidos en el que pueda localizar un cliente el recurso que busca. Esto plantea el reto complejo de asegurar que si el recurso buscado se encuentra disponible en el sistema el cliente pueda localizarlo.

Como otros autores [5] se distingue entre tres aspectos de la tolerancia a fallos en una overlay. Una overlay es una red virtual que conecta nodos. El primer aspecto es la replicación de datos, técnica que consiste en tener varias copias de los mismos datos en diferentes nodos que deben fallar simultáneamente para que los datos se pierdan. El segundo aspecto es el mantenimiento de la overlay que hace que se deba aplicar alguna técnica o técnicas para reconfigurar la overlay, eliminando de ella los nodos en fallo. Dado que la reconfiguración de la overlay no es inmediata y que por tanto, el sistema debe continuar sirviendo las peticiones de sus clientes mientras se reconfigura la overlay, el tercer aspecto es la flexibilidad estática (*static resilience*) del algoritmo que se ejecuta sobre la overlay. La flexibilidad estática es la capacidad que tiene el algoritmo que se ejecuta sobre la overlay de continuar sirviendo a los clientes antes de que los algoritmos de recuperación ante fallos se ejecuten.

De los tres aspectos anteriores este trabajo no se centrará en la replicación de datos ni en el mantenimiento de la overlay sino que la contribución de la presente tesis se centrará en presentar un algoritmo de búsqueda con elevada flexibilidad estática. Para evaluar la flexibilidad estática se consideran identificados los nodos en estado de fallo y que las técnicas de reconfiguración de la overlay aún no se han ejecutado, es decir, que los nodos en estado de fallo aún pertenecen a la overlay. Sobre este escenario estático se ejecutan búsquedas y se evalúa si se localizan servicios eficientemente en estas condiciones.

Los resultados de flexibilidad estática obtenidos son muy alentadores. Por ejemplo, en un sistema distribuido formado por alrededor de 1.000.000 de nodos, si consideramos que los recursos buscados y presentes en la overlay están uniformemente distribuidos y que un 30% de los nodos fallan de forma uniformemente distribuida, el algoritmo que se presenta en esta tesis obtiene alrededor de un 94% de los recursos existentes mientras que el algoritmo propuesto por HaoRen et al. [6] (que se describe más adelante en el documento) obtiene sólo un 4% En la misma línea, considerando el algoritmo que se aplica en HyperD [7] para el ejemplo mostrado en su artículo [8] (también descrito en el documento) aplicando el algoritmo presentado en esta tesis se obtiene el 100% de los recursos presentes frente a un 55,5%.

Los algoritmos con los que se ha comparado el algoritmo propuesto en esta tesis son algoritmos propuestos para sistemas distribuidos de gran escala y que están basados, como la propuesta de esta tesis, en una overlay con topología en hipercubo. Particularizando para el caso de HaoRen et al., después de la revisión de una publicación, ha sido necesaria la comparativa para constatar que el algoritmo propuesto en este documento no es el mismo que el algoritmo propuesto por HaoRen et al. y que además el que se propone aquí tiene mayor flexibilidad estática.

Es importante destacar que un sistema de descubrimiento formado por una overlay junto con su algoritmo de búsqueda con menor flexibilidad estática que otro debe ser más rápido reconfigurando la overlay para que ambos tengan una tolerancia en presencia de fallos similar. Los altos valores de flexibilidad estática que se han obtenido en esta tesis hacen que aunque reconfigurar eliminando los nodos en fallo en un hipercubo sea un proceso costoso no es necesario reconfigurar de forma inmediata el hipercubo para el correcto funcionamiento del sistema de descubrimiento. La razón es que aunque el hipercubo no esté reconfigurado el sistema de descubrimiento sigue sirviendo las peticiones de los clientes eficientemente.

Para finalizar este apartado, la calidad del algoritmo que se propone en esta tesis se ha evaluado primero midiendo si al aplicarlo en sistemas distribuidos de gran escala se degrada considerablemente el sistema, segundo midiendo su flexibilidad estática y comparando los resultados obtenidos con algoritmos propuestos por otros autores y tercero en términos de tiempos de búsqueda.

Además el tiempo de búsqueda se ha obtenido por emulación en sistemas formados por alrededor de mil nodos y en un entorno real distribuido geográficamente. Para la última evaluación se ha implementado un sistema, formado por 150 máquinas distribuidas geográficamente a lo largo de Europa donde se ejecutaban procesos de otros usuarios que no se controlaban y se han ejecutado búsquedas en condiciones de tráfico real. Las máquinas se escogieron de forma aleatoria sin conocer si se encontraban en estado de fallo o no y un 24,67% de ellas fallaron o bien porque ya estaban en fallo o porque fallaron durante las búsquedas, lo que constata el hecho de que los algoritmos a emplear en sistemas distribuidos de gran escala deben ser algoritmos tolerantes a fallos.

1.2 Contribuciones de la tesis

Las principales contribuciones de la presente tesis son las siguientes:

- Una arquitectura descentralizada con información parcial de los fallos del sistema, llamada HOverlay y que permite la búsqueda de servicios en sistemas distribuidos de gran escala (de órdenes de magnitud de cientos, millares y millones).
- Dos algoritmos de búsqueda (previos a la propuesta final), llamados Algoritmo-v_d y Algoritmo-$v_{a.}$ para sistemas de gran escala que se adaptan a los fallos que se producen irremediablemente en todo sistema.
- Un algoritmo de búsqueda (propuesta final) llamado Algoritmo-$t_{aux.}$ también para sistemas de gran escala que se adapta a los fallos irremediable de todo sistema y que aprende de las búsquedas realizadas anteriormente, lo que le permite, en búsquedas futuras poder adaptarse mejor a fallos que Algoritmo-v_d y Algoritmo-v_a, por ejemplo cuando se produce una partición de red (network partition), es decir, todas las conexiones de red entre dos grupos de sistemas fallan simultáneamente. Para evaluar Algoritmo-t_{aux} y compararlo con otros algoritmos se ha implementado:
    - Un simulador que permite evaluar en un sistema simulado distribuido la flexibilidad estática de algoritmos que se ejecutan sobre el sistema basado en una topología en hipercubo y que puede tener hasta millones de nodos. Este elevado número de nodos

ha requerido simulaciones costosas en tiempo. Utilizando este prototipo cuando en el sistema se producen fallos, Algoritmo-$t_{aux.}$ se adapta incluso un 90% más.

- Un simulador que permite en un sistema emulado distribuido medir tiempos de búsqueda y efectividad. Utilizando este prototipo al simular un 30% de nodos en fallo distribuidos uniformemente y cuando los servicios buscados y presentes en el sistema son escasos, Algoritmo-$t_{aux.}$ encuentra en el 100% de las simulaciones realizadas como mínimo un servicio que satisface la búsqueda.
- Un prototipo que permite medir tiempos de búsqueda y efectividad en un sistema real distribuido geográficamente y basado en una topología en hipercubo. El prototipo se ha probado en un conjunto de 150 máquinas distribuidas a lo largo de Europa.

1.3 Esquema de la tesis

El material del presente documento se ha organizado en 6 capítulos:

- Capítulo II presenta el estado del arte. Se describen estándares, servicios de descubrimiento para sistemas distribuidos y para la topología en hipercubo propuestas donde la topología se genera con interconexiones físicas y donde la topología se genera con interconexiones lógicas. Al final del capítulo, se realiza una síntesis de todo lo presentado en él.
- Capítulo III define qué es un hipercubo. Después se presentan 3 posibles propuestas de otros autores para mantener un hipercubo. Posteriormente se describen en el tercer apartado de este capítulo dos búsquedas en la arquitectura (propuestas de otros autores con las que compararemos).
- Capítulo IV describe el servicio de descubrimiento de servicios. Primero se muestra la arquitectura que se propone en esta tesis. Después se presenta la búsqueda que se propone en este documento.
- Capítulo V dedicado a evaluaciones realizadas. Primero se describen las herramientas utilizadas en las evaluaciones. Se detalla, para cada herramienta, para qué se ha utilizado y se expone por qué algunas herramientas han sido desestimadas después de valorarlas y probarlas. Después se muestra una evaluación de escalabilidad de la propuesta que se realiza en esta tesis. Posteriormente, se muestran evaluaciones bajo condiciones de fallo de la propuesta de esta tesis comparando con 2 propuestas de otros autores que han sido descritas en un capítulo anterior. A continuación se muestran tiempos de búsqueda obtenidos por simulación. Para concluir el capítulo se muestran tiempos de búsqueda en un sistema distribuido real formado por 150 máquinas distribuidas geográficamente a lo largo de Europa, donde se ha implementado el servicio de descubrimiento descrito en esta tesis.
- Finalmente, capítulo VI resume las conclusiones del presente trabajo y sugiere áreas de trabajo futuro.

II. ESTADO DEL ARTE

En este capítulo se describen trabajos relacionados con la propuesta presentada dividiendo el capítulo en 4 apartados. El primer apartado se dedica a estándares y cómo aplicarlos a la propuesta presentada más adelante en el documento. El segundo apartado se centra en describir brevemente servicios de descubrimiento de servicios propuestos. En el tercer apartado de este capítulo se presentan para la topología en hipercubo propuestas donde la topología se genera con interconexiones físicas y donde la topología se genera con interconexiones lógicas En el cuarto y último apartado del capítulo se concluye con una síntesis de todos apartados anteriores.

2.1 Estándares

Tal y como se ha comentado anteriormente ejemplos de sistemas distribuidos donde puede aplicarse la propuesta de esta tesis son los entornos de Grid. Al igual que un portátil comprado para una universidad catalana no podrá recargar su batería directamente enchufando a la corriente en la habitación de un hotel de Luisiana (Estados Unidos), la solución desarrollada para un entorno de Grid particular no siempre funcionará para otro. Si los entornos de Grid llegaron para ser ampliamente adoptados – si están para ofrecer soluciones reales, con riesgos aceptables, para la industria y la e-Ciencia – entonces deben ser interoperables, lo que implica el desarrollo de estándares. También se ha comentado que otros ejemplos de sistemas distribuidos donde puede aplicarse la propuesta de este documento son los entornos de Cloud, pero a día de hoy estos entornos no han desarrollado estándares para ser interoperables y habitualmente pertenecen a una única organización. Así este apartado se centra en estándares para entornos de Grid y cómo aplicarlos a la propuesta de esta tesis, ya que es necesario que una solución para sistemas distribuidos de gran escala sea interoperable como debe serlo una solución para Grid.

2.1.1 Grid Monitoring Architecture

Grid Monitoring Architecture (GMA) fue propuesto por Tierney et al. [9], y ha sido aceptado como estándar (en la categoría de información), para sistemas de monitorización de Grid por el Open Grid Forum [10] (o Global Grid Forum, como se llama ahora).

Se trata de una arquitectura simple compuesta por tres componentes:

- Productor: una fuente de datos en el Grid, por ejemplo, un sensor.
- Consumidor: un usuario de los datos disponibles en el Grid.
- Servicio de Directorio: un almacén de datos que guarda detalles de los productores y consumidores.

La figura 1 muestra la arquitectura de GMA. Un productor registra una descripción de su servicio ofrecido en el Servicio de Directorio. Un consumidor contacta con el Servicio de Directorio para localizar proveedores que dispongan del servicio que busca. Una comunicación directa entre el consumidor y cada uno de los productores seleccionados es establecida entonces. Los consumidores pueden registrarse también en el Servicio de Directorio para ser notificados si nuevos productores que dispongan del servicio buscado se registran en el Servicio de Directorio.

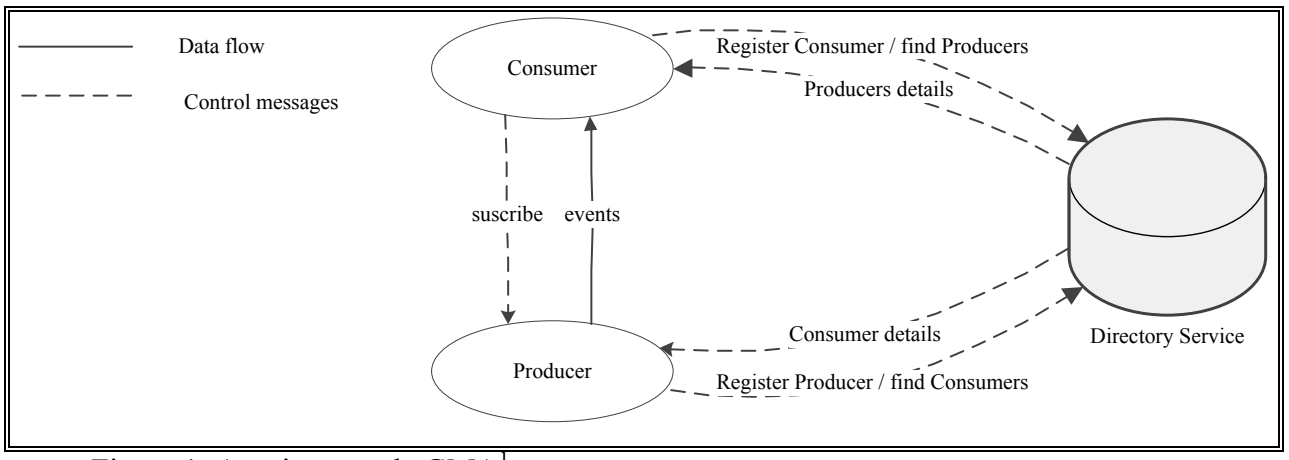

Figura 1: Arquitectura de GMA[1]

La arquitectura presentada en esta tesis puede seguir la misma arquitectura que el GMA, añadiendo una forma de interconectar los Servicios de Directorio presentes en el Grid entre sí.

2.1.2 Globus Toolkit

Globus Toolkit (GT) [12] es un software de código abierto cuyo objetivo es ser un estándar de facto apara desarrollar sistemas distribuidos que compartan servicios. Es ampliamente utilizado en la ciencia y, recientemente, en aplicaciones comerciales. Está siendo desarrollado por la Globus Alliance y otros colaboradores en todo el mundo.

GT está formado por un conjunto de componentes. Por ejemplo, *Grid Resource Allocation & Management* (GRAM), para el envío de trabajos a una máquina determinada. El componente usado para el descubrimiento de servicios es MDS, (*Metacomputing Directory Service, Monitoring and Discovery Service* o *Monitoring and Discovery System*, dependiendo de la versión del GT). MDS define e implementa mecanismos para descubrimiento de recursos/servicios y para monitorización en sistemas distribuidos.

Dependiendo de la version de GT el servicio de descubrimiento de MDS varía. MDS1 (*Metacomputing Directory Service*), liberado en GT 1.1.2 y anteriores, está basado en una base de datos centralizada LDAP [13]. MDS2 (*Monitoring and Discovery Service*), liberado en GT 1.1.3 y GT2.x, está basado en bases de datos LDAP distribuidas. MDS3 (*Monitory and Discovery System*), liberado con GT 3.x, está basado en servicios distribuidos siguiendo el estándar OGSI [14]. MDS4 (Monitoring and Discovery System), liberado con GT4.x, está basado en Web Services [15] descentralizados.

La figura 2 muestra conceptualmente la arquitectura de MDS2, MDS3 y MDS4 dentro de una organización. Está formada por *Resources* y *Aggregators*. *Resources* son los recursos que ofrece una organización al sistema distribuido, y *Aggregators* son índices que almacenan información de los *Resources*. Los *Aggregators* pueden registrarse en otros *Aggregators*. En la figura 2 el *Aggregator* superior contiene información de 6 *Resources*. A medida que subimos en

[1] fuente: [11].

la jerarquía, tenemos información de más *Resources*, pero la información generalmente está menos actualizada. Los clientes pueden buscar servicios conectándose a los *Resources* (la información será más reciente pero es necesario que el cliente conozca *Resources* que existen en el sistema para poderse conectar a ellos) o pueden buscar servicios conectándose a los *Aggregators* (aquí no es necesario conocer *Resources* que hay en el sistema pero la información está menos actualizada).

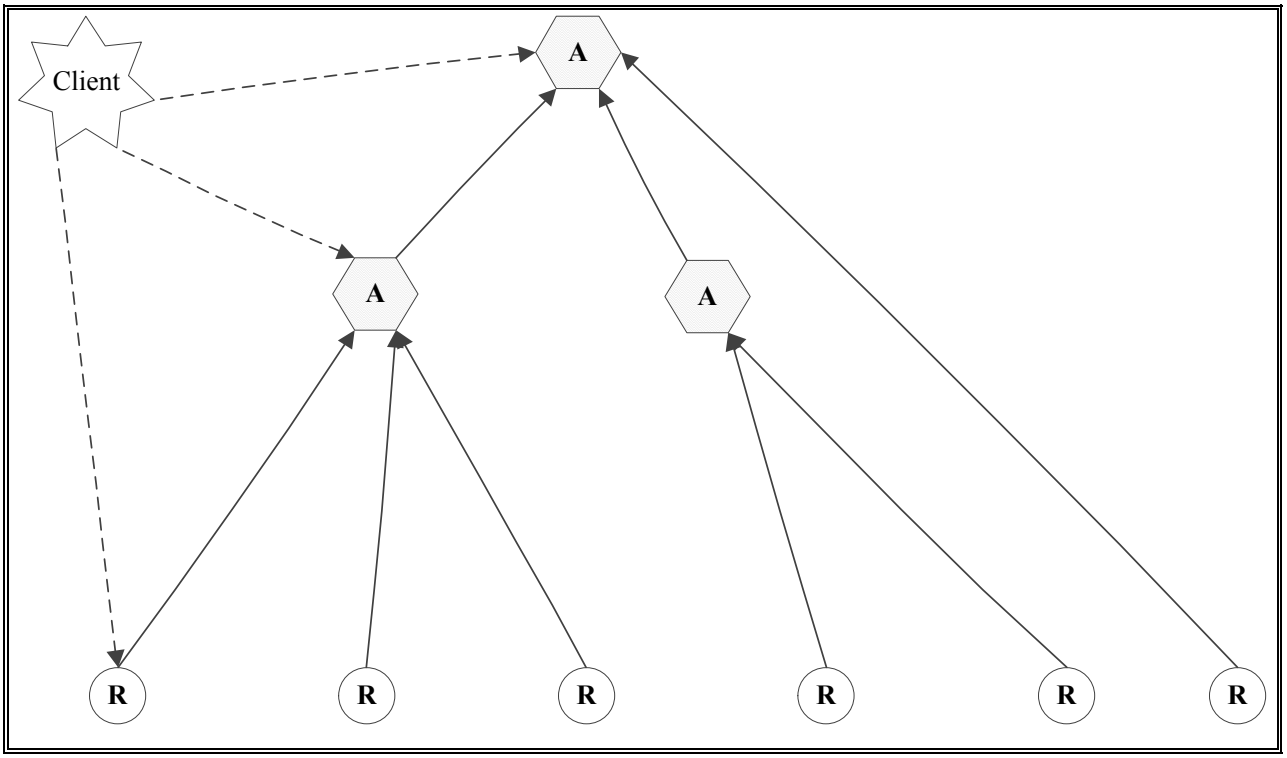

Figura 2: Arquitectura de MDS2, MDS3 y MDS4.

Centrándonos en la última versión del MDS, MDS4 [16], se han implementado dos *Aggregators*: *Index Service* y *Trigger Service*. Un *Index Service* monitoriza información de los servicios del sistema y la publica en un único lugar. Generalmente, se espera que una organización implemente uno o más *Index Services*, que almacenen información de todos los recursos disponibles en la organización. El *Index Service* de MDS4 guarda, no sólo información de la localización del servicio, sino también una versión cacheada del servicio, y mantiene la copia cacheada actualizada, utilizando mecanismos de tiempo de vida. En general, los *Index Services* pueden organizarse en jerarquías, pero no hay un índice global que disponga de información de todos los recursos del sistema. El *Aggregator* del *Trigger Service* guarda información de los servicios y ejecuta alguna acción en función de esa información (por ejemplo, enviar un e-mail a los administradores, cuando se alcanza cierto valor de carga del sistema o cuando el espacio en disco disponible está por debajo de algún valor, con el objetivo de identificar posibles anomalías en el sistema).

La propuesta de esta tesis puede implementarse utilizando MDS4. Esto implicaría interconectar los *Index Services* que consideren cada una de las organizaciones que forman el sistema distribuido siguiendo la topología presentada en esta tesis y que los clientes se conecten a cualquiera de los *Index Service*.

2.2 Servicios de descubrimiento

Se han publicado bastantes servicios de descubrimiento en el contexto de sistemas distribuidos de gran escala, aunque no todavía suficientes. Este apartado describe el servicio de descubrimiento de Secure Service Discovery Service (SDS), Intentional Naming System (INS), arquitecturas Peer-to-Peer (P2P), TeraGrid, Uniform Interface to Computing Resources (UNICORE), Advanced Resource Connector y Resource Selection Service. Los 3 primeros se consideran tolerantes a fallos de todos los comparados en [17]. El resto son relevantes en Grid.

2.2.1 Secure Service Discovery Service

La arquitectura de SDS [18] se compone de servidores SDS organizados siguiendo una topología en árbol jerárquica. Ejemplos de jerarquías en árbol que pueden utilizarse son las basadas en dominios administrativos (compañías, gobiernos, organizaciones, etc.), las basadas en la topología de la red (saltos de red), las basadas en medidas de métricas de red (ancho de banda, latencia, etc.) y las basadas en localización geográfica.

Cada servidor SDS puede participar en una o más jerarquías, manteniendo punteros separados a los servidores SDS padres e hijos de cada una de las jerarquías a las que pertenecen.

Cada servidor SDS es responsable de un conjunto de dominios. Una vez un servidor SDS ha establecido su conjunto de dominios, almacena las descripciones de servicios de los dominios de los que es responsable y, las descripciones de servicios de los dominios de sus hijos en la jerarquía SDS a la que pertenece; cachea estas descripciones y propaga un resumen de esta información a su servidor SDS padre. La figura 3 muestra la arquitectura de SDS. El servidor SDS C contiene descripciones de servicios de los dominios C, A y B, ya que los servidores SDS A y B son nodos hijos del servidor C en la jerarquía del ejemplo, propagando un resumen de estas descripciones al servidor SDS D que es su servidor SDS padre.

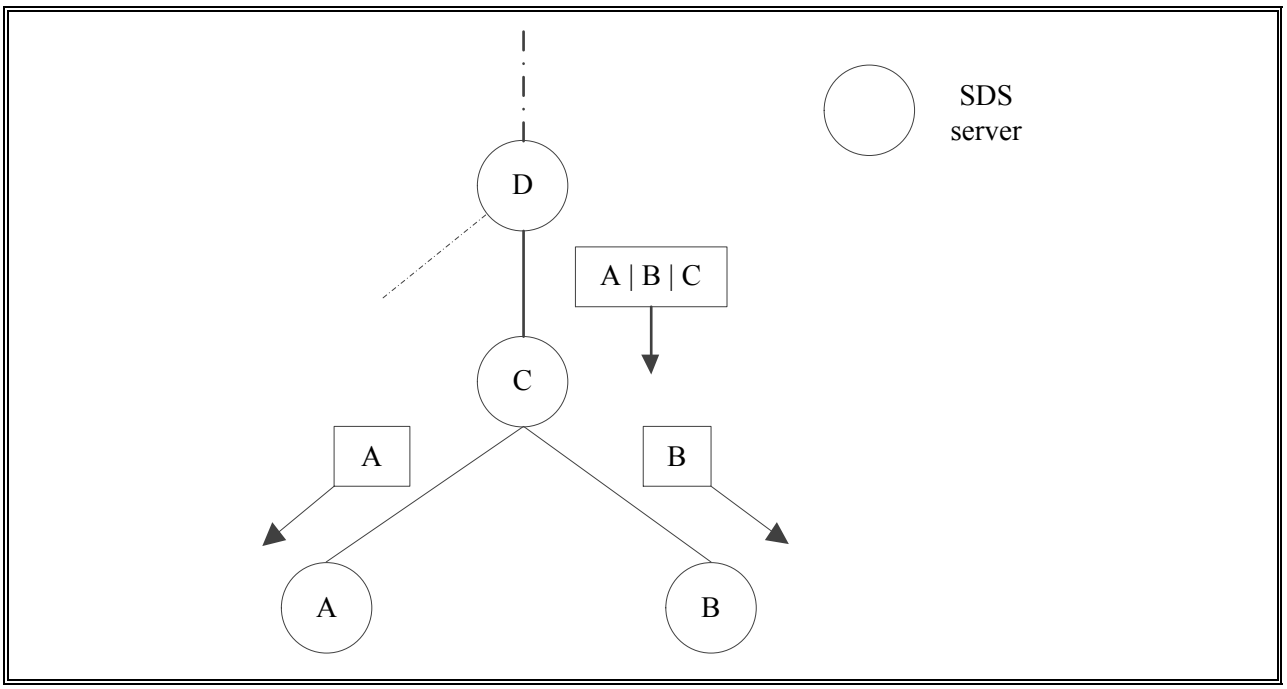

Figura 3: Arquitectura de SDS.

La función principal de un servidor SDS es localizar servicios que le piden los clientes. Cuando un cliente busca un determinado servicio, el cliente se conecta a un servidor SDS y envía su petición. El servidor SDS busca en su base de datos local y devuelve al cliente todas las descripciones de servicios que cumplen los requisitos de la petición de éste. En caso de que el servidor SDS no disponga de ninguna descripción de servicio que cumpla las necesidades del cliente, el servidor SDS reenvía la petición recibida a su servidor SDS padre.

Un servidor SDS puede utilizar uno o varios servidores de respaldo que comparten las descripciones de servicios de éste. Si el servidor SDS deja de funcionar, uno de los servidores SDS de respaldo toma su relevo de forma transparente al cliente.

2.2.2 Intentional Naming System

El sistema de descubrimiento INS está formado por una overlay de intermediarios que se llaman *Intentional Name Resolvers* (INRs) [19]. Los INS se agrupan por dominios y se propone que cada dominio tenga un "resolvedor de espacio de dominio" (DSR) que contiene la lista de INRs de su dominio. La información de los servicios que ofrece un dominio se replica en todos los INRs. Esta información replicada se almacena, dentro de cada INR, en una estructura en árbol llamada name-*tree*. Cuando un INR recibe una búsqueda de servicios por parte de un cliente, el INR busca servicios que satisfagan la petición dentro de su *name-tree*. Cabe destacar que este servicio de descubrimiento se considera distribuido porque hay un conjunto de INRs y de gran escala porque en las evaluaciones realizadas escala para varios miles de servicios, pero los mismos autores reconocen dejan como trabajo futuro la adaptación de INS a redes de área extensa que utilicen Internet.

2.2.3 Peer-To-Peer

A grandes rasgos la arquitectura Peer-To-Peer (P2P) puede dividirse en P2P no estructurado y estructurado (sobretodo *Distributed Hash Tables* o DHTs). Se utilizará el término nodo para referirse a lo que hasta ahora se ha llamado intermediarios (entidad que media entre el consumidor y el proveedor de servicios) o servidores.

Adriana Iamnitchi e Ian Foster [20] sugieren una arquitectura distribuida P2P no estructurada para entornos de Grid. Sin embargo, este tipo de P2P no garantiza que un dato buscado y presente en el sistema se encuentre, incluso aunque el sistema carezca de fallos, ya que las búsquedas pueden no consultar el nodo que tiene el dato buscado.

En P2P estructurado basado en DHTs cada nodo mantiene información de enrutado de O(log N) nodos, donde N es el total de nodos presentes en la overlay. Los datos y los nodos se mapean en la overlay dependiendo de un identificador. A cada nodo se le asigna la responsabilidad de gestionar una fracción del total de datos. Cuando un nodo de la overlay recibe una petición de búsqueda, las DHTs (en general) son capaces de encontrar un dato presente en la overlay en un número limitado, O(log N), de saltos entre nodos.

2.2.4 TeraGrid

TeraGrid se inició en 2001 como encargo de la National Science Foundation (NSF) [21]. NSF es una agencia de gobierno de U.S.A que apoya la investigación y la educación en todos los campos no médicos de la ciencia y de la ingeniería. El encargo consistía en crear un centro de investigación único Teraflop/s, *the Distributed Terascale Facility* o DTF [22]. DFT estaría constituido por cuatro nodos, una arquitectura del sistema común, una red de ancho de banda amplia y por aplicaciones de software distribuidas. El proyecto finalizó en 2011, como un entorno de Grid que combinaba 11 asociaciones. En la actualidad, el proyecto continúa como Extreme Science and Engineering Digital Environment (XSEDE) [23].

TeraGrid fue coordinado a través del *Grid Infrastructure Group* (GIG), de la Universidad de Chicago, que trabajaba junto con asociaciones de U.S.A que proveían servicios al entorno de Grid. Los usuarios de TeraGrid provienen principalmente de universidades de U.S.A, llegando a ser unos 4.000 usuarios de más de 200 universidades diferentes.

TeraGrid integra computadores de altas prestaciones, recursos de datos, herramientas e instalaciones experimentales. En total, TeraGrid dispone de más de un petaflop de capacidad computacional y de más de 30 petabytes de datos, accesibles vía redes de altas prestaciones. Los usuarios también pueden acceder a más de 100 bases de datos específicas por áreas.

El proyecto TeraGrid implementó un servicio de descubrimiento de servicios llamado *Integrated Information Service* (IIS) para cubrir sus necesidades de localización de servicios [24].

La arquitectura de IIS tiene dos componentes, uno local y otro global. Cada proveedor de servicios que pertenece al entorno de Grid trabaja con un servicio de información local donde publica los servicios que ofrece. Un servicio de información global indexa, agrega y republica la información obtenida de los servicios de información local.

El servicio de información global es tolerante a fallos utilizando técnicas de replicación, y por tanto, replicando el servicio de información global en varios servidores simultáneamente ubicados en distintos lugares.

La figura 4 muestra la arquitectura de IIS. En la izquierda hay múltiples servicios de información local, cada uno de los cuales publica su contenido en los servidores del servicio de información global. Todos los servicios de información global son redundantes. En la derecha están los clientes, cada uno de los cuales accede al servicio de información global utilizando un servicio de nombres dinámico (info.teragrid.org) que, de forma transparente, direcciona a los clientes a uno de los servidores de información global. Utilizando el servidor de nombres dinámico del dominio "dyn.teragrid.org", IIS puede direccionar el nombre info.teragrid.org de un servidor de servicios de información global a otro en unos 15 minutos (el tiempo necesario para actualizar todas las cachés en Internet).

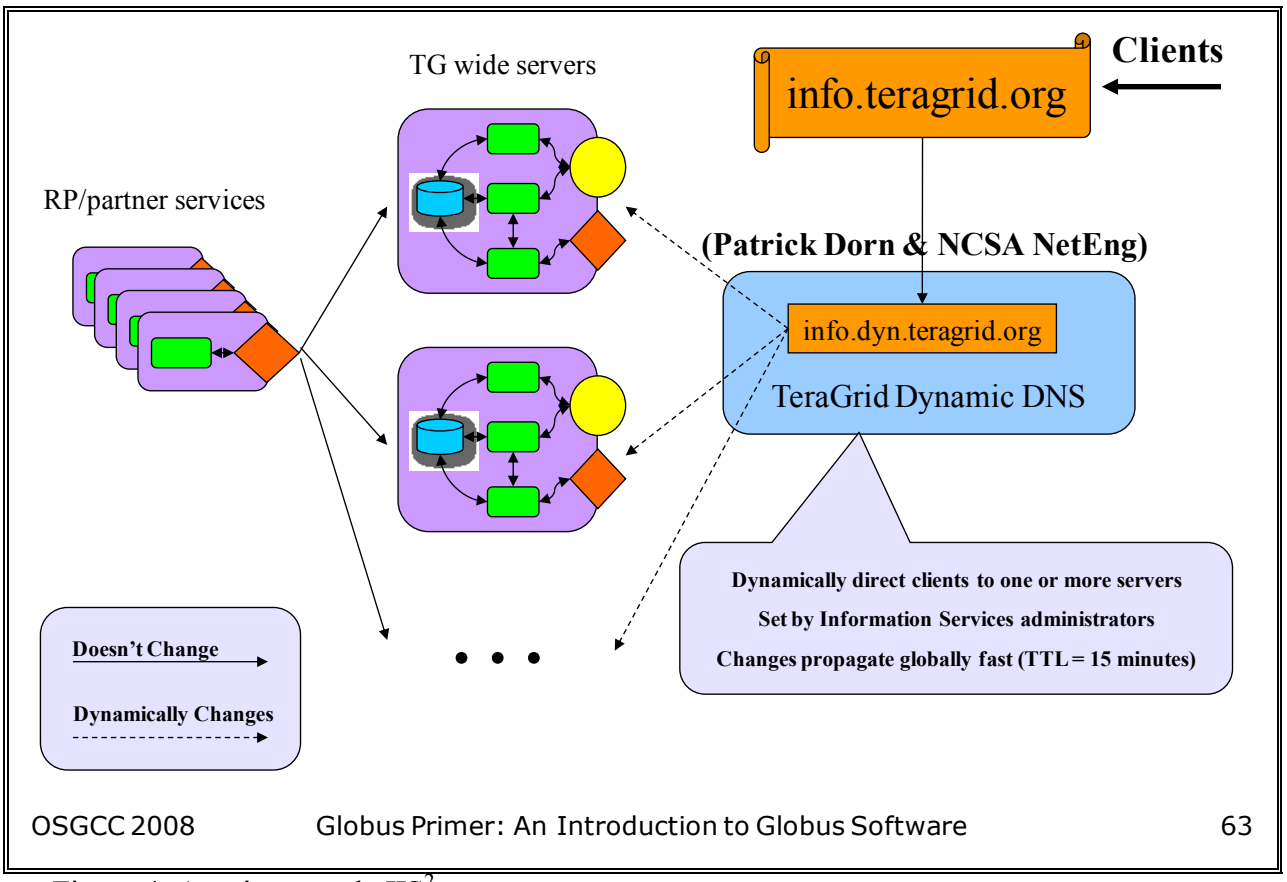

Figura 4: Arquitectura de IIS[2].

[2] Fuente:(Presentation slides) http://www.mcs.anl.gov/~liming/primer/

2.2.5 Uniform Interface to Computing Resources

Uniform Interface to Computing Resources [25] (UNICORE) ofrece software de cliente y de servidor para Grid. UNICORE es un middleware de código abierto bajo licencia BSD.

La arquitectura de UNICORE 6 se compone de tres capas, una capa cliente, una capa de servicios y una capa de sistema. La figura 5 muestra un ejemplo de arquitectura de UNICORE 6 con dos nodos y todos los servicios centrales agrupados bajo un gateway. El gateway actúa como punto de entrada a un nodo UNICORE y se encarga de la autentificación de todos los mensajes que lleguen. En la capa de servicios, para operar como una infraestructura distribuida UNICORE, es necesario un servicio global de registro, donde los diferentes servicios, una vez puestos en marcha, se registran. Este servicio de registro es contactado por los clientes para localizar servicios. Por tanto, el servicio de registro es el servicio central de un Grid UNICORE 6. El resto de componentes mostrados en la figura 5 están relacionados con servicios distintos del servicio de descubrimiento.

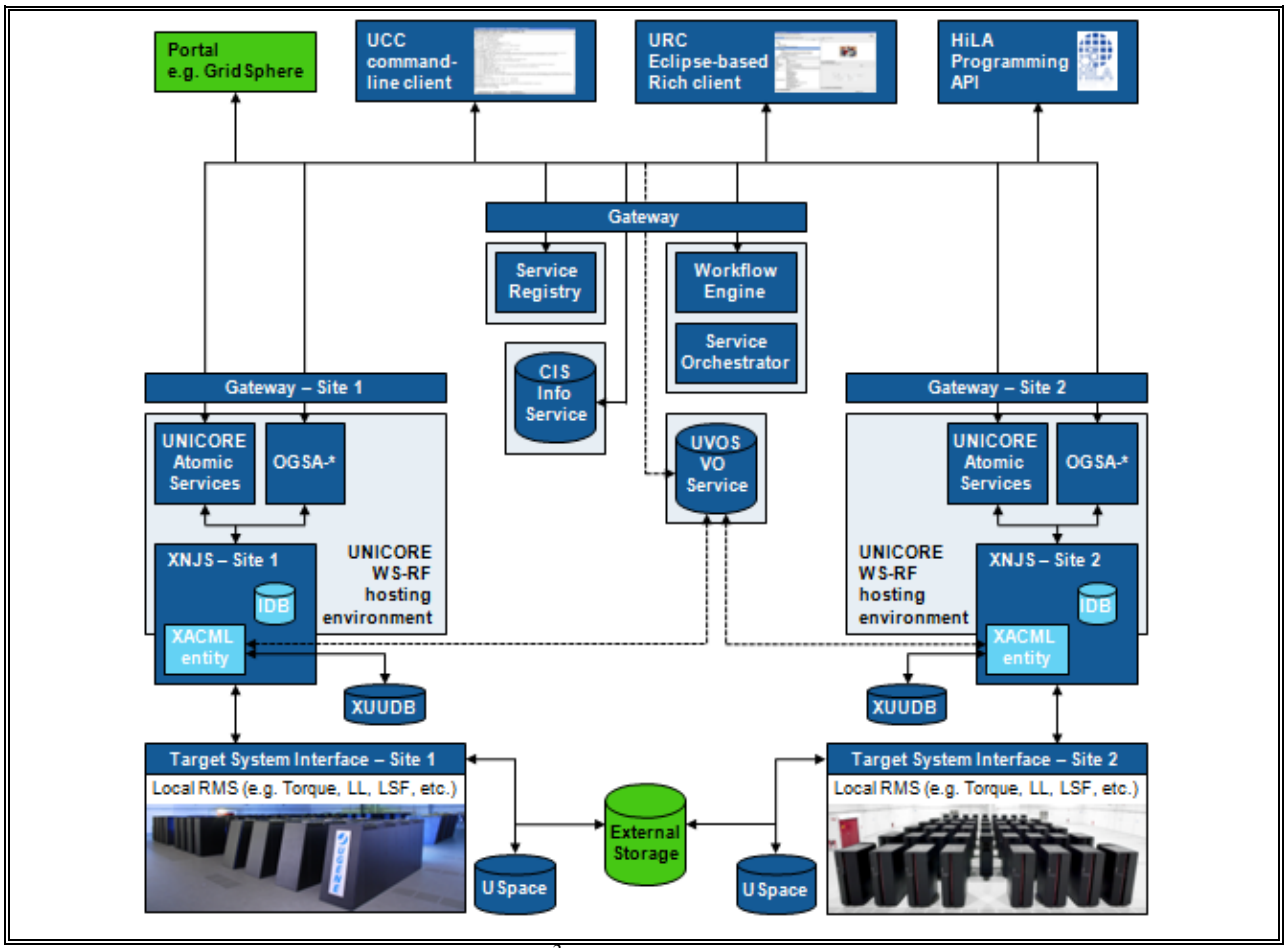

Figura 5: Arquitectura de UNICORE 6[3]

[3] Fuente: http://www.unicore.eu/unicore/architecture.php

2.2.6 Advanced Resource Connector

Advanced Resource Connector (ARC) [26] es un middleware de código abierto usado en varias infraestructuras Grid, como el multidisciplinario Nordic Data Grid [27], el Estonian Grid [28], Swegrid [29] y otros. The NorduGrid Collaboration [30] coordina el desarrollo de ARC. Esta colaboración fue originalmente establecida por varias instituciones académicas nórdicas y por centros de investigación, pero estuvo siempre abierta a la incorporación de nuevos miembros. El nuevo contrato de colaboración, firmado en 2011, tiene 11 colaboradores de 10 países.

ARC define el Grid como un conjunto de recursos que se registran en un sistema de información común. El sistema de información común sirve como eje central del Grid y consiste en tres componentes: *Local Information Services* (LIS), *Index Services* (IS) y *Registration Processes* (RP). Los *Local Information Services* son responsables de la descripción y caracterización de recursos. Los *Index Services* se utilizan para mantener listas dinámicas de recursos disponibles, conteniendo la información de contacto de los proveedores de los recursos. Un proveedor de recursos puede ser un LIS u otro IS. El contenido del *Index Service* es publicado por los proveedores vía el *Registration Process* y es periódicamente borrado por el *Index Service*. De esta forma se mantiene una lista dinámica de proveedores y recursos disponibles. Los *Index Services* son básicamente listas planas de URLs de contacto de otros IS y LIS.

Los *Local Information Services* y los *Index Services* de ARC están organizados en una jerarquía en árbol mutinivel que utiliza el *Registration Process*. Los LIS representan el nivel más bajo de la jerarquía en árbol. Los recursos se registran en el primer nivel de *Index Services*, que se registra en el segundo nivel de *Index Services*, y así sucesivamente. El registro finaliza en el *Top Level Indice*, que representa el root del árbol jerárquico. Para evitar que el sistema tenga un único punto de fallo, ARC sugiere un árbol con múltiples roots.

2.2.7 Resource Selection Service

Resource Selection Service (ReSS) [31] es una infraestructura para seleccionar recursos en los que se van a ejecutar trabajos. El sistema ReSS se ha implementado en dos importantes infraestructuras Grid: Open Science Grid (OSG) [32] y FermiGrid [33], el Grid del campus de Fermilab. OSG es un consorcio de laboratorios nacionales y universidades de U.S.A que proporciona un Grid para satisfacer las necesidades computacionales de comunidades científicas. OSG pone a disposición de sus colaboradores cientos de recursos computacionales y de datos.

El sistema ReSS está compuesto por dos componentes: un proveedor de información desplegado en los recursos y un repositorio de información central donde buscar recursos. ReSS publica la información de los recursos en el repositorio de información central.

2.3 Hipercubo

La tesis presentada propone un servicio de descubrimiento con una arquitectura descentralizada que permite la búsqueda de servicios distribuidos. Se basa en una overlay con topología hipercubo. Sobre la overlay se propone un algoritmo que localiza eficientemente servicios en presencia de fallos antes de que los algoritmos de recuperación ante fallos actúen y reconfiguren la overlay, es decir, se proponen algoritmos con elevada flexibilidad estática.

Para obtener en el servicio de descubrimiento elevada flexibilidad estática es necesario escoger una topología adecuada y las propiedades del hipercubo [34] la convierten en una buena candidata. Esta afirmación se constata, por ejemplo, en [5] para sistemas basados en DHTs que utilizan algoritmos de enrutamiento. En dicho trabajo además, se evalúa la flexibilidad estática de algoritmos de enrutamiento muy conocidos que aprovechan las propiedades de la topología sobre la que se ejecutan. Las topologías evaluadas son árbol, hipercubo, *butterfly*, anillo y una topología híbrida que combina árbol y anillo. Los resultados obtenidos muestran que el enrutamiento en el hipercubo tiene la mejor flexibilidad estática cuando el porcentaje de nodos no-vivos es no mayor que el 30% del total de nodos presentes en la overlay (por ejemplo, un Grid). Los algoritmos propuestos en esta tesis no utilizan enrutado pero utilizan caminos alternativos que evitan los nodos en fallo aprovechando las propiedades de la topología en hipercubo tal y como hace el algoritmo de enrutamiento en el hipercubo.

En el área de multiprocesadores se han propuesto múltiples algoritmos en los que los procesadores se interconectan en hipercubo. De estos algoritmos los que tienen más afinidad con el presentado en esta tesis son los algoritmos denominados de *broadcast* en presencia de fallos. Un algoritmo de este tipo intenta enviar el mismo mensaje al máximo número de procesadores teniendo en cuenta que esos procesadores pueden estar en fallo.

Algunos de esos algoritmos para hipercubo en multiprocesadores proponen algoritmos en los que es necesario conocer información global. Por ejemplo, en [35] se necesita conocer el número total de procesadores en fallo. Este tipo de algoritmos no se pueden aplicar a sistemas distribuidos de gran escala ya que en un sistema distribuido no se conoce a priori información global del sistema como es el número total de fallos.

Otros de esos algoritmos para hipercubo en multiprocesadores envían múltiples copias del mismo mensaje a cada procesador usando caminos diferentes [36] [37] [38]. Este tipo de algoritmos aplicados a sistemas distribuidos de gran escala generan elevado *overhead*, ya que al enviar múltiples copias para aumentar la probabilidad de que se reciba el mensaje se consume un alto ancho de banda.

En cualquier caso las interconexiones del hipercubo sobre el que se ejecuta el algoritmo propuesto en esta tesis son interconexiones lógicas, lo que nos permite generar interconexiones nuevas si el algoritmo lo requiere como hace el Algoritmo-t_{aux}. Esto no es posible en un multiprocesador ya que las redes de interconexión son físicas y por tanto no se pueden modificar.

En los últimos años el hipercubo se está proponiendo en sistemas distribuidos como en SCAFFOLD [39], HyperCUP [40], Edutella [41], HyperD y en el sistema propuesto por HaoRen et al. Con respecto a las arquitecturas de los anteriores sistemas distribuidos la arquitectura de la presente propuesta añade información parcial de fallos del sistema. Esta información permitirá que la búsqueda en sistemas distribuidos de gran escala (de órdenes de magnitud de cientos, millares y millones) se adapte a los fallos que se producen irremediablemente en el sistema.

SCAFFOLD utiliza el hipercubo como estructura para guardar datos en un nodo pero no especifica algoritmo alguno de búsqueda entre los nodos que guardan estos datos. HyperCUP, Edutella y HyperD utilizan una overlay con topología en hipercubo y el algoritmo de búsqueda entre los nodos que forman un hipercubo es el mismo.

El algoritmo que se presenta en esta tesis tiene mayor flexibilidad estática que el algoritmo de búsqueda utilizado en HyperCUP, Edutella y HyperD y que el algoritmo de búsqueda utilizado en la propuesta de HaoRen et al. Por ejemplo, en un sistema distribuido formado por alrededor de 1.000.000 de nodos que comparten sus recursos y en el que un 30% de los nodos fallan de forma uniformemente distribuida, si utilizamos como métrica para comparar la flexibilidad estática de un algoritmo el % de nodos consultados en presencia de fallos cuando el sistema no dispone del recurso buscado, el algoritmo presentado en esta tesis consulta alrededor de un 94% de los nodos, mientras que el algoritmo propuesto por HaoRen et al. sólo un 4%. En la misma línea, considerando el algoritmo que se aplica en HyperD para el ejemplo mostrado en su artículo se consulta el 100% de los nodos frente a un 55,5%.

2.4 Síntesis

Con respecto a arquitecturas descentralizadas en hipercubo la arquitectura de la presente propuesta añade información parcial de fallos del sistema pudiéndose implementar siguiendo los estándares descritos en el primer apartado de este capítulo (GMA y Globus Toolkit).

Los sistemas descritos en el segundo apartado de este capítulo se pueden agrupar en centralizados (INS, TeraGrid, UNICORE 6 y ReSS), en jerárquicos (SDS y ARC), P2P no estructurados y P2P estructurados basados en DHTs.

Con respecto a los servicios de descubrimiento centralizados, aunque en sistemas distribuidos se han usado y se usan topologías centralizadas, estos sistemas centralizados no son escalables ya que las prestaciones del sistema se degradan considerablemente si se aplican en sistemas distribuidos de gran escala (el servidor es un cuello de botella). Por otro lado, sin aplicar técnicas de replicación al único servidor del sistema de descubrimiento, el sistema tiene un único punto de fallo (el servidor).

Con respecto a los servicios de descubrimiento jerárquicos, éstos usan replicación de información de los servidores hijos en su o sus servidores padres lo que también hace que sean poco escalables y como en los servicios de descubrimiento centralizados, sin aplicar técnicas de replicación (en el servidor padre de toda la jerarquía), el sistema tiene un único punto de fallo.

Los servicios de descubrimiento P2P no estructurados sí son escalables y no tienen un único punto de fallo, pero no garantizan que un dato buscado y presente en el sistema se encuentre, incluso aunque el sistema carezca de fallos, ya que los algoritmos de búsqueda se basan en esquemas que podrían no consultar el nodo que tiene el dato buscado.

En relación con los sistemas de descubrimiento P2P estructurados basados en DHTs, estos son escalables y no presentan un único punto de fallo al igual que el sistema de descubrimiento presentado en esta tesis. También como en las DHTs, en este trabajo cada nodo de la overlay mantiene información de enrutado de O(log N) nodos, los datos se encuentran distribuidos entre los nodos y una búsqueda de servicios se resuelve en O(log N) o O(log N)+1 pasos como máximo. A diferencia de las DHTs, en el sistema propuesto los datos no están previamente organizados entre los nodos por lo que las actualizaciones de datos generan menos *overhead* en el sistema. Actualizar un dato en el sistema de descubrimiento que se propone implica el envío de un solo mensaje, mientras que en una DHT como Chord [42] se envían 2* O((log N)^2) mensajes.

En la overlay propuesta, al no estar los datos previamente organizados, no se aplican algoritmos de búsqueda basados en enrutamiento. Esto hace que, a diferencia de las DHTs donde una búsqueda envía O(log *N*) mensajes, el sistema de descubrimiento propuesto requiere en una búsqueda entre 0 y *N* mensajes. Así pues, no organizar previamente los datos entre los nodos genera menos *overhead* en la actualización de datos y mayor *overhead* en la búsqueda. Por otro lado, independientemente de la decisión que se tome de organizar previamente o no los datos entre los nodos, los algoritmos de búsqueda propuestos en este trabajo pueden aplicarse a sistemas de descubrimiento P2P estructurados basados en DHTs como Chord para realizar búsquedas que no pueden resolverse, utilizando enrutamiento; por ejemplo, buscar en un sistema de almacenamiento de ficheros información de todos los ficheros con una determinada extensión.

III. HIPERCUBO

En este capítulo se define qué es un hipercubo y, aunque no es el tema del presente trabajo, se describen algunas propuestas de otros autores que abordan el mantenimiento de una red con topología en hipercubo. También se describen algoritmos de búsqueda en un hipercubo con los que se comparará el servicio propuesto en esta tesis.

3.1 Definición de hipercubo

Un hipercubo *n*-dimensional completo (H_n) tiene $V(H_n) = 2^n$ nodos. Cada nodo tiene un identificador que va de 0 a 2^n-1. Dos nodos están conectados directamente entre ellos (se dice que son vecinos en la dimensión *m*) si las representaciones binarias de sus identificadores se diferencian exactamente en el bit *m*. Así, en un hipercubo H_n completo, cada vértice tiene exactamente *n* vecinos. La figura 6 muestra la arquitectura para 2 (H_1), 4 (H_2) y 8 (H_3) nodos respectivamente (hipercubos completos).

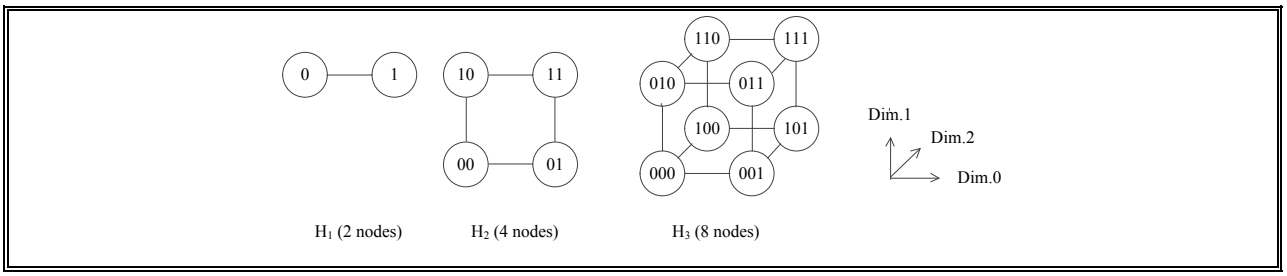

Figura 6: Hipercubos completos.

Un hipercubo *n*-dimensional incompleto tiene $V(H_n)=N$ nodos donde $2^{n-1} < N < 2^n$. Así, el hipercubo *n*-dimensional incompleto no está totalmente ocupado. La figura 7 muestra hipercubos 3-dimensionales incompletos formados por 5, 6 y 7 nodos hasta formar un hipercubo 3-dimensional completo H_3 (8 nodos) donde los nodos se van añadiendo con identificadores en orden decimal creciente. Por ejemplo, el caso de 5 nodos está formado por nodos cuyos identificadores en decimal van de 0 a 4, el caso de 6 nodos está formado por nodos cuyos identificadores en decimal van de 0 a 5, etc.

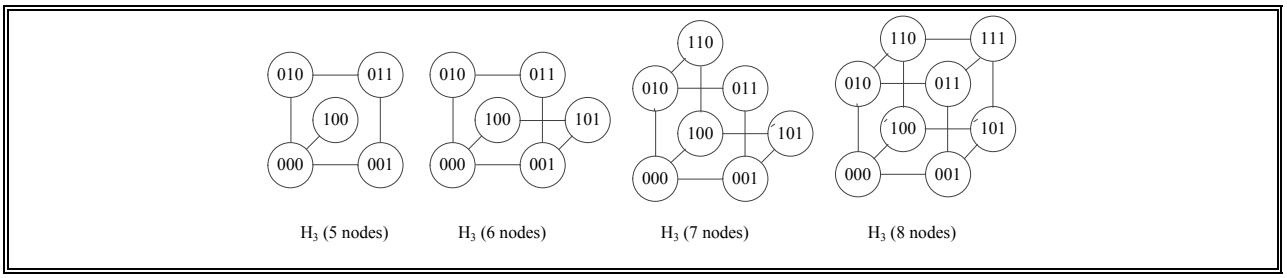

Figura 7: Hipercubos incompletos.

3.2 Mantenimiento de un hipercubo

Para presentar un servicio de descubrimiento de servicios en su totalidad, en este apartado se describen propuestas de otros autores que abordan el mantenimiento de un hipercubo. Cualquiera de estas propuestas o alguna combinación de ellas pueden aplicarse al servicio propuesto en esta tesis.

3.2.1 HyperCast

En HyperCast[44] todos los nodos que forman o quieren formar parte del hipercubo pertenecen a un grupo *multicast*. Ese grupo multicast tiene asociado una dirección de Internet. El *multicast* es el envío de información en una red a múltiples destinos simultáneamente.

Cada nodo de un hipercubo en HyperCast tiene asociada una dirección física y una dirección lógica. La dirección física está formada por la IP del nodo y un puerto UDP común a todos los nodos que utiliza el protocolo. La dirección lógica es un identificador de 31 bits.

En HyperCast, los nodos obtienen su identificador lógico siguiendo un código Gray, para que los identificadores lógicos del último nodo añadido y el siguiente a añadir difieran en binario sólo en un bit y por tanto sean vecinos. En la tabla 1 se muestra el orden de los identificadores siguiendo un código binario y siguiendo un código Gray.

Index	i=	0	1	2	3	4	5	6	7
Binary code	Bin(i)=	000	001	010	011	100	101	110	111
Gray code	G(i)=	000	001	011	010	110	111	101	100

Tabla 1: Comparación entre códigos binarios y códigos de Gray.

En el protocolo se envían 4 tipos de mensajes:

- Mensaje *beacon*: es un mensaje multicast que contiene la dirección física y la dirección lógica del nodo que envía el mensaje, junto con la dirección lógica y un número de secuencia del nodo conocido como HRoot. HRoot es el último nodo añadido a la overlay en hipercubo.
- Mensaje *ping*: es un mensaje que envía cada nodo periódicamente a los nodos de su tabla de vecinos.
- Mensaje *leave*: es un mensaje que envía un nodo cuando quiere abandonar el hipercubo.
- Mensaje *kill*: es un mensaje que recibe un nodo cuando su identificador lógico ya está presente en la overlay.

El objetivo del protocolo es mantener el hipercubo en un estado estable. Se dice que el hipercubo es estable si satisface las siguientes propiedades:

- El hipercubo es *consistente*: todos los nodos tienen un identificador lógico diferente.

- El hipercubo es *compacto*: en un grupo multicast formado por *M* nodos, los nodos tienen identificadores lógicos que van de G(0) a G(*M*-1) donde G(*i*) es el código de Gray de índice *i* (ver tabla 1).

- El hipercubo está *conectado*: todos los nodos conocen la dirección física de cada uno de sus vecinos en el hipercubo.

El hipercubo pasa de un estado estable a uno inestable cuando están entrando y/o saliendo nodos y/o se están produciendo fallos en el hipercubo.

En la figura 8 se ilustran los mensajes enviados cuando el hipercubo está estable, es decir, el hipercubo es consistente, compacto y está conectado. En ese caso la overlay está formada por 5 nodos siendo por tanto HRoot el nodo con identificador G(*i*=4)=110. HRoot envía mensajes tipo *beacon* periódicamente y estos mensajes son utilizados por el resto de nodos para determinar qué vecinos deben tener presentes en sus tablas de vecinos. También periódicamente cada nodo envía mensajes tipo *ping* a sus vecinos para informarles de que aún está presente en el hipercubo.

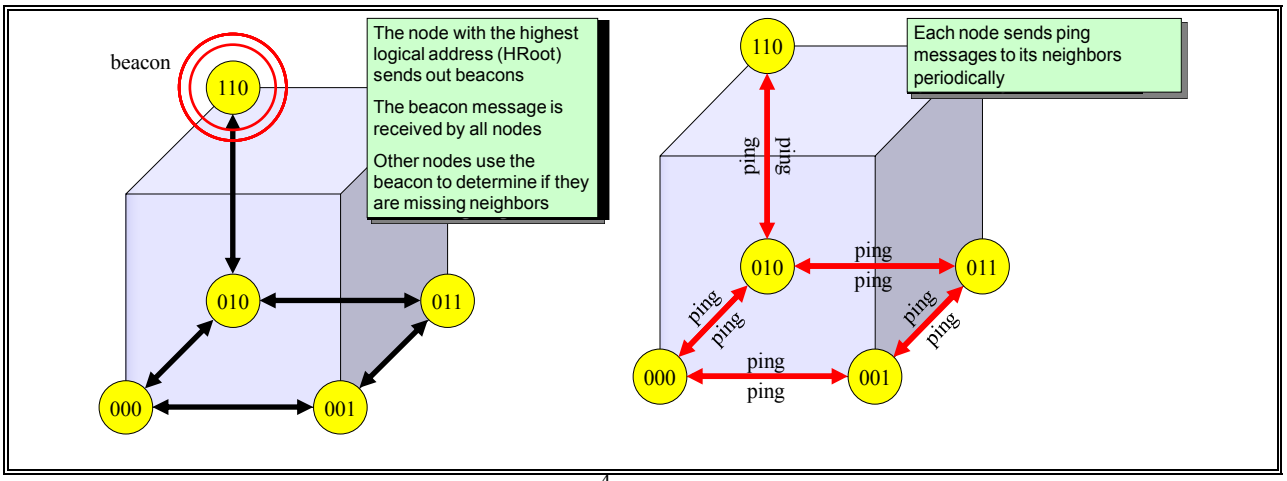

Figura 8: HyperCast - hipercubo estable[4]

En la figura 9 se ilustra el proceso que se sigue cuando un nuevo nodo se incorpora a la overlay de la figura 8. En ese caso el nuevo nodo primero debe unirse al grupo *multicast* del hipercubo para poder enviar mensajes a todos los nodos del hipercubo y poder recibir mensajes de éstos. Después enviará un mensaje de tipo *beacon* periódicamente para anunciar su presencia al grupo. Cuando el nodo HRoot actual (G(*i*=4)=110) lo reciba incorporará al nuevo nodo como su vecino y actualizará que el nuevo HRoot del hipercubo es el nuevo nodo(G(*i*=5)=111), enviándole un mensaje de tipo *ping* ya que es su vecino. Este mensaje informará al nuevo nodo de su identificador lógico en el hipercubo. Una vez el nuevo nodo recibe el mensaje de *ping* responde con otro mensaje de ping a su vecino. Con el mensaje de *ping* recibido el nuevo nodo es capaz de determinar que él mismo es el nodo HRoot del hipercubo y empieza a enviar mensajes de tipo *beacon* periódicamente como nodo HRoot. Los vecinos del nuevo HRoot recibirán los

[4] Fuente: (Talk on Hypercube Protocol, 2000) http://www.comm.utoronto.ca/hypercast/papers.html

mensajes, añadirán al nodo como su vecino y contestarán con mensajes de *ping* ya que son vecinos de éste. Así el nuevo HRoot conocerá las direcciones físicas y lógicas de sus vecinos y podrá completar su tabla de vecinos, pasando el hipercubo de un estado inestable a un estado estable.

En la figura 10 se ilustra el proceso que se sigue cuando falla un nodo en la overlay de la figura 9. Si un nodo falla inesperadamente, los vecinos del nodo que ha fallado percibirán el fallo ya que no obtendrán respuesta a los *pings* que envíen al nodo que ha fallado. Estos vecinos esperarán un tiempo preestablecido $t_{timeout}$ y pasado ese tiempo sin obtener respuesta a los pings empezarán a enviar periódicamente mensajes tipo *beacon* para indicar que han detectado un vecino en fallo. Si el nodo que ha fallado vuelve, los mensajes de tipo *beacon* se utilizan para restablecer el hipercubo al estado estable en el que estaba. Si el nodo que ha fallado no vuelve pasado el periodo de tiempo t_{mising}, los vecinos del nodo en fallo asumen que el nodo que ha fallado ya no volverá y que debe reemplazarse por otro. Entonces empieza un proceso en el que uno o más vecinos envían un *ping* al nodo HRoot para que éste reemplace al nodo en fallo. Cuando el HRoot actual recibe el mensaje de *ping* envía mensajes tipo *leave* a sus vecinos, asume el identificador lógico del nodo que ha fallado y contesta con un mensaje tipo *ping* al nodo que le envió el mensaje *ping* con su identificador lógico nuevo. Como el antiguo HRoot ahora no tiene su tabla de vecinos completa envía mensajes de tipo *beacon*. Los vecinos del antiguo HRoot responden al antiguo HRoot con mensajes de *ping*. Esto completa el proceso y el hipercubo vuelve a estar en un estado estable.

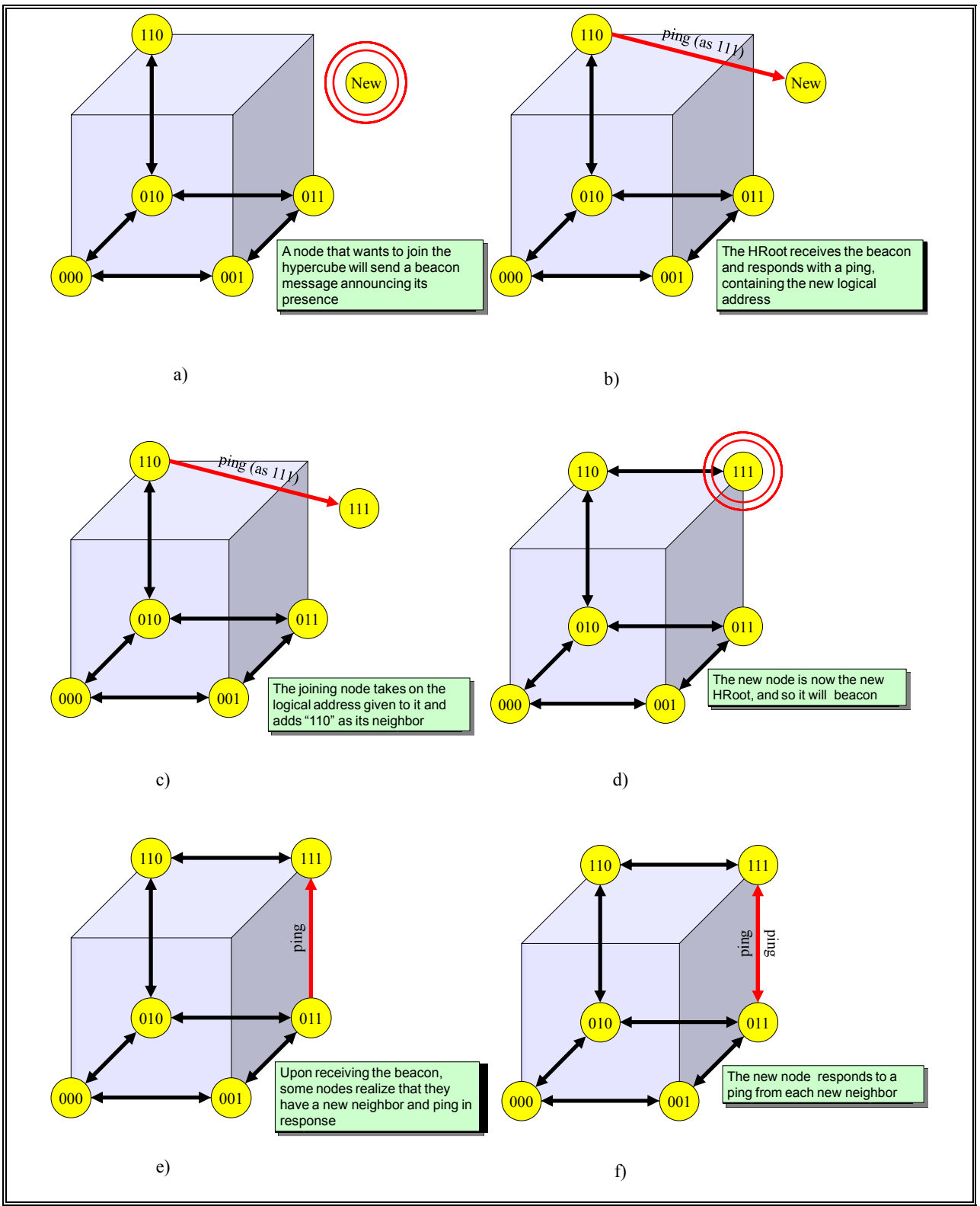

Figura 9: HyperCast - un nodo se incorpora[5]

[5] Fuente: (Talk on Hypercube Protocol, 2000) http://www.comm.utoronto.ca/hypercast/papers.html

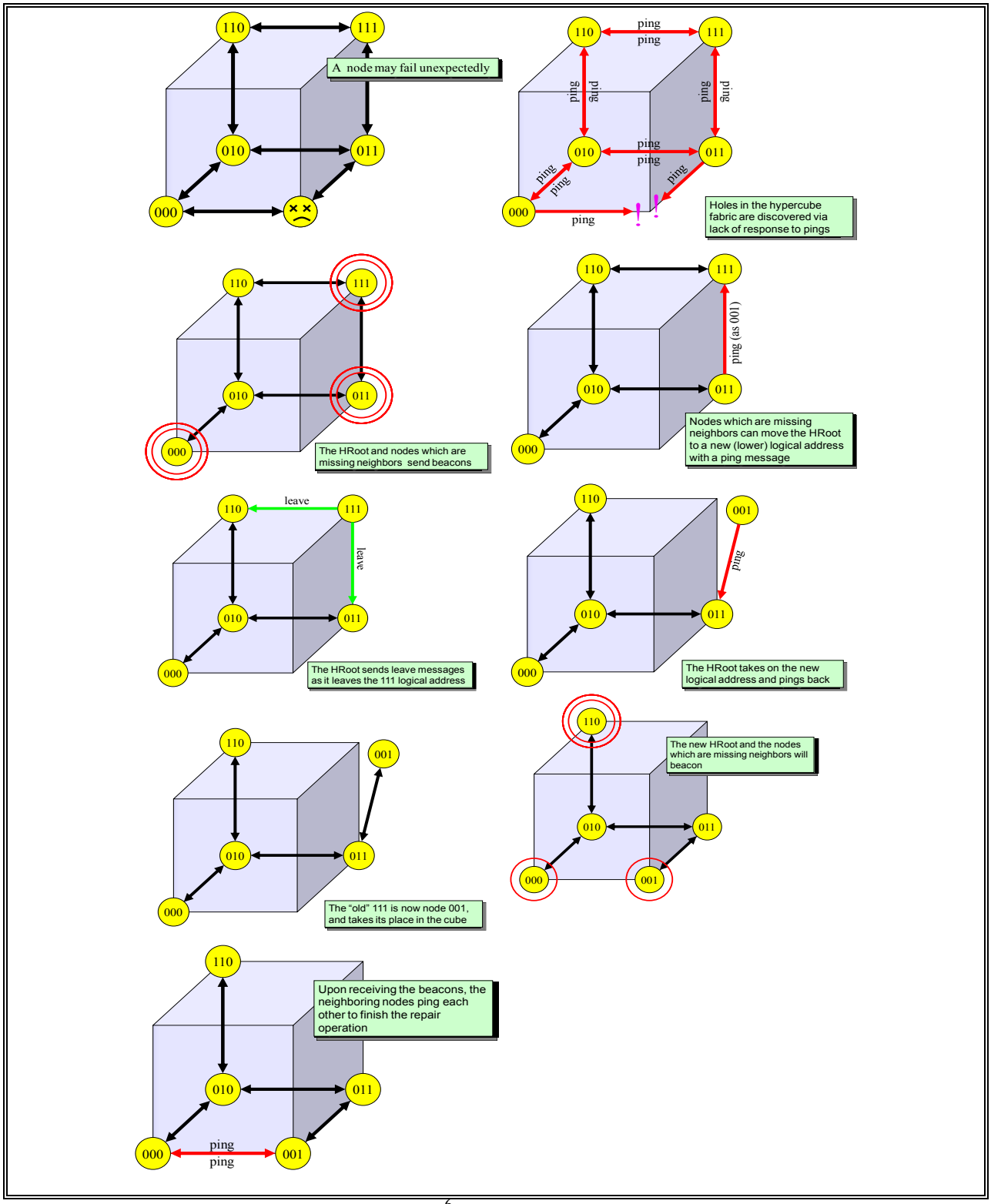

Figura 10: HyperCast - uno nodo falla[6]

[6] Fuente: (Talk on Hypercube Protocol, 2000) http://www.comm.utoronto.ca/hypercast/papers.html

El trabajo que detalla el protocolo de HyperCast centra sus experimentos en determinar si el protocolo es escalable para un gran número de nodos. Dado que no les fue posible simular millones de nodos los experimentos simularon miles de nodos y se asumió que si los datos obtenidos para miles de nodos mostraban que HyperCast era escalable, era razonable asumir que sería escalable también para un número mayor de nodos.

Las medidas analizadas fueron el número de paquetes transmitidos, el número de bytes transmitidos y el tiempo necesario para devolver al hipercubo a un estado estable.

Estas medidas analizan dos comportamientos del protocolo. Primero, el tráfico de control transmitido debe ser escalable con el tamaño del grupo *multicast* que forma el hipercubo. Si el control de tráfico se incrementa linealmente con el tamaño del grupo, entonces el protocolo no sería capaz de soportar un gran número de nodos. Segundo, se quería mostrar que el tiempo necesario para devolver al hipercubo a un estado estable no es dependiente del tamaño del hipercubo. Este tiempo muestra cómo de rápido el protocolo de HyperCast puede incorporar cambios dinámicos en el hipercubo.

Se diseñaron tres experimentos sobre el *testbed* descrito anteriormente para evaluar con respecto al tamaño del hipercubo el efecto del número de nodos que se unen al hipercubo, el efecto del número de nodos que fallan inesperadamente y el efecto de la pérdida de paquetes.

En el experimento del efecto del número de nodos que se unen al hipercubo con respecto al tamaño del hipercubo se concluyó que HyperCast es escalable.

En el experimento del efecto de nodos que fallan inesperadamente con respecto al tamaño del hipercubo se concluyó que a medida que el número de nodos aumenta el tiempo necesario para reparar un nodo en fallo no aumenta. Esto permitía mostrar que el protocolo de HyperCast escala bien para hipercubos con un elevado número de nodos.

En el experimento del efecto de la pérdida de paquetes con respecto al tamaño del hipercubo se vio que el tráfico *multicast* crecía rápidamente a medida que aumentaba la pérdida de paquetes. La solución propuesta a esta problemática fue que era necesario ajustar el valor de $t_{timeout}$ en función de la pérdida de paquetes. Recordemos que $t_{timeout}$ es el tiempo máximo que espera un nodo las respuestas a los mensajes ping de sus vecinos y que pasado ese tiempo sin obtener respuesta de un nodo vecino empieza a enviar periódicamente mensajes tipo beacon para indicar que ha detectado un vecino en fallo.

3.2.2 Propuesta de HaoRen et al.

En la propuesta de HaoRen et al. se obtiene el identificador lógico de un nodo siguiendo un código Gray como en HyperCast. En la tabla 1 (ya presentada anteriormente) se muestra el orden de los identificadores siguiendo un código binario y siguiendo un código Gray. En esta propuesta se describe cuando un nodo se incorpora y cuando un nodo sale, considerando en este último caso que la overlay ya no va a contener más a ese nodo.

Cuando un nodo sale, ese nodo se convierte en un espacio libre (*free space*) y otro nodo presente en el hipercubo asume su lugar. El nodo que asume su lugar es el nodo vecino en la dimensión más pequeña que tiene el nodo. Como los autores, para hacer referencia a la dimensión más pequeña que tiene un nodo se utilizará el término L_{min}. En la figura 11 se muestra un ejemplo

en un hipercubo 3-dimensional formado por 6 nodos (figura 11.a) donde el nodo G(i=3)=010 (L_{min}=0) sale del hipercubo (figura11.b). Así el nodo que asume el lugar de G(i=3)=010 es G(i=2)=011.

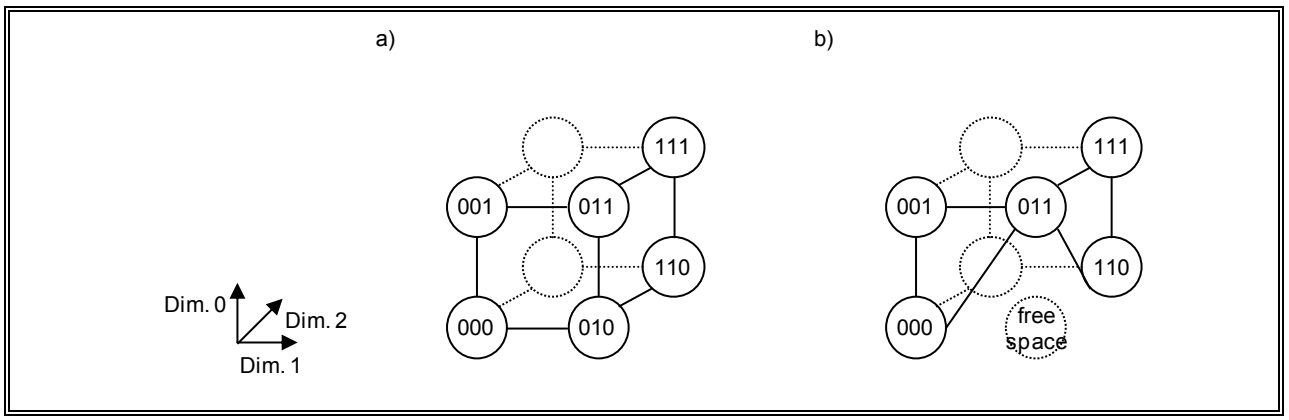

Figura 11: HaoRen et al. – un nodo sale.

Cuando un nodo se incorpora si hay espacios libres en el hipercubo, el nodo que se incorpora es el nodo con valor i menor siguiendo el orden de códigos de Gray. En caso contrario, el nodo que se incorpora es el nodo con identificador siguiente siguiendo el orden de códigos de Gray.

Cabe notar que para aplicar la técnica que describen los autores en su propuesta es necesario que el nodo que se incorpora obtenga de alguna forma como mínimo cómo contactar con los nodos que han asumido el lugar de otro y el valor i mayor del hipercubo. Por ejemplo, en la figura 11 cómo contactar con el nodo G(i=2)=011 y que i=5.

3.2.3 HyperD

A diferencia de la propuesta de HaoRen et al. y de HyperCast donde cada nodo de la overlay tiene asociado un único identificador lógico, en HyperD cada nodo de la overlay tiene asociada una dirección física y como mínimo un identificador lógico, pudiendo tener más de uno. Los identificadores lógicos son siempre los identificadores de un hipercubo n-dimensional completo, por lo que en HyperD se distingue entre hipercubo lleno (todos los nodos tienen un único identificador lógico) e hipercubo parcial (algún o algunos nodos tienen más de un identificador lógico), siendo el caso más habitual a medida que aumenta el número de nodos de la overlay el de tener un hipercubo parcial. En la figura 12 se muestra un ejemplo de un 1-dimensional lleno HyperD (H_1) con 2 nodos, de un 2-dimensional parcial HyperD (H_2) con 3 nodos y de un 2-dimensional lleno con 4 nodos respectivamente.

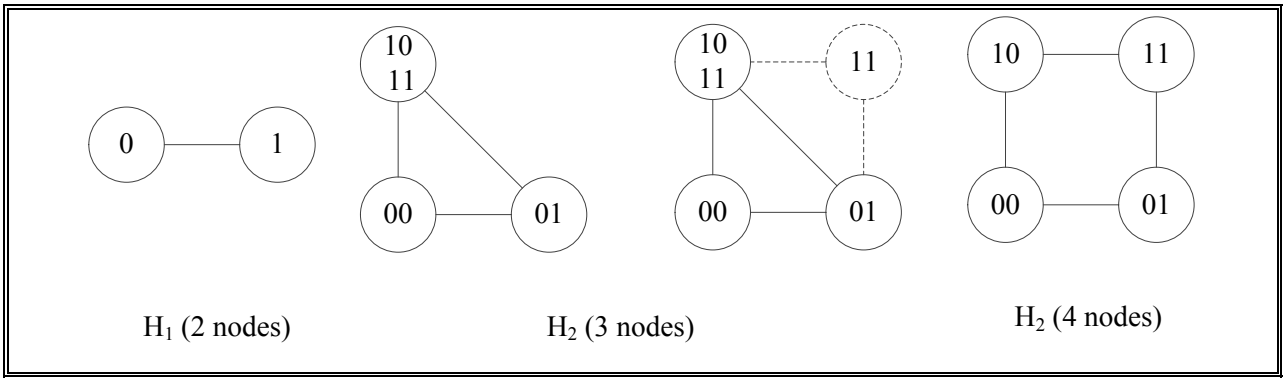

Figura 12: HyperD con 2, 3 y 4 nodos.

En HyperD cada nodo almacena la dirección física y lo o los identificadores lógicos de cada uno de sus vecinos. Periódicamente se actualiza esta información contactando con sus vecinos.

Cuando un nodo presente en HyperD deja de formar parte de HyperD, bien porque así lo anuncia o porque falla, el nodo aún presente en HyperD que tiene el identificador lógico adyacente al identificador lógico mayor (en decimal) del nodo que deja de formar parte de HyperD añade a sus identificadores lógicos los identificadores lógicos del nodo que ya no forma parte de HyperD. En la figura 13 se ilustra el proceso que se sigue cuando un nodo deja de formar parte de HyperD. El nodo que deja de formar parte de HyperD es el nodo con los identificadores lógicos 011 y 111 (figura 13.a), por tanto, el identificador lógico mayor de ese nodo es 111. Así, el nodo con identificador lógico adyacente a 111 (nodo con identificador 110) pasa a tener los identificadores lógicos que tenía y los nuevos (011 y 111) (figura 13.b)

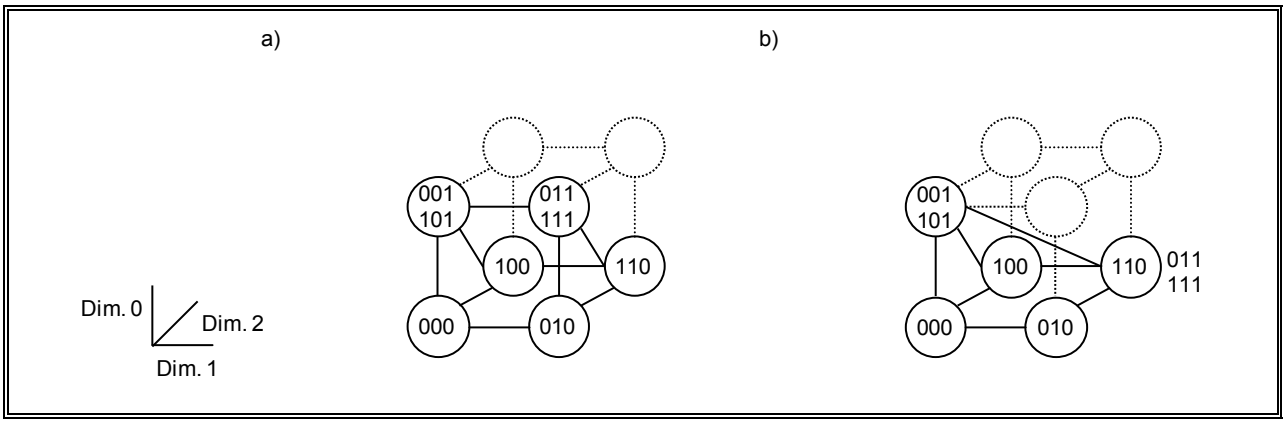

Figura 13: HyperD – un nodo deja de formar parte.

Cuando un nuevo nodo se incorpora contacta con un nodo presente en HyperD y se inicia un procedimiento de asignación de identificador/es lógico/s del hipercubo *n*-dimensional completo que forma HyperD para el nuevo nodo. Si el hipercubo es un hipercubo *n*-dimensional lleno todos los nodos doblan sus identificadores lógicos, pasando el hipercubo a ser un hipercubo (*n*+1)-dimensional y el nuevo nodo toma el identificador lógico de valor *i* mayor siguiendo el orden del código de Gray, conectándose además con todos los vecinos del nodo con identificador lógico G(*i*=0) siguiendo el orden del código de Gray. Si el hipercubo es un hipercubo *n*-

dimensional parcial el nuevo nodo toma algún/os identificadores lógicos de otro nodo ya presente en el hipercubo, dejando ese nodo ya presente de tener esos identificadores. En la figura 14 se muestra el ejemplo de un 2-dimensional lleno (figura 14.a) que pasa a ser un hipercubo 3-dimensional parcial (figura 14.b) y en la figura 15 se muestra un ejemplo de un 3-dimensional parcial con 5 nodos (figura 15.a) donde el nuevo nodo obtiene después del proceso de asignación los identificadores lógicos 011 y 111 (figura 15b)

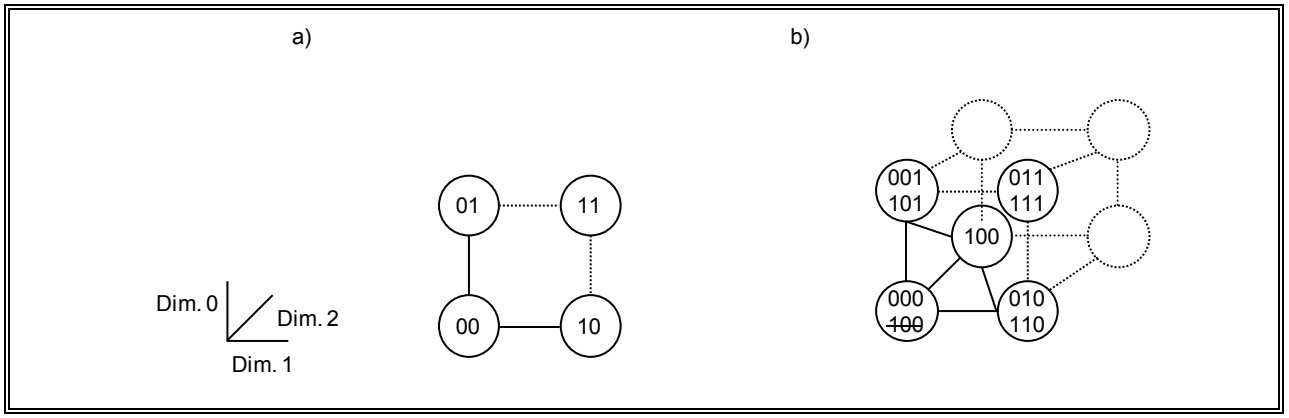

Figura 14: HyperD – un nodo se incorpora (hipercubo lleno).

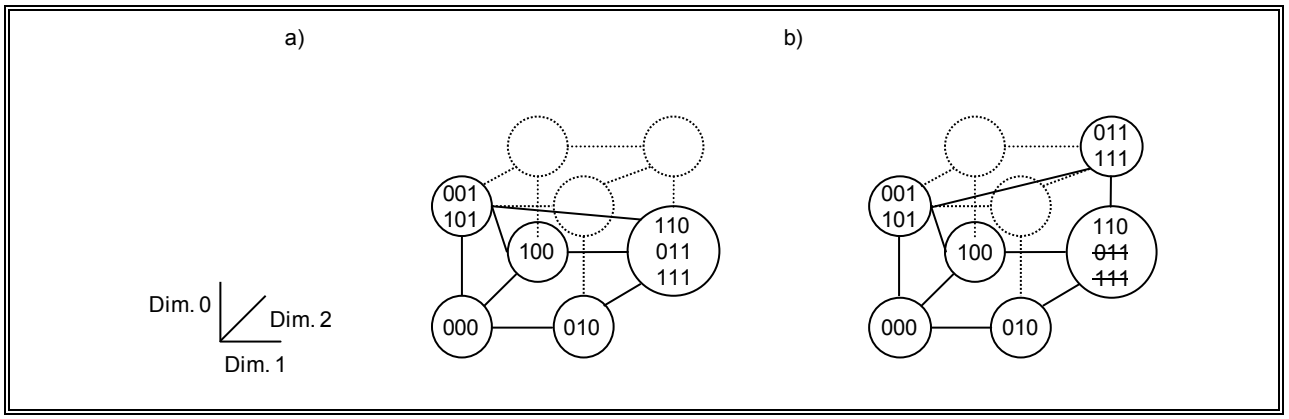

Figura 15: HyperD – un nodo se incorpora (hipercubo parcial).

3.3 Búsquedas en un hipercubo

En este apartado se describen 2 algoritmos para búsquedas en un hipercubo que no pueden resolverse utilizando enrutamiento. Estas propuestas son propuestas de otros autores ya que la propuesta de esta tesis se presenta en un apartado del siguiente capítulo.

3.3.1 Búsqueda propuesta por HaoRen et al.

Para un entorno de Grid, HaoRen et al. propone un algoritmo de transposición de los nodos basado en datos obtenidos a través de una búsqueda. La overlay puede ser un hipercubo *n*-dimensional completo o incompleto. Los identificadores lógicos de los nodos van de G(0) a G(*M*-

1), donde *M* es el número de nodos del hipercubo y G(*i*) es el código de Gray de índice *i* (ver Tabla 1). Cada nodo del hipercubo tiene un único identificador lógico.

Para representar la evolución de la búsqueda que realiza el algoritmo se utiliza una representación en árbol. Los nodos del árbol representan nodos del hipercubo. Cuando un nodo envía la búsqueda a uno de sus nodos vecinos lo indicamos con una flecha que va del nodo emisor al nodo receptor. El algoritmo se ejecuta en varios pasos, siendo el número de flechas que hay hasta el nodo inicial el número de pasos en el que el nodo es accedido.

La figura 16 muestra un ejemplo de búsqueda desde el nodo 010 (nodo inicial) en un hipercubo 3-dimensional completo. Como podemos ver, no todos los nodos deben propagar la petición a todos sus vecinos. De hecho, el nodo inicial es el único nodo que propaga la petición a todos sus vecinos. Para el resto de nodos que no propagan a todos sus vecinos, si el nodo recibe la petición por su vecino en dimensión *i* solo envía la petición a sus vecinos en dimensiones mayores que la dimensión *i*.

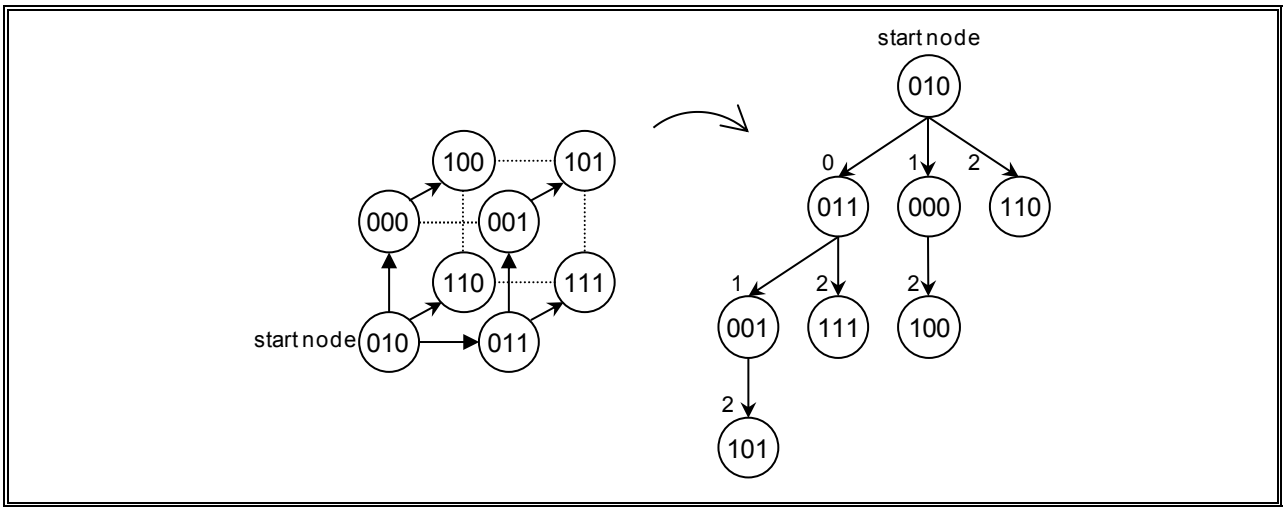

Figura 16: HaoRen et al. - ejemplo de búsqueda (hipercubo completo).

La figura 17 muestra un ejemplo de búsqueda en un hipercubo 3-dimensional incompleto desde el nodo 010 (figura 17.a) (nodo inicial) y desde el nodo 101 (figura 17.b) (nodo inicial). Como podemos ver, no todos los nodos deben propagar la petición a todos sus vecinos. De hecho, si el nodo inicial tiene vecino en la dimensión 0 (como en la figura 17.a), el nodo inicial es el único nodo que propaga la petición a todos sus vecinos y si el nodo inicial no tiene vecino en la dimensión 0 (como en la figura 17.b) es un nodo vecino del nodo inicial el que propaga la petición a todos sus vecinos. Para el resto de nodos que no propagan a todos sus vecinos, si el nodo recibe la petición por su vecino en dimensión *i* solo envía la petición a sus vecinos en dimensiones mayores que la dimensión *i*.

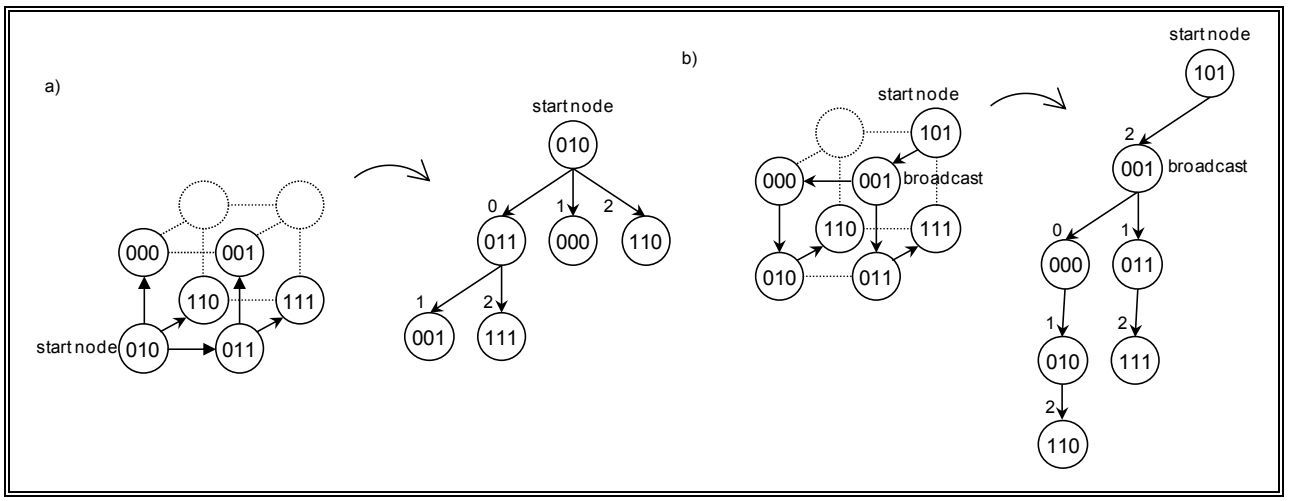

Figura 17: HaoRen et al. - ejemplo de búsqueda (hipercubo incompleto).

3.3.2 Búsqueda en HyperD

En HyperD la overlay siempre es un hipercubo *n*-dimensional completo que puede ser lleno (cuando todos los nodos tienen un único identificador lógico) o parcial (si algún nodo tiene más de un identificador lógico).

Para representar la evolución de la búsqueda que realiza el algoritmo se utiliza una representación en árbol. Los nodos del árbol representan nodos del hipercubo. Cuando un nodo envía la búsqueda a uno de sus nodos vecinos lo indicamos con una flecha que va del nodo emisor al nodo receptor. El algoritmo se ejecuta en varios pasos, siendo el número de flechas que hay hasta el nodo inicial el número de pasos en el que el nodo es accedido.

La figura 18 muestra un ejemplo de búsqueda desde el nodo inicial marcado como (3) cuyos identificadores lógicos son 0010 y 1010 en un hipercubo 4-dimensional parcial. Como podemos ver, no todos los nodos deben propagar la petición a todos sus vecinos. De hecho, el nodo inicial es el único nodo que propaga la petición a todos sus vecinos. Para el resto de nodos que no propagan a todos sus vecinos, si el nodo recibe la petición por su vecino en dimensión *i* solo envía la petición a sus vecinos en dimensiones mayores que la dimensión *i*.

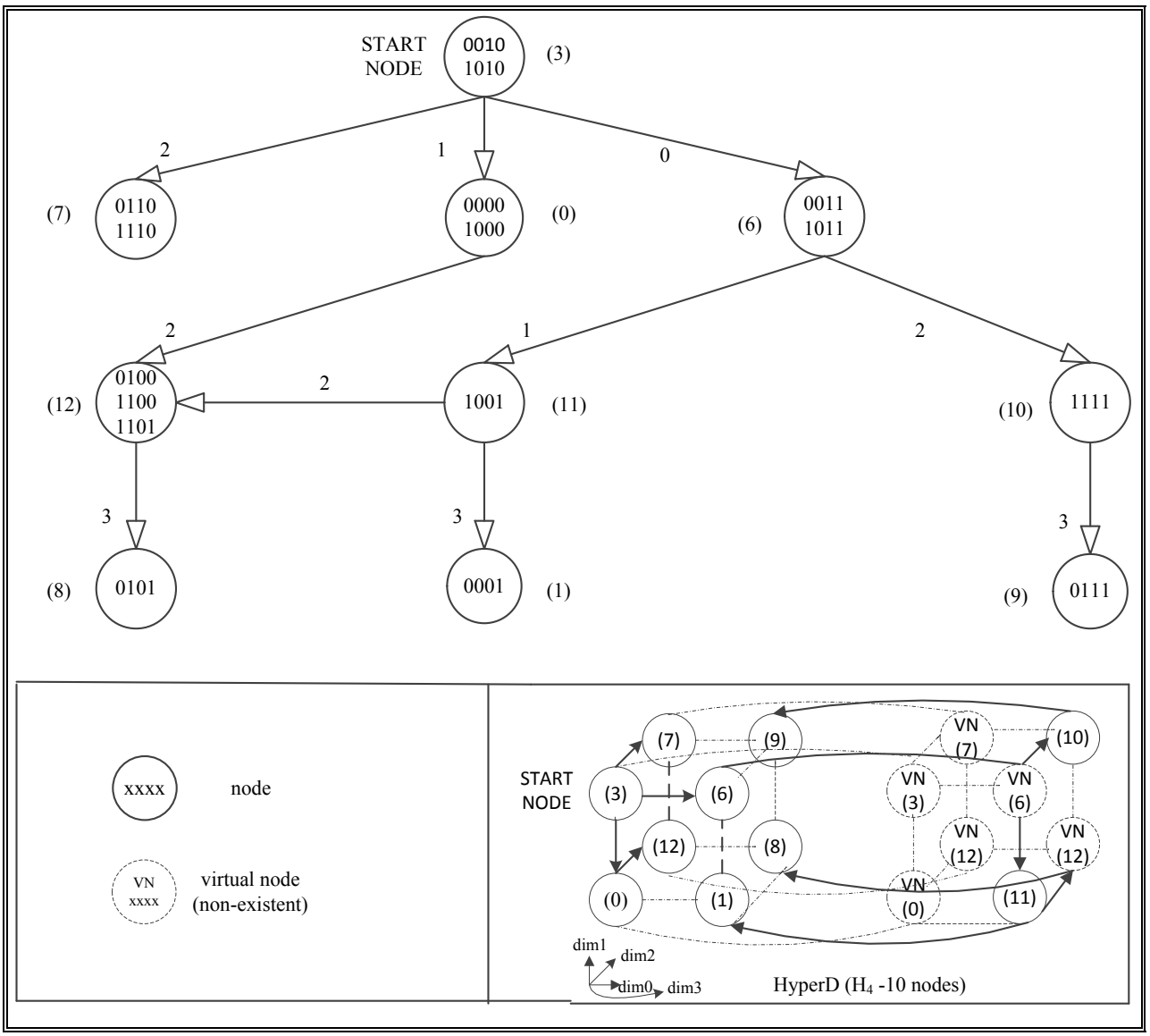

Figura 18: HyperD - ejemplo de búsqueda.

IV. DESCUBRIMIENTO DE SERVICIOS

En un sistema distribuido, antes de que un cliente pueda acceder a un servicio, el cliente tiene que localizar el proveedor del servicio. El objetivo del descubrimiento de servicios es la localización de proveedores que satisfagan las necesidades del cliente, escogiendo entre proveedores candidatos y componiendo, si es necesario, varios servicios en un único servicio. En este trabajo se propone introducir *brokers* especializados en localizar proveedores y por tanto servicios. Un *broker* es responsable de almacenar y mantener información de los servicios que ofrecen sus proveedores y también información de los propios proveedores, por ejemplo, cómo acceder al proveedor.

En el sistema distribuido presentado, los proveedores anuncian sus servicios en componentes llamados *Broker Information System* (BIS) y los clientes usan los BIS para localizar servicios y proveedores. En [43], los *brokers* se clasifican en dos tipos: *forwarding brokers*, donde el *broker* juega el papel de intermediario entre cliente y proveedor para todas las comunicaciones y *handle driven brokers*, donde el *broker* actúa como servidor de nombres que localiza proveedores de servicios retornando la IP del proveedor seleccionado. En este trabajo, el *broker* propuesto pertenece al segundo tipo, ya que las comunicaciones entre cliente y proveedor tienen lugar directamente, sin intermediario, y se inician una vez se ha seleccionado el proveedor del servicio al que accederá el cliente.

Para localizar servicios/proveedores en el sistema distribuido se propone una overlay de BIS. El término overlay lo utilizamos como red virtual construida por encima del nivel de TCP/IP en la capa de aplicación.

4.1 Arquitectura del servicio

La arquitectura propuesta se llama HOverlay. En HOverlay, la interconexión tiene una topología en hipercubo con base $b = 2$. El hipercubo puede ser completo o incompleto.

En esta overlay cada nodo almacena la localización de sus n nodos vecinos, como en cualquier overlay en hipercubo, pero también almacena el estado de estos vecinos. El estado de un vecino puede ser vivo o no-vivo. Un nodo vivo es un BIS que está ejecutándose adecuadamente y un nodo no-vivo es un BIS que no es accesible a través de la red, que está desconectado, que está sobrecargado, etc. Para determinar el estado vivo o no-vivo de sus vecinos cada nodo puede, por ejemplo, enviar cada cierto tiempo un mensaje de ping a cada vecino y si el vecino responde en un cierto tiempo considerar que éste está en estado vivo y si no lo hace que está en estado no-vivo.

Cabe destacar que en un hipercubo n-dimensional incompleto, si un nodo no dispone de vecino en una dimensión, este nodo considera que su vecino en esa dimensión se encuentra en estado no-vivo. Por ejemplo, en la figura 7 el nodo (111) es un nodo no-vivo para H_3 con 5, 6 y 7 nodos.

4.2 Búsqueda en el servicio

En este apartado se presenta el algoritmo de búsqueda de recursos propuesto en esta tesis y llamado Algoritmo-t_{aux} que es tolerante a los fallos que se producen continuamente en los entornos distribuidos de gran escala. En estos entornos, el algoritmo se ejecuta para realizar una búsqueda de recursos que satisface la petición que realiza un cliente del sistema distribuido. El algoritmo se auto-adapta a fallos que encuentran en el sistema, por ejemplo, nodos inaccesibles por red, nodos que no están activos o nodos que están trabajando lentamente, con el objetivo de que estos nodos, que se llamarán nodos no-vivos, no afecten a la búsqueda de recursos y se obtenga una respuesta adecuada independientemente de los fallos que haya en el sistema. Algoritmo-t_{aux} es el resultado de un proceso de mejora que se inició con un algoritmo (presentado primero en este apartado) al que se hace referencia como Algoritmo-v_d y que se mejoró con otro algoritmo (presentado segundo en este apartado) al que se hace referencia como Algoritmo-v_a. Se presentan los algoritmos previos porque cada uno de ellos introduce una nueva mejora al algoritmo final propuesto. Algoritmo-v_d utiliza un vector llamado v_d (vector de dimensiones), Algoritmo-v_a usa el vector v_d anterior y un nuevo vector llamado v_a (vector añadido) y finalmente, Algoritmo-t_{aux} utiliza los vectores v_d y v_a anteriores más una tabla que se llama t_{aux} (tabla auxiliar). Diferenciar cada uno de los algoritmos nos permite cuantificar en el capítulo de evaluaciones la mejora que introduce cada uno de ellos al Algoritmo-t_{aux}.

4.2.1 Algoritmo-v_d

Para presentar el Algoritmo-v_d primero se introduce la idea base del algoritmo, después se detalla un ejemplo completo del algoritmo y finalmente se presenta el diagrama de flujo. En los dos primeros sub-apartados se ha supuesto que ninguno de los nodos tiene el recurso o recursos solicitados para ilustrar mejor la propagación de la búsqueda en el hipercubo. Si algún nodo consultado tuviera el recurso solicitado ese nodo ya no propagaría la petición del cliente. El nombre de Algoritmo-v_d viene dado porque usa un vector llamado v_d (vector de dimensiones).

4.2.1.1 Algoritmo-v_d: representación en árbol

Para representar la evolución de la búsqueda que realiza el algoritmo en la overlay se utiliza una representación en árbol. Los nodos del árbol representan nodos del hipercubo. Cuando un nodo envía la búsqueda de recurso a uno de sus nodos vecinos lo indicamos con una flecha que va del nodo emisor al nodo receptor. El algoritmo se ejecuta en varios pasos, siendo el número de flechas que hay hasta el nodo inicial el número de pasos en el que el nodo es accedido.

En el Algoritmo-v_d cada nodo trabaja con un vector llamado v_d. Este vector le indica a un nodo el conjunto de dimensiones (nodos vecinos) a los que debe propagar el mensaje de búsqueda. Por ejemplo, $v_d = \{2, 1\}$, significa que el nodo debe propagar la búsqueda a través de sus vecinos en las dimensiones 2 y 1 (si es posible).

La figura 19.a muestra un ejemplo de una búsqueda de recursos que se inicia en el nodo 000 dentro de un hipercubo 3-dimensional. En este caso, el hipercubo es completo. El nodo 000 envía en el primer paso la petición a todos sus vecinos, es decir, a los nodos 001, 010 y 100. En el segundo paso, el nodo 001 es consultado y envía la petición a su vecino en la dimensión 1 (011) y a su vecino en la dimensión 2 (101). Como podemos ver, no todos los nodos deben propagar la petición a todos sus vecinos. De hecho, el nodo inicial es el único nodo que propaga la petición a todos sus vecinos.

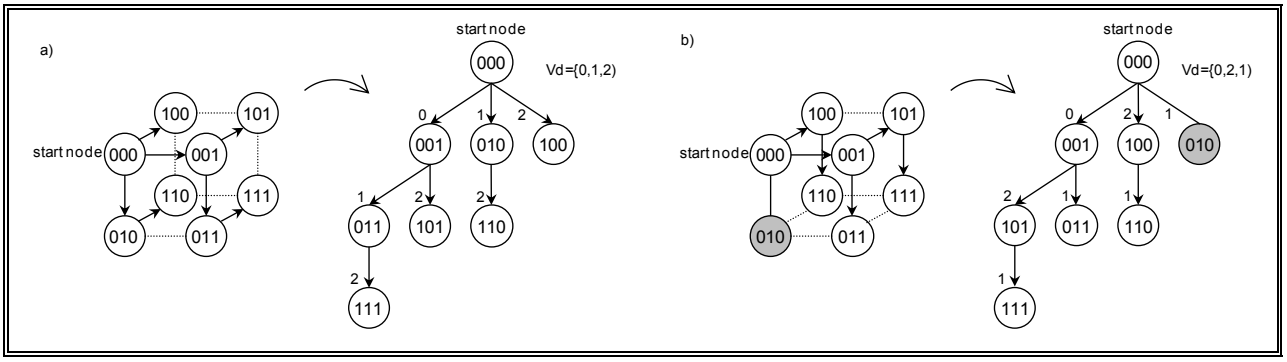

Figura 19: Representación en árbol del Algoritmo-v_d.

En la figura 19.a se ha asumido que el vector v_d es siempre $v_d = \{0, 1, 2\}$ y que un nodo si recibe la petición por su vecino en dimensión *i* solo envía la petición a sus vecinos en dimensiones mayores que la dimensión *i*. Por ejemplo el nodo 001 recibe la petición por su vecino en dimensión 0 y sólo envía la petición a sus vecinos en dimensiones 1 (011) y 2 (101).

En la figura 19.b se introduce un nodo no-vivo (010). Reordenando la lista v_d de $v_d = \{0, 1, 2\}$ a $v_d = \{0, 2, 1\}$, el nodo inicial 000 decide las propagaciones que deben realizar cada uno de sus nodos vecinos y coloca el nodo no-vivo en la última posición de la lista. El objetivo es reducir el efecto del nodo no-vivos en la propagación de la búsqueda de recursos a través del hipercubo. Colocando las dimensiones correspondientes a los vecinos no-vivos en las últimas posiciones del vector v_d conseguimos que los nodos no-vivos tengan menos nodos hijos que los nodos vivos. De esta forma el número de nodos vivos no consultados durante la propagación de la petición de recursos debido a los nodos no-vivos se reduce. Asumiendo que todos los nodos tengan como máximo un vecino no-vivo, cada nodo puede colocar la dimensión de su vecino no-vivo en la última posición del vector v_d, convirtiendo así a su vecino no-vivo en un nodo que no tiene hijos en el árbol. En esas condiciones, los nodos no-vivos (que podrían llegar a ser la mitad de los nodos totales del hipercubo) no afectarían a la propagación de la petición de recursos del resto de nodos y además todos los nodos vivos del hipercubo serían consultados.

4.2.1.2 Algoritmo-v_d: un ejemplo completo

En este apartado, se presenta un ejemplo particular pero detallado del proceso de búsqueda de recursos utilizando el Algoritmo-v_d. La figura 20 muestra la overlay en hipercubo y el árbol de la búsqueda en el hipercubo. En el ejemplo, se asume un hipercubo 3-dimensional completo. Los nodos grises representan nodos vivos y los nodos rayados representan nodos que se encuentran en estado no-vivo. Las flechas de la figura 20 indican cómo se realiza la propagación y el sentido de ésta y las líneas (sin flecha final) indican que la propagación no es posible en esa dimensión.

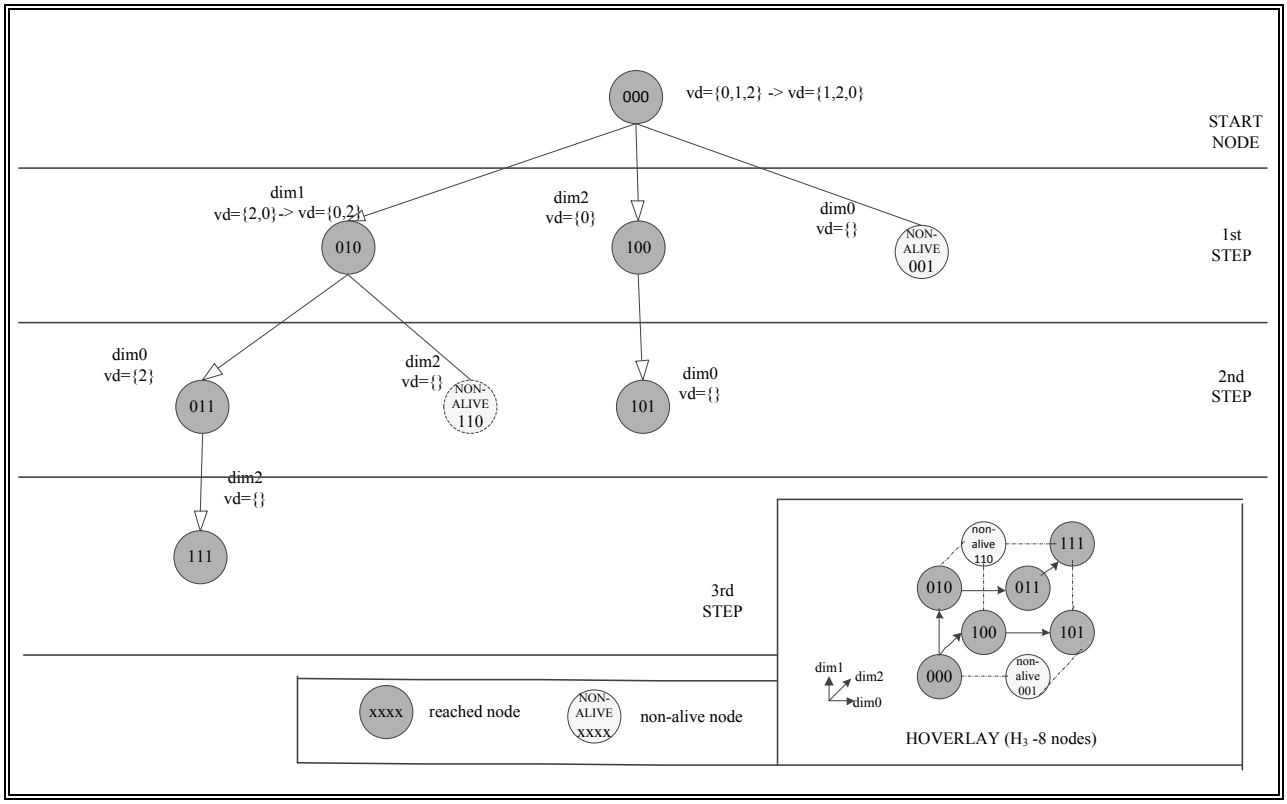

Figura 20: Ejemplo del Algoritmo-v_d en un hipercubo 3-dimensional completo.

En el ejemplo, el valor de la lista v_d en el nodo inicial es {0, 1, 2} y después de chequear el estado de los nodos vecinos en la lista v_d, ésta es reordenada a {1, 2, 0}. En este caso 0 se coloca en la última posición de v_d = {1, 2, 0} porque estamos asumiendo que el vecino del nodo 000 en la dimensión 0 (001) es un nodo no-vivo. El orden de los elementos 1 y 2 en v_d (que representan nodos vivos en el ejemplo) no tiene efectos sobre el resultado final de la propagación.

En el primer paso de tres (justo la dimensión del hipercubo), el vecino del nodo inicial en la dimensión 1 (010) recibe la petición de recurso *P* y la lista v_d = {2, 0} y el vecino en la dimensión 2 (100) recibe *P* and v_d = {0}.

En el segundo paso, fijándonos en el nodo 010, cuando el mensaje compuesto por la petición de recursos *P* junto con la lista v_d = {2, 0} se recibe, v_d se reordena como v_d = {0, 2} ya que asumimos que su vecino en la dimensión 2 (110) es un nodo no-vivo; el vecino 011 recibe v_d = {2}.

Y así para el resto de nodos. En tres pasos todos los nodos del hipercubo 3-dimensional completo se consultan una única vez aunque dos nodos se encuentren en estado no-vivo (001 y 110). Cabe observar que en el ejemplo podríamos haber asumido también que los nodos 111 y 101 son nodos no-vivos sin que el esquema de propagación se vea afectado por estos nodos adicionales no-vivos.

4.2.1.3 Algoritmo-v_d: procedimiento de búsqueda

El procedimiento de búsqueda se inicia cuando un cliente conecta con uno de los nodos del sistema y solicita un recurso (o recursos). El nodo inicial contactado por el cliente, si no dispone del recurso solicitado, inicia una búsqueda. Un mensaje de petición de recursos se compone de la petición que realiza el cliente (*message.request*) y un vector (*message.*v_d). Cada nodo es consultado una única vez. Si el nodo consultado dispone del recurso solicitado ese nodo ya no propaga la petición del cliente ya que se ha encontrado al menos un recurso que satisface la solicitud del cliente. Por el contrario, si el nodo consultado no dispone del recurso solicitado, al no poder asegurar el nodo consultado si en el sistema ya se ha encontrado dicho recurso, el nodo consultado propaga la petición. La figura 21 muestra el diagrama de flujo que se ejecuta en cada nodo del hipercubo al recibir una petición de recurso.

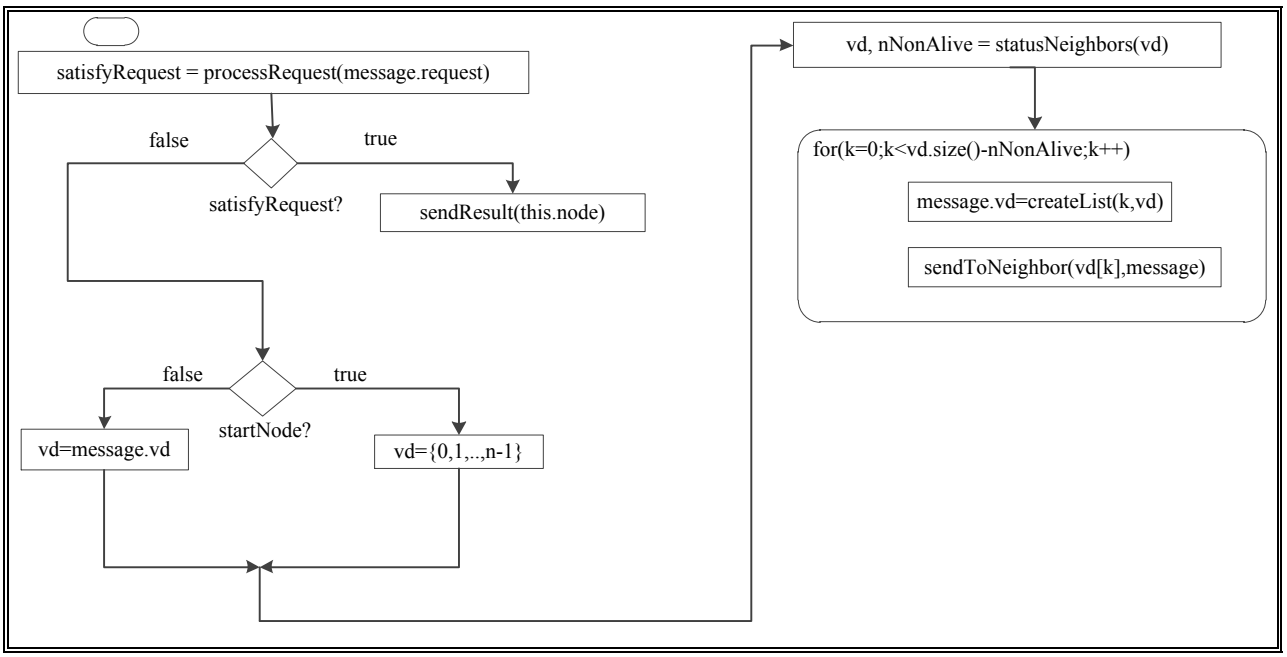

Figura 21: Diagrama de flujo del Algoritmo-v_d.

1) Cuando una petición de recurso se recibe en un nodo la función *processRequest*(*message.request*) se ejecuta. Si la petición incluida en el mensaje puede ser satisfecha por el nodo se establece el valor de *satisfyRequest* a true y la función *sendResult*(*this.node*) se ejecuta para informar al cliente que el recurso o recursos se encuentran en este nodo. En otro caso *satisfyRequest* se establece a *false*.

2) Cuando *satisfyRequest* es *false*, el mensaje compuesto por la petición (message.*request*) y un vector de dimensiones (*message.*v_d) se propaga (si es posible). Si el nodo que recibe el mensaje es el nodo inicial, es decir, el nodo contactado por el cliente (*startNode* es *true*), el vector v_d se inicializa con el valor v_d = {0, 1,.., n-1} (el conjunto completo de

dimensiones del hipercubo). En otro caso el vector v_d se inicializa con el vector recibido en el mensaje de petición de recursos. En ambos casos el vector indica el conjunto de dimensiones (vecinos del nodo) a los que hay que propagar el mensaje. Cabe observar que el nodo inicial propaga la petición a todos sus vecinos.

3) El nodo ejecuta la función *statusNeighbors*(v_d) y reordena el vector v_d de tal forma que las dimensiones de los nodos vecinos en estado no-vivo (que no son accesibles por red, que funcionan lentamente, que no están activos, ect.) se colocan en las últimas posiciones de la lista. Por ejemplo si v_d = {0, 1, 2, 3} y el vecino en la dimensión 1 no contesta, entonces, v_d se reordena como {0, 2, 3, 1}. La función *statusNeighbors*(v_d) también retorna un valor entero llamado *nNonAlive*. El valor entero de *nNonAlive* indica el número de nodos no-vivos que hay en el reordenado vector v_d.

4) Para cada posición k en el vector que representa un nodo vecino en estado vivo, el nodo ejecuta la función *createList*(k, v_d) que genera un nuevo vector compuesto por todas aquellas dimensiones localizadas después de la posición k en el vector ordenado y actualiza v_d con el valor del nuevo vector. En otras palabras, si el número de elementos en v_d (v_d.*size*()) es q la función *createList*(k, v_d) retorna { $v_d[k+1]$, ..., $v_d[q-1]$}. Por ejemplo, si v_d = {0, 2, 3, 1} y k = 1, la llamada a *createList*(k=1, v_d) devolverá {3, 1}. El nuevo vector y la petición se envían al vecino vivo correspondiente en la dimensión $v_d[k]$ a través de la función *sendToNeighbor*($v_d[k]$,*message*).

4.2.2 Algoritmo-v_a

Para presentar el Algoritmo-v_a primero se introduce la idea base del algoritmo, después se detalla un ejemplo completo del algoritmo y finalmente se presenta el diagrama de flujo. En los dos primeros sub-apartados se ha supuesto que ninguno de los nodos tiene el recurso o recursos solicitados para ilustrar mejor la propagación de la búsqueda en el hipercubo. Si algún nodo consultado tuviera el recurso solicitado ese nodo ya no propagaría la petición del cliente. El nombre de Algoritmo-v_a viene dado porque además del vector v_d (vector de dimensiones) que usa Algoritmo-v_d, se utiliza también un vector llamado v_a (vector añadido).

4.2.2.1 Algoritmo-v_a: representación en árbol

Como en el algoritmo anterior, Algoritmo-v_d, una representación en árbol se utiliza para mostrar el proceso de búsqueda del Algoritmo-v_a. Los nodos del árbol representan nodos del hipercubo. Cuando un nodo envía la búsqueda de recurso a uno de sus nodos vecinos se indica con una flecha que va del nodo emisor al nodo receptor. Cuando un nodo no puede ser accedido por encontrarse el nodo que debería enviarle el mensaje de petición de recursos (que llamaremos nodo padre) en estado no-vivo se indica con una línea discontinua. El algoritmo se ejecuta en varios pasos siendo el número de flechas que hay hasta el nodo inicial el número de pasos en el que el nodo es accedido.

El Algoritmo-v_a se basa en la misma idea base que el Algoritmo-v_d mostrada de nuevo en la figura 22.a y 22.b pero además añade una nueva idea ilustrada por la figura 22.c. Las figuras 22.a y 22.b han sido explicadas en el apartado 3.3.3.1.1. En la figura 22.c se muestra un ejemplo donde el nodo inicial tiene dos vecinos no-vivos: el vecino en la dimensión 1 (010) y el vecino en la dimensión 2 (100). Dada esta situación, el Algoritmo-v_a consulta al nodo 110 a través del nodo 111 en el cuarto paso. Cabe notar que el nodo 110 no es consultado en el segundo paso ya que el nodo 010 es un nodo no-vivo.

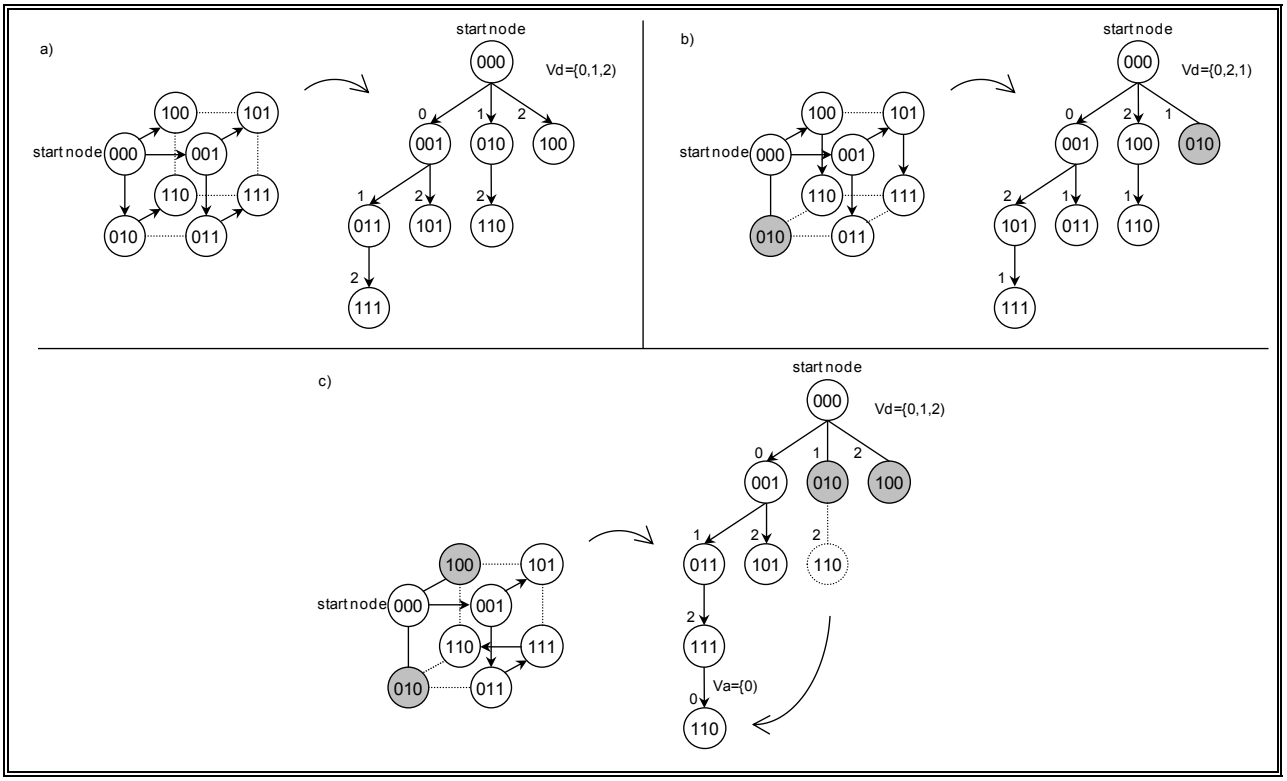

Figura 22: Representación en árbol del Algoritmo-v_a.

En el Algoritmo-v_a cada nodo trabaja con dos vectores, llamados v_d y v_a, Estos vectores son establecidos por el nodo padre de la propagación. La lista v_d se reordena como en el Algoritmo-v_d para reducir el efecto de los nodos no-vivos en la propagación de la petición de recursos del cliente en el hipercubo. La lista v_a se utiliza para consultar nodos hijos de vecinos no-vivos utilizando un camino alternativo. Este nuevo camino utiliza sólo nodos del hipercubo no consultados antes. Así los nodos son consultados como máximo una vez y no se requieren mensajes adicionales.

4.2.2.2 Algoritmo-v_a: un ejemplo completo

Este apartado presenta un ejemplo de búsqueda de recursos utilizando el Algoritmo-v_a. La figura 23 muestra la overlay en hipercubo y la representación en árbol del proceso de búsqueda.

La overlay en este caso está formada por un hipercubo 4-dimensional incompleto que contiene 12 nodos. Los nodos grises representan nodos vivos, los nodos rayados representan nodos existentes que se encuentran en estado no-vivo y los nodos blancos con línea discontinua representan nodos que no existen en el sistema porque el hipercubo es incompleto y por tanto no contiene todos los nodos. El algoritmo propuesto no distingue entre nodos no-vivos y nodos no-presentes en el hipercubo, por lo que ambos tipos de nodos se marcan como nodos no-vivos.

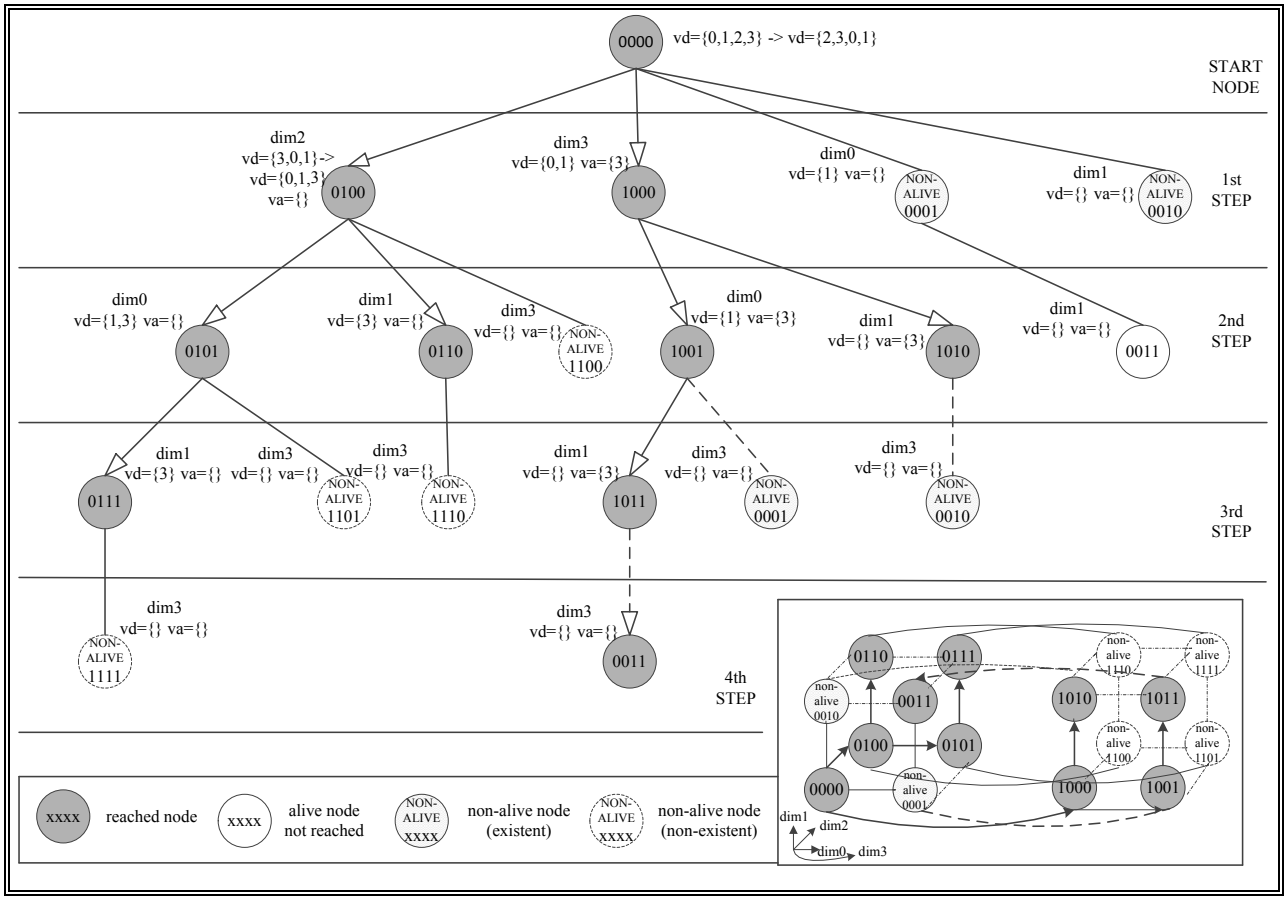

Figura 23: Ejemplo del Algoritmo-v_a en un hipercubo 4-dimensional incompleto.

Una petición de servicio o servicios *P* se inicia en el nodo 0000 de un hipercubo 4-dimensional. En el primer paso de este ejemplo, el valor del vector v_d en el nodo inicial es {0, 1, 2, 3} y después de que el nodo inicial chequee el estado de sus vecinos, el vector v_d se reordena como {2, 3, 0, 1}. En este caso, 0 y 1 se colocan en las dos últimas posiciones de v_d porque en el ejemplo asumimos que los nodos vecinos en las dimensiones 0 (0001) y 1 (0010) son nodos no-vivos. La petición se propagará a través de los nodos vivos del nodo inicial pero antes un nuevo vector v_d se genera para cada uno de éstos nodos vecinos vivos. El nuevo vector v_d se crea eliminado la dimensión usada en la propagación y aquellas dimensiones de las posiciones previas a la dimensión usada en la propagación. Por ejemplo, para el nodo vecino del nodo inicial (0000) en la dimensión 3 (1000) se genera el vector v_d = {0, 1} ya que v_d = {2, 3, 0, 1}.

El nodo inicial tiene dos vecinos no-vivos y por tanto uno de ellos tiene un hijo en el árbol de propagación (0011). Para consultar al nodo 0011 el algoritmo intenta encontrar un camino alternativo a través del último nodo vivo del vector; el vecino en la dimensión 3 (1000) en este caso. Por este motivo, la dimensión 3 se añade al vector v_a del nodo 1000 (inicialmente el vector v_a es un vector vacío). Los vectores generados para cada uno de los nodos vivos del nodo inicial, v_d y v_a, se envían junto con la petición a cada uno de estos nodos (0100 y 1000) en el primer paso.

En los siguientes pasos, cada nodo recibe la petición, v_d y v_a de su nodo padre y realiza el mismo procedimiento que el nodo inicial, es decir, reordena v_d y añade nuevas dimensiones a v_a dependiendo del estado de los vecinos a los que debe propagar la petición del cliente. En el ejemplo, el nodo 0100 recibe $v_d = \{3, 0, 1\}$ y reordena como $v_d = \{0, 1, 3\}$ ya que su vecino en la dimensión 3 (1100) es un nodo no-vivo. Cada nodo envía la petición a sus vecinos en las dimensiones de v_d después de generar sus vectores v_d y v_a. Además, si un nodo recibe un vector v_a no vacío, el nodo reenvía la petición a cada uno de sus vecinos en las dimensiones que indica v_a sólo si el vecino en esa dimensión no es su nodo padre. Recordemos que su nodo padre es el nodo que le ha enviado el mensaje de petición de recursos. En el ejemplo, el nodo 1000 recibe $v_a = \{3\}$ de su vecino en la dimensión 3 y éste nodo es su nodo padre (0000). Cuando la propagación se realiza a vecinos en las dimensiones marcadas por v_a, la petición se envía junto con vectores v_d y v_a vacíos ya que esos nodos no deben realizar ninguna propagación más.

Las flechas en la figura 23 indican cómo se realiza la propagación de la petición a través del hipercubo y líneas (sin flecha) indican que la propagación no puede realizarse en esa dimensión. Las líneas con trazado continuo indican que esa propagación se realiza con la información del vector v_d y la líneas con trazado discontinuo significan que esa propagación se realiza con la información del vector v_a.

En cinco pasos todos los nodos vivos del ejemplo son consultados una única vez. Cabe notar que si el nodo 0001 no hubiera estado en estado no-vivo (nodo con relleno rayado), el nodo 0011 (nodo con relleno en blanco) se habría consultado en el segundo paso del algoritmo a través del vecino en la dimensión 1 del nodo 0001 utilizando el vector v_d, pero el algoritmo es capaz de consultar a este nodo en el cuarto paso utilizando el vector v_a a través del vecino en la dimensión 3 del nodo 1011.

4.2.2.3 Algoritmo-v_a: procedimiento de búsqueda

El procedimiento de búsqueda se inicia cuando un cliente conecta con uno de los nodos del sistema y solicita un recurso (o recursos). Si el nodo inicial contactado por el cliente no dispone del recurso solicitado inicia una búsqueda que propaga un mensaje de petición de recursos por el hipercubo. Un mensaje de petición de recursos se compone de la petición que realiza el cliente (*message.request*) y dos vectores (*message*.v_d y *message*.v_a). Cada nodo del hipercubo es consultado una única vez. Si el nodo consultado dispone del recurso solicitado ese nodo ya no propaga la petición del cliente ya que se ha encontrado al menos un recurso que satisface la solicitud del cliente. Por el contrario, si el nodo consultado no dispone del recurso solicitado, al

no poder asegurar el nodo consultado si en el sistema ya se ha encontrado dicho recurso, el nodo consultado propaga la petición. La figura 24 muestra el diagrama de flujo que se ejecuta en un nodo del hipercubo cuando llega una petición de recurso.

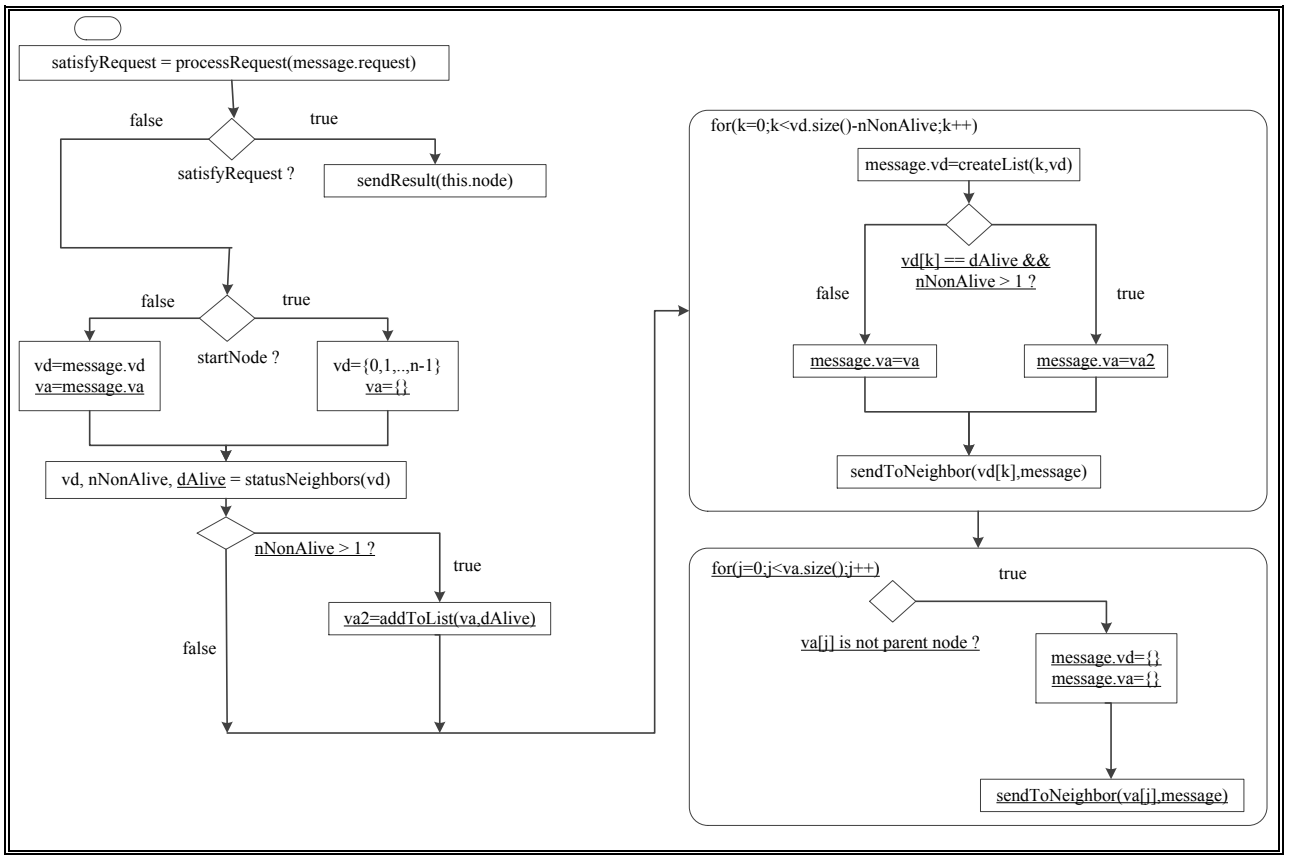

Figura 24: Diagrama de flujo del Algoritmo-v_a. Subrayados aparecen resaltados los cambios con respecto al diagrama de flujo del Algoritmo-v_d.

1) Cuando un nodo recibe un mensaje de petición de recurso (o recursos) se ejecuta la función *processRequest*(*message.request*). Si la petición no puede ser satisfecha por el nodo el valor de *satisfyRequest* se establece a *false*. En otro caso, el valor de *satisfyRequest* se establece a *true* y la función *sendResult*(*this.node*) se ejecuta para informar al cliente de que la petición puede satisfacerse en este nodo.

2) Si *satisfyRequest* es *false* y el nodo que procesa la petición es el nodo inicial contactado por el cliente (*startNode* es *true*), el vector v_d se inicializa como $v_d = \{0, 1, \ldots, n\text{-}1\}$ (el conjunto de dimensiones del hipercubo) y v_a se inicializa como $v_a = \{\}$ (un vector vacío). En otro caso (*startNode* is *false*) v_d y v_a se inicializan respectivamente con los vectores recibidos en el mensaje de petición de recursos.

3) El nodo ejecuta la función *statusNeighbors*(v_d) y reordena el vector v_d de tal forma que las dimensiones correspondientes a los nodos vecinos no-vivos se colocan en las últimas posiciones del vector. Por ejemplo, si v_d = {0, 1, 2, 3} y el vecino en la dimensión 1 es no-vivo entonces v_d se reordena como {0, 2, 3, 1}. La función *statusNeighbors*(v_d) también retorna dos valores enteros *nNonAlive* y *dAlive*. El valor entero de *nNonAlive* es el valor del número de nodos no-vivos del vector reordenado v_d. El valor entero de *dAlive* contiene la dimensión del último vecino vivo almacenada en v_d. Por ejemplo, si v_d = {2, 1, 0, 3} y los vecinos del nodo en las dimensiones 0 y 3 son nodos no-vivos, *nNonAlive* = 2 y *dAlive* = 1. El valor de *nNonAlive* se utilizará para enviar mensajes de petición de recursos sólo a los nodos vecinos vivos de un nodo y el valor de *dAlive* se utilizará para consultar nodos que se consultarían si no hubiera nodos no-vivos en la overlay.

4) Si el número de vecinos no-vivos (*nNonAlive*) es mayor que uno, el nodo ejecuta la función *addToList*(v_a, *dAlive*). Esta función añade *dAlive* al final del vector v_a y retorna un nuevo vector (v_{a2}) que se utilizará, si es necesario, más adelante en lugar de v_a.

5) Para cada posición k en el vector v_d que representa un nodo vecino vivo, se ejecuta la función *createList*(k, v_d) que crea un nuevo vector compuesto de todas las dimensiones que se encuentran después de la posición k en el vector ordenado v_d y se actualiza v_d con este nuevo vector. En otras palabras, si el número de elementos en v_d (v_d.*size*()) es q la función retorna {$v_d[k+1]$, ..., $v_d[q-1]$}. Por ejemplo, si v_d = {0, 2, 3, 1} y k = 1, la llamada a *createList*(k, v_d) devuelve {3, 1}. Para todos los nodos vecinos vivos, excepto para el último vecino vivo, el vector v_a toma el valor del vector v_a recibido en el mensaje de petición de recursos. Para el último nodo vecino vivo (v_d[k] = *dAlive*) y si el número de nodos vecinos no vivos es mayor que 1, el vector v_a toma el valor del vector v_{a2} generado en el paso anterior. La petición de recursos que realizó el cliente, el vector v_d y el vector v_a se envían al nodo vecino correspondiente en la dimensión $v_d[k]$ a través de la función *sendToNeigbor*($v_d[k]$, *message*).

6) Finalmente, el nodo también propaga la petición a cada uno de sus vecinos en las dimensiones marcadas por v_a ejecutando la función *sendToNeigbor*($v_a[j]$, *message*) sólo si el vecino en la dimensión $v_a[j]$ no es su nodo padre. Recordamos que su nodo padre es el nodo que le ha enviado el mensaje de petición de recursos. Los vectoress v_d y v_a, incluidos en este mensaje son vectores vacíos ya que los nodos que recibirán este mensaje de petición de recursos no propagan el mensaje a ninguno de sus nodos vecinos.

4.2.3 Algoritmo-t_{aux}

Para presentar el algoritmo final que se propone en este documento primero se introduce la idea base del algoritmo, después se detalla un ejemplo completo del algoritmo y finalmente se presenta el diagrama de flujo. En los dos primeros sub-apartados se ha supuesto que ninguno de los nodos tiene el recurso o recursos solicitados para ilustrar mejor la propagación de la búsqueda en el hipercubo. Si algún nodo consultado tuviera el recurso solicitado ese nodo ya no propagaría

la petición del cliente. El nombre de Algoritmo-t_{aux} viene dado porque además de los vectores v_d (vector de dimensiones) y v_a (vector añadido) que usa Algoritmo-v_a se utiliza también una tabla llamada t_{aux} (tabla auxiliar).

4.2.3.1 Algoritmo-t_{aux}: representación en árbol

Como en los algoritmos anteriores, Algoritmo-v_d y Algoritmo-v_a, una representación en árbol se utiliza para mostrar el proceso de búsqueda del Algoritmo-t_{aux}. Los nodos del árbol representan nodos del hipercubo. Cuando un nodo envía el mensaje de petición de recursos a uno de sus nodos vecinos se indica con una flecha que va del nodo emisor al nodo receptor. Cuando un nodo no puede ser accedido por encontrarse su nodo padre en estado no-vivo se indica con una línea discontinua. El algoritmo se ejecuta en varios pasos, siendo el número de flechas que hay hasta el nodo inicial el número de paso en el que el nodo es accedido.

El Algoritmo-t_{aux} se basa en las ideas base de Algoritmo-v_d (mostradas de nuevo en las figuras 25.a y 25.b) y de Algoritmo-v_a (mostrada de nuevo en la figura 25.c) pero además añade una nueva idea ilustrada por la figura 25.d. Las figuras 25.a y 25.b han sido explicadas en el apartado 3.3.3.1.1. La figura 25.c ha sido explicada en el apartado 3.3.3.2.1. En la figura 25.d se muestra un ejemplo donde el nodo inicial debe enviar el mensaje de petición de recursos a tres vecinos no-vivos: el vecino en la dimensión 0 (001), el vecino en la dimensión 1 (010) y el vecino en la dimensión 2 (100). Dada esta situación, el nodo inicial (000) envía el mensaje de petición al último nodo al que se accedería si sus vecinos no estuvieran no-vivos (111). En la figura puede verse cómo a través de este nodo (111) se consultan los mismos nodos que a través del nodo inicial (000).

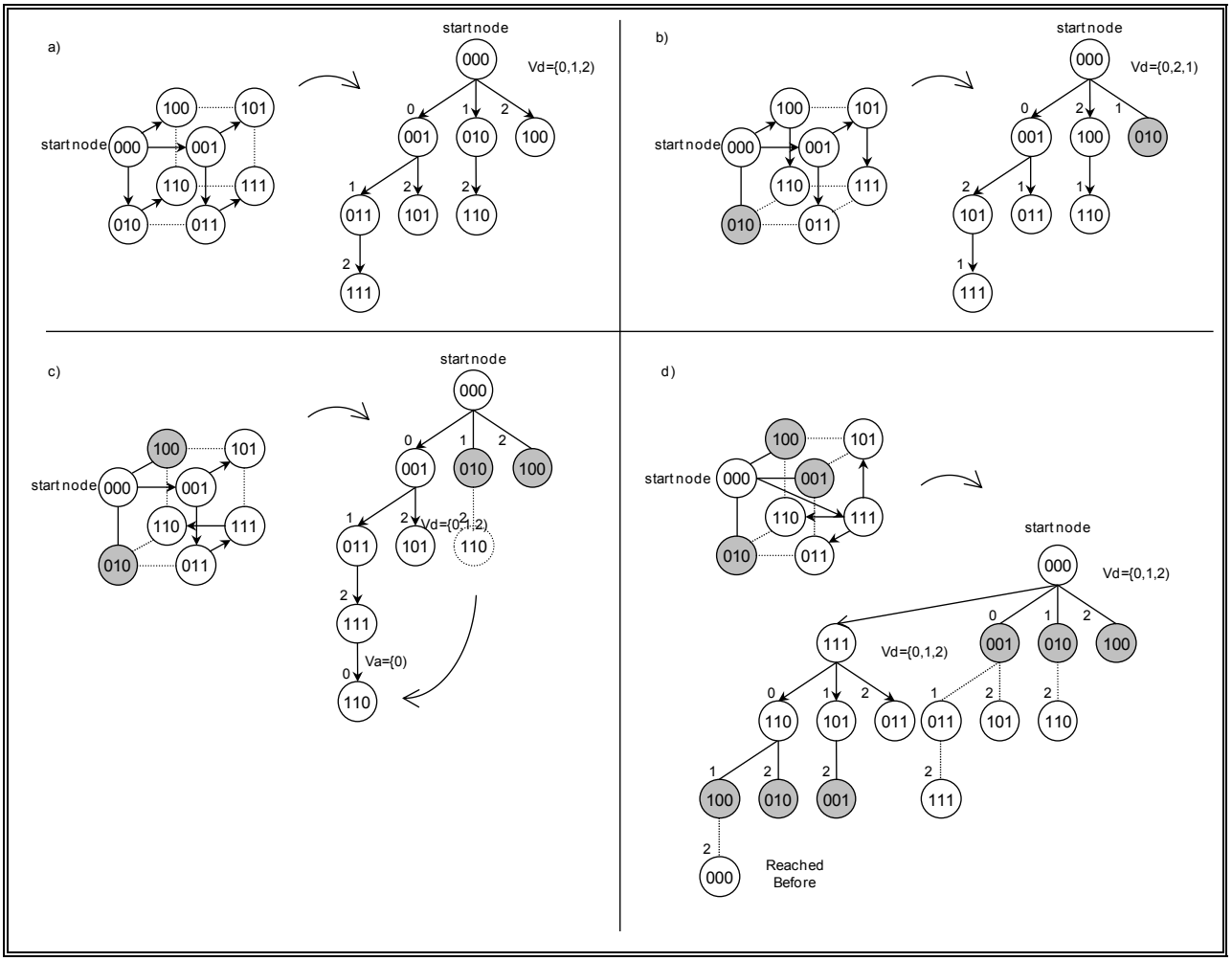

Figura 25: Representación en árbol del Algoritmo-t_{aux}.

En el Algoritmo-t_{aux} cada nodo además de la tabla de vecinos dispone de una tabla llamada t_{aux} que contiene información de nodos que no son vecinos. La forma en que se rellena t_{aux} se mostrará más adelante. El Algoritmo-t_{aux} trabaja con tres vectores, llamados v_d, v_a y w_l. Los vectores v_d y v_a funcionan como en el Algoritmo-v_a. El vector w_l se utiliza para rellenar las tablas t_{aux} que tienen los nodos y en los siguientes sub-apartados se muestra cómo se usa. Los vectores son establecidos por el nodo padre. El vector v_d se reordena como en el Algoritmo-v_d para reducir el efecto de los nodos no-vivos en la propagación del mensaje de petición de recursos del cliente en el hipercubo. El vector v_a se utiliza como en el Algoritmo-v_a para consultar nodos utilizando un camino alternativo. La tabla t_{aux} se utiliza sólo cuando todos los nodos vecinos por los que se debe propagar el mensaje están en estado no-vivo. Este nuevo camino utiliza sólo nodos del hipercubo no consultados antes. Así los nodos son consultados como máximo una vez.

4.2.3.2 Algoritmo-t_{aux}: un ejemplo completo

Este apartado presenta un ejemplo de búsqueda de recursos utilizando el Algoritmo-t_{aux}. La figura 26 muestra la overlay en hipercubo, la representación en árbol del proceso de búsqueda y

cómo se añade la información de un nodo a una tabla t_{aux}. La overlay en este caso está formada por un hipercubo 4-dimensional incompleto que contiene 12 nodos. Los nodos grises representan nodos vivos consultados, los nodos rayados representan nodos existentes que se encuentran en estado no-vivo y los nodos blancos con línea discontinua representan nodos que no existen en el sistema porque el hipercubo es incompleto y por tanto no contiene todos los nodos. El algoritmo propuesto no distingue entre nodos no-vivos y nodos que no existen en el hipercubo, por lo que ambos tipos de nodos se marcan como nodos no-vivos.

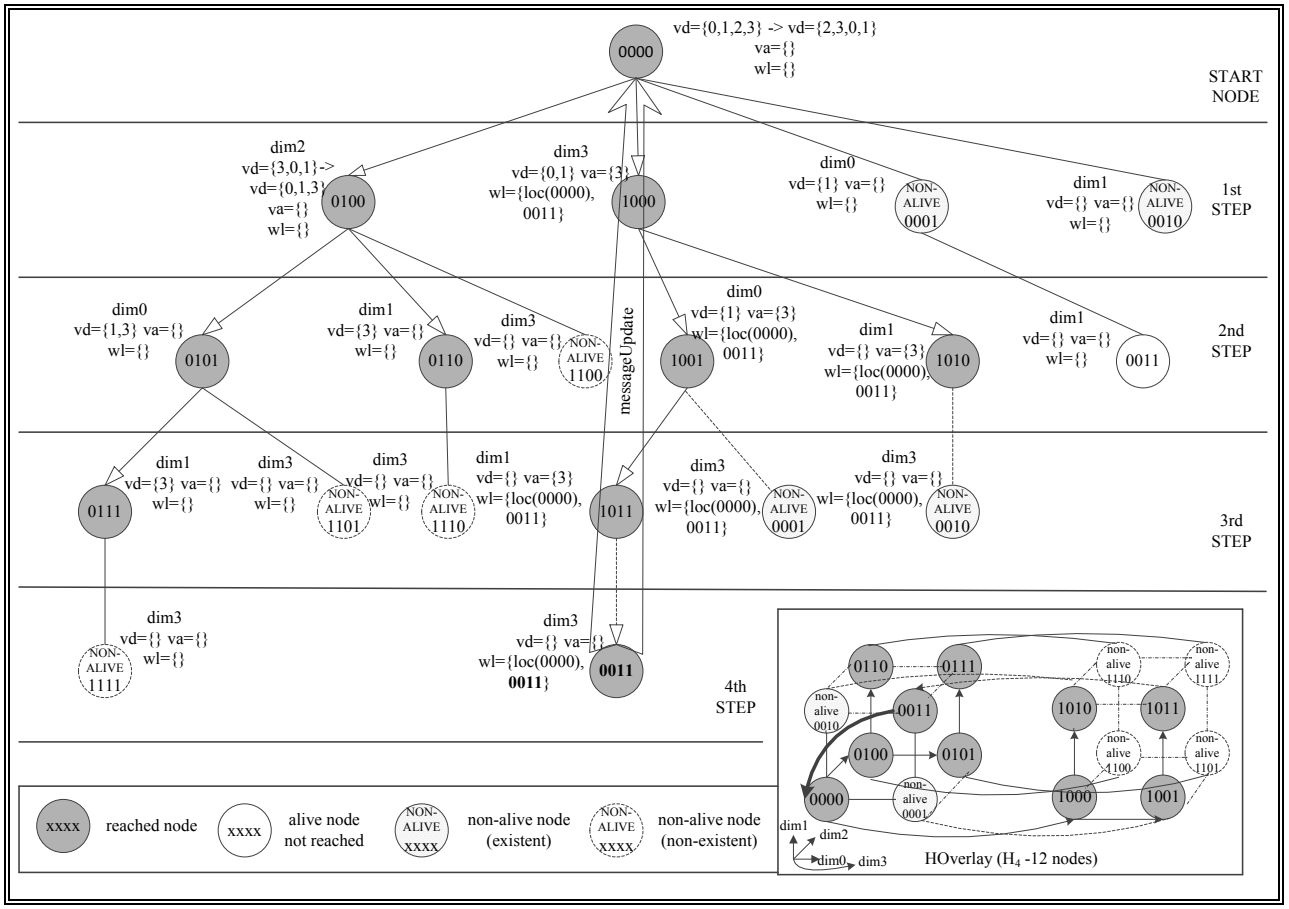

Figura 26: Ejemplo del Algoritmo-t_{aux} en un hipercubo 4-dimensional incompleto.

Una petición de recurso o recursos *P* se inicia en el nodo 0000 de un hipercubo 4-dimensional. En el primer paso de este ejemplo, el valor del vector v_d en el nodo inicial es {0, 1, 2, 3} y después de que el nodo inicial chequee el estado de sus vecinos, el vector v_d se reordena como {2, 3, 0, 1}. En este caso, 0 y 1 se colocan en las dos últimas posiciones de v_d porque en el ejemplo asumimos que los nodos vecinos en las dimensiones 0 (0001) y 1 (0010) son nodos no-vivos. La petición se propaga a través de los nodos vivos del nodo inicial pero antes un vector v_d se genera para cada uno de éstos nodos vecinos vivos. Estos vectores v_d se crean eliminado la dimensión usada en la propagación y aquellas dimensiones de las posiciones previas a la

dimensión usada en la propagación. Por ejemplo, para el nodo vecino del nodo inicial (0000) en la dimensión 3 (1000) se genera la lista v_d = {0, 1} ya que v_d = {2, 3, 0, 1}.

El nodo inicial tiene dos vecinos no-vivos y por tanto uno de ellos tiene un hijo en el árbol de propagación (0011). Para consultar al nodo 0011 el algoritmo intenta encontrar un camino alternativo a través del último nodo vivo del vector; el vecino en la dimensión 3 (1000) en este caso. Por este motivo, la dimensión 3 se añade al vector v_a del nodo 1000 (inicialmente v_a es un vector vacío).

Como el nodo inicial tiene dos vecinos no-vivos, la información de un nodo debería añadirse a la tabla t_{aux} del nodo inicial (si es posible). Por este motivo, el nodo inicial calcula cuál es el último nodo en el árbol de propagación al que se accede a través de v_a (0011) y añade al vector w_l del nodo 1000 su localización y el identificador calculado (w_l = {loc(0000), 0011}). La localización del nodo 0000 (loc(0000)) es una IP y un puerto donde el nodo está escuchando mensajes que le pueden llegar. Inicialmente el vector w_l es un vector vacío. Los vectores generados para cada uno de los nodos vivos del nodo inicial, v_d, v_a y w_l se envían junto con la petición a cada uno de estos nodos (0100 y 1000) en el primer paso.

Los valores w_l = {loc(0000), 0011} hacen que si el mensaje de petición de recursos llega hasta el nodo 0011, éste nodo envíe su localización al nodo 0000, a través de un mensaje que en la figura 26 se llama *messageUpdate*. Cuando el nodo 0000 reciba este mensaje añadirá la localización del nodo 0011 en su tabla t_{aux}. Así una tabla t_{aux} de un nodo contiene el identificador de un nodo del hipercubo que no es vecino suyo y cómo contactar con ese nodo (la IP del nodo con ese identificador y un puerto donde está escuchando).

Las flechas en la figura 26 indican cómo se realiza la propagación de la petición a través del hipercubo y líneas (sin flecha) indican que la propagación no puede realizarse en esa dimensión. Las líneas con trazado continuo indican que esa propagación se realiza con la información del vector v_d y la líneas con trazado discontinuo significan que esa propagación se realiza con la información del vector v_a.

Como ya hemos dicho anteriormente la tabla t_{aux} sólo se utiliza cuando todos los vecinos por los que hay que propagar están en estado no-vivo. Así, Algoritmo-t_{aux} permite continuar la búsqueda cuando el hipercubo se segmenta. Cabe notar que cuando esta situación se produce en los primeros pasos de la búsqueda, gran parte del hipercubo no es consultado si se usan los algoritmos previos a Algoritmo-t_{aux}. Como ejemplo de uso de la tabla t_{aux}, la figura 27 muestra el caso de mensaje de una petición de recursos iniciada en el nodo 1000, que llega al nodo 0000 y que debe propagarse a través de tres vecinos no-vivos en un hipercubo 4-dimensional.

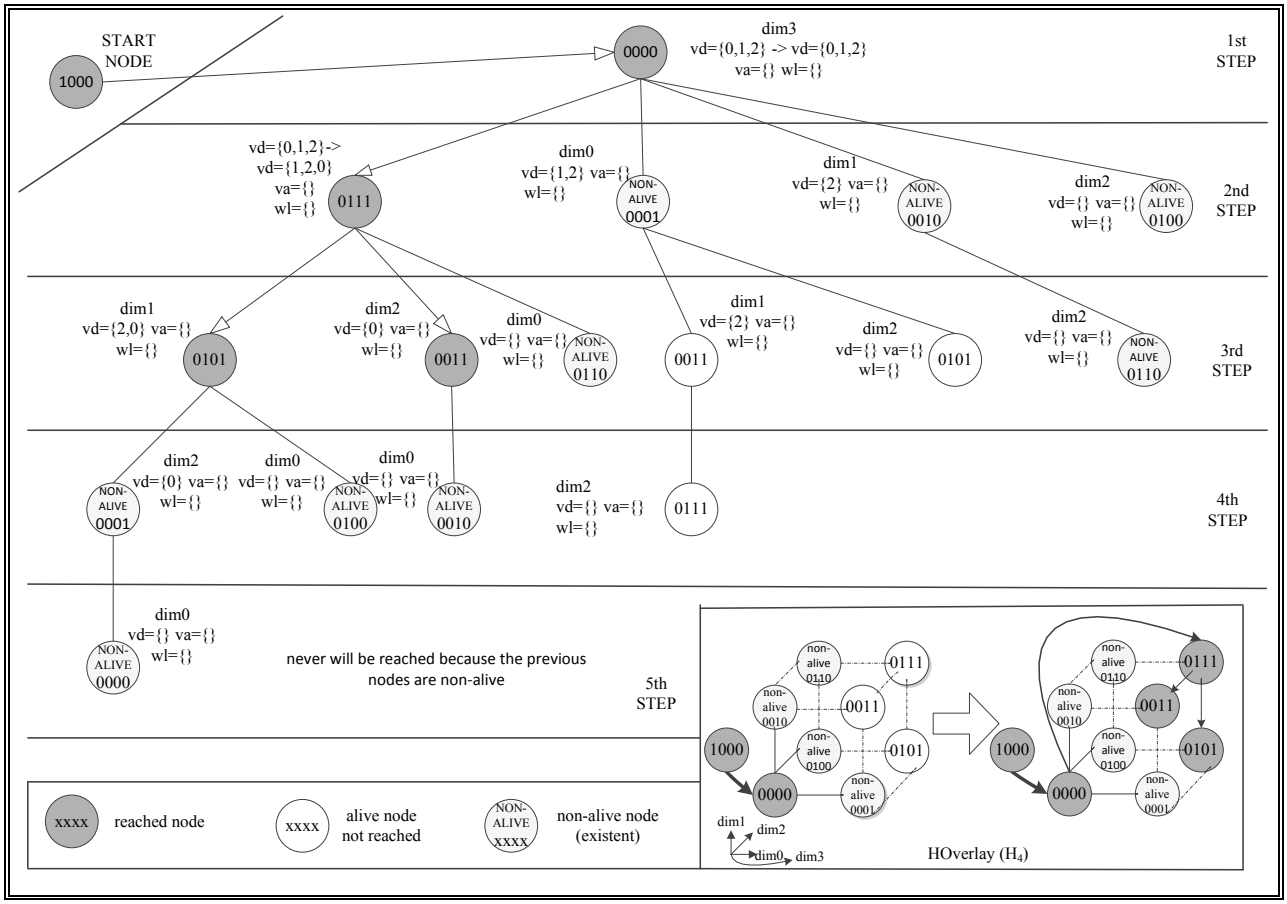

Figura 27: Ejemplo de uso de la tabla t_{aux} en el Algoritmo-t_{aux}.

Una petición de recurso o recursos *P* llega al nodo 0000 con los valores v_d = {0, 1, 2}, v_a = {} y w_l = {}. Como los vecinos en las dimensiones 0 (0001), 1 (0010) y 2 (0100) del nodo están en estado no-vivo, se calcula el último nodo al que se accedería utilizando v_d (vecino en la dimensión 2 del vecino en la dimensión 1 del vecino en la dimensión 0 del nodo 0000, es decir, el nodo 0111) y se mira en la tabla t_{aux} si tenemos información de la localización del nodo calculado (0111). En el ejemplo suponemos que sí, por lo que se envía al nodo calculado los valores de v_d, v_a y w_l recibidos. El proceso de búsqueda en este nodo (0111) se trata como cualquier otro mensaje de petición de recursos. Por ejemplo, el nodo 0111 reordena el vector v_d = {0, 1, 2} recibido a v_d = {1, 2, 0} ya que su vecino en la dimensión 0 (0110) está en estado no-vivo.

Cabe notar que la figura 27 no muestra la búsqueda completa en el hipercubo. Para el ejemplo mostrado en la figura, la búsqueda se inicia en el nodo 1000 y sólo se muestra la propagación de *P* por su vecino 0000. Algoritmo-t_{aux}, en estas condiciones, permite consultar 4 nodos más que sus algoritmos previos en un hipercubo formado por 16 nodos.

Cabe notar también que, en la misma figura, el ejemplo muestra el uso de la tabla t_{aux} en el primer paso de la búsqueda, pero que puede producirse en cualquiera de los pasos de Algoritmo-t_{aux}.

4.2.3.3 Algoritmo-t_{aux}: procedimiento de búsqueda

El procedimiento de búsqueda se inicia cuando un cliente busca un recurso o recursos en el sistema. El cliente contacta con uno de los nodos del hipercubo que llamamos nodo inicial (*start node*). Si *start node* no puede satisfacer la petición del cliente se inicia una búsqueda por el hipercubo. La figura 28 muestra el diagrama de flujo que se ejecuta en un nodo cuando recibe un mensaje de petición de recursos.

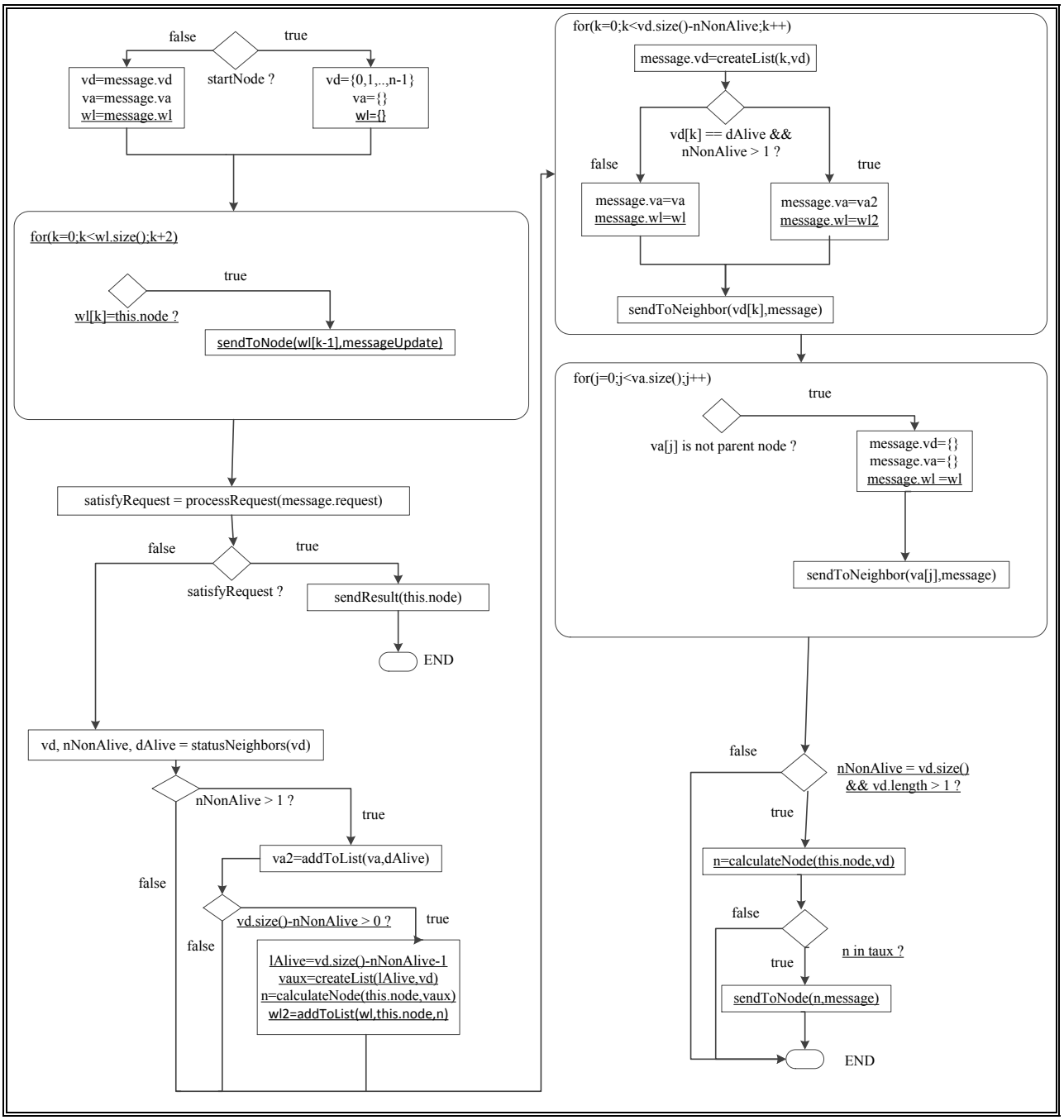

Figura 28: Diagrama de flujo del Algoritmo-t_{aux}. Subrayados aparecen resaltados los cambios con respecto al diagrama de flujo del Algoritmo-v_a.

Un mensaje de petición de recursos se compone de la petición que realiza el cliente (*message.reques*t) y tres vectores (*message.*v_d, *message.*v_a y *message.*w_l).

1) Si el nodo que procesa la petición es el nodo inicial contactado por el cliente (*startNode* es *true*), el vector v_d se inicializa como v_d = {0, 1,…, *n*-1} (el conjunto de dimensiones del hipercubo), v_a se inicializa como v_a = {} (un vector vacío) y v_l = {} (un vector vacío). En otro caso (*startNode* es *false*) v_d, v_a y v_l se inicializan respectivamente con los vectores recibidos en el mensaje de petición de recursos.

2) Si el identificador del nodo que ha recibido el mensaje se encuentra en alguna o algunas de las posiciones impares del vector w_l ($w_l[k]$ = *this.node* siendo *k* un índice inicializado a 0 que recorre las posiciones impares del vector) la función *sendToNode* se ejecuta para enviar un mensaje de tipo *messageUpdate*. La localización (IP y puerto) del nodo al que se enviará el mensaje de tipo *messageUpdate* se encuentra en $w_l[k\text{-}1]$. Por ejemplo, si el nodo con identificador 0011 recibe w_l = {loc(0000), 0011} enviará un mensaje de tipo *messageUpdate* (que contiene su IP y puerto) al nodo 0000. La IP y puerto del nodo 0000 se encuentran en loc(0000). Esta información al ser recibida por el nodo 0000 se utilizará para actualizar su tabla t_{aux}.

3) El nodo que ha recibido el mensaje ejecuta la función *processRequest*(*message.request*). Si la petición puede ser satisfecha por el nodo, el valor de *satisfyRequest* se establece a *true* y la función *sendResult*(*this.node*) se ejecuta para informar al cliente de que la petición puede satisfacerse en este nodo. Si la petición no puede ser satisfecha por el nodo el valor de *satisfyRequest* se establece a *false* y se continúa la propagación del mensaje por el hipercubo (si es posible). Así, cuando el nodo consultado dispone del recurso solicitado ese nodo ya no propaga la petición del cliente ya que se ha encontrado al menos un recurso que satisface la solicitud del cliente. Por el contrario, cuando el nodo consultado no dispone del recurso solicitado, al no poder asegurar el nodo consultado si en el sistema ya se ha encontrado dicho recurso, el nodo consultado propaga la petición.

4) El nodo ejecuta la función *statusNeighbors*(v_d) y reordena el vector v_d de tal forma que las dimensiones correspondientes a los nodos vecinos no-vivos se colocan en las últimas posiciones del vector. Por ejemplo, si v_d = {0, 1, 2, 3} y el vecino en la dimensión 1 está en estado no-vivo entonces v_d se reordena como {0, 2, 3, 1}. La función *statusNeighbors*(v_d) también retorna dos valores enteros *nNonAlive* y *dAlive*. El valor entero de *nNonAlive* es el valor del número de nodos no-vivos del vector reordenado v_d. El valor entero de *dAlive* contiene la dimensión del último vecino vivo almacenada en v_d. Por ejemplo,si v_d = {2, 1, 0, 3} y los vecinos del nodo en las dimensiones 0 y 3 son nodos no-vivos, *nNonAlive* = 2 y *dAlive* = 1. El valor de *nNonAlive* se utilizará para enviar mensajes de petición de recursos sólo a los nodos vecinos vivos de un nodo y el valor de *dAlive* se utilizará para consultar nodos que se consultarían si no hubiera nodos no-vivos en la overlay

5) Si el número de vecinos no-vivos (*nNonAlive*) es mayor que uno, el nodo ejecuta la función *addToList*(v_a, *dAlive*). Esta función añade *dAlive* al final de la lista v_a y retorna una nueva lista (v_{a2}), que se utilizará, si es necesario, más adelante en lugar de v_a.

6) Si el número de vecinos no-vivos (*nNonAlive*) es mayor que uno y el tamaño de v_d es mayor que *nNonAlive* se crea una lista auxiliar (v_{aux}) compuesta de todas las dimensiones de vecinos no-vivos en el vector reordenado v_d. La función *createList*(*lAlive*, v_d) crea un vector compuesto de todas las dimensiones que se encuentran después de la posición *lAlive* del vector reordenado v_d. Por ejemplo, si v_d = {2, 3, 0, 1} y *nNonAlive* = 2 entonces *lAlive* = 1 y v_{aux} = {0, 1}. Una vez obtenido el vector auxiliar v_{aux} se ejecuta la función *calculateNode*(*this.node*, v_{aux}) que devuelve el nodo *n* que se alcanzaría a partir del nodo que ha recibido la petición (*this.node*) recorriendo las dimensiones del vector v_{aux}. Por ejemplo, siguiendo con el ejemplo anterior, si *this.node* es el nodo con identificador 0000, *calculateNode* devolvería el nodo *n* = 0011 (vecino en la dimensión 1 del vecino en la dimensión 0 del nodo 0000). Finalmente en este paso se ejecuta la función *addtoList*(w_l, *loc(this.node)*, *n*) que añade en este orden al vector w_l la localización del nodo que ha recibido el mensaje de petición de recursos (como *loc(this.node)*) y el nodo *n* calculado.

7) Para cada posición *k* en el vector v_d que representa un nodo vecino vivo, se ejecuta la función *createList*(*k*, v_d) que crea un nuevo vector compuesto de todas las dimensiones que se encuentran después de la posición *k* en el vector reordenado v_d y se actualiza v_d con este nuevo vector. En otras palabras, si el número de elementos en v_d (v_d.size()) es *q* la función retorna {v_d[k+1], ..., v_d[q-1]}. Por ejemplo, si v_d = {0, 2, 3, 1} y *k* = 1, la llamada a *createList*(*k*, v_d) devuelve {3, 1}. Sólo para el último vecino vivo (v_d[k] = *dAlive*) y si *nNonAlive* es mayor que 1, el vector v_a toma el valor de v_{a2} generado anteriormente, y el vector w_l el valor de w_{l2} también generado en un paso anterior. Para el resto de nodos vivos, se mantienen para los vectores v_a y w_l los datos recibidos en el mensaje de petición de recursos. La petición de recurso, y los vectores v_d, v_a y w_l se envían al vecino correspondiente en la dimensión v_d[*k*] a través de la función *sendToNeigbor*(v_d[*k*], *message*).

8) En este punto el nodo también intenta propagar la petición a cada uno de sus vecinos en las dimensiones marcadas por v_a ejecutando la función *sendToNeigbor*(v_a[*j*], *message*) sólo si el vecino en la dimensión v_a[*j*] no es el nodo que le envió el mensaje (es decir, no es su nodo padre). Los vectores v_d, v_a y w_l, incluidos en este mensaje son vectores vacíos ya que los nodos que recibirán este mensaje no deben reenviar el mensaje a ningún vecino.

9) Finalmente, si el número de nodos no vivos es igual al tamaño de v_d y mayor que 1, no se ha podido propagar el mensaje a los vecinos de v_d y, en este caso, se utiliza la información de la tabla t_{aux} para intentar propagar el mensaje. Utilizando la función *calculateNode*(*this.node*, v_d) se obtiene el último nodo *n* que debería alcanzarse propagando el mensaje desde el nodo que ha recibido el mensaje (*this.node*). Por ejemplo, si el nodo

que ha recibido el mensaje es el nodo 0000 y v_d = {0, 2, 3}, *calculateNode* devolverá n = 1101 (vecino en la dimensión 3 del vecino en la dimensión 2 del vecino en la dimensión 0 del nodo 0000). Si en la tabla t_{aux} se tiene la localización del nodo n se propaga el mensaje de petición de recursos al nodo n utilizando la función *sendToNode*(*n*, *message*). El mensaje enviado al nodo n es el mensaje recibido en el nodo *this*.*node*.

.

V. EVALUACIONES

En este capítulo se describen las herramientas utilizadas en la evaluación del algoritmo de búsqueda de recursos presentado en esta tesis y se describen las evaluaciones realizadas. Con respecto a las herramientas utilizadas, se detalla para cada herramienta, para qué se ha utilizado y se exponen el o los motivos por los que algunas herramientas han sido desestimadas o utilizadas después de unas primeras pruebas iniciales. Con respecto a las evaluaciones realizadas se describe el experimento, se muestran los resultados obtenidos y se discuten éstos. Para referirnos a Algoritmo-v_d, Algoritmo-v_a, Algoritmo-t_{aux} se utilizan los términos v_d, v_a y t_{aux} cuando en el contexto queda claro que no nos referimos a los vectores v_d, v_a o a la tabla t_{aux}

Cabe notar que si bien el algoritmo propuesto en esta tesis es t_{aux}, al evaluar t_{aux} al mismo tiempo se han evaluado los algoritmos previos v_d y v_a. Dado que Algoritmo-t_{aux} introduce una mejora a Algoritmo-v_a y que Algoritmo-v_a introduce otra mejora a Algoritmo-v_d al evaluar en las mismas condiciones los tres algoritmos se pueden cuantificar las contribuciones que cada una de las mejoras introducen al algoritmo final propuesto.

5.1 Herramientas utilizadas

En este apartado se describen las herramientas utilizadas en las evaluaciones de los algoritmos de búsqueda de recursos presentados en esta tesis. Se detallará, para cada herramienta, para qué se ha utilizado y se aprovechará el capítulo para exponer el o los motivos por los que algunas herramientas han sido desestimadas después de unas primeras pruebas iniciales.

5.1.1 GridSim

GridSim [45] [46] es un toolkit programado íntegramente en Java que permite modelar y simular la ejecución de procesos y tareas en entornos de computación distribuida. En el proceso de simulación se tiene en cuenta a los usuarios, a las aplicaciones, a los recursos y a los planificadores de servicios, para diseñar y evaluar el rendimiento y la eficiencia de un determinado algoritmo de planificación, topología de red o política de asignación de recursos. La última versión de GridSim es del 2010, por lo que parece que el proyecto ya no continúa.

Destacamos de la arquitectura de GridSim que se articula encima de SimJava [47]. SimJava está programado en Java y GridSim lo utiliza como capa de más bajo nivel para manejar los eventos y las interacciones entre diferentes entidades, clases o instancias a objetos que se definen en GridSim.

En la figura 29 podemos ver la arquitectura de GridSim. Todos los componentes de GridSim se comunican entre sí a través del paso de mensajes, definido en la primera capa por SimJava. La segunda capa modela los elementos de la infraestructura distribuida o recursos del entorno de Grid como son clusters, repositorios, etc. La tercera y cuarta capa se encargan de modelar y simular los servicios específicos de Grids computacionales y de Grid de datos respectivamente. Entre esta tercera y cuarta capa se definen servicios de intercambio de información entre las dos capas como el *Grid Information Service* que dispone de información de los recursos disponibles. La última capa, *User Code*, contiene los componentes que permiten a los usuarios interaccionar con el entorno de Grid simulado.

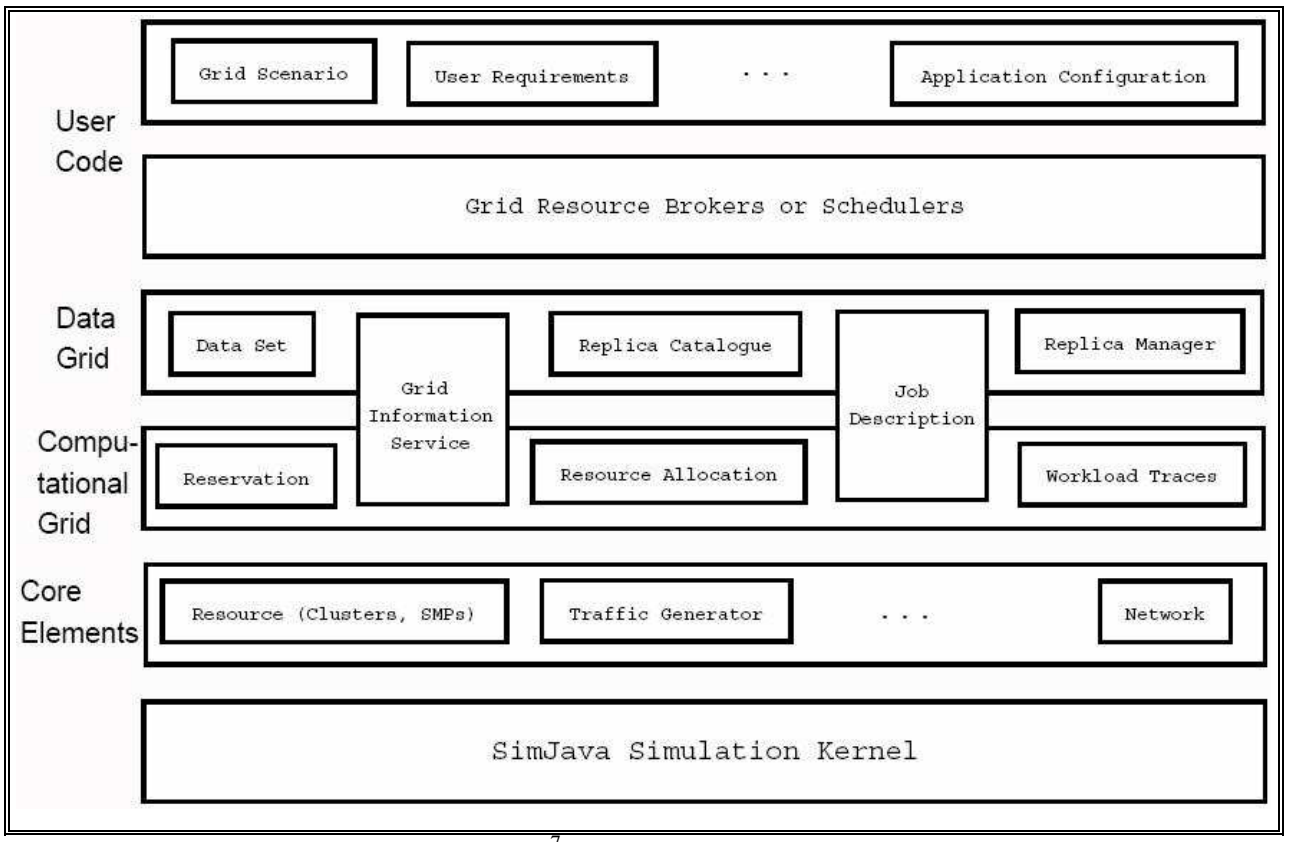

Figura 29: Arquitectura de GridSim[7]

Las funcionalidades principales de la herramienta GridSim son las siguientes:

- Permite incorporar caídas de los recursos del Grid en tiempo de ejecución.
- Permite definir diferentes políticas de planificación de ejecución de tareas y generar nuevas políticas de planificación.
- Permite interconectar recursos y otras entidades en una topología de red.
- Permite realizar simulaciones donde el tráfico de red está basado en distribuciones probabilísticas.
- Permite crear algoritmos de encaminamiento dentro del entorno de Grid.

[7] Fuente: http://www.cct.lsu.edu/~dsk/eScience2007Posters/sulistio.html#1

El entorno de desarrollo generado con GridSim forma parte de un proyecto final de carrera de l'Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels (Universitat Politècnica de Catalunya – BarcelonaTech), realizado por Andreu Pere Isern Deyà.

En el proyecto anteriormente referenciado GridSim presentó algunas complicaciones. La deficiencia más importante es su baja escalabilidad, que no permite realizar simulaciones del orden de nodos que se requieren en la presente tesis (cientos, miles o millones), ya que a medida que el número de nodos de la simulación aumenta, la memoria RAM consumida crece exponencialmente. Así, esta herramienta no ha podido ser utilizada en la mayoría de las evaluaciones de la presente tesis y sólo ha podido ser utilizada para verificar el funcionamiento de los algoritmos en ejemplos concretos con muy pocos nodos.

5.1.2 Aplicaciones propias

Una vez desestimado el uso de GridSim como simulador por no permitir realizar simulaciones de entornos distribuidos de gran escala, entendiendo por gran escala, como ya se ha dicho anteriormente, sistemas formados por cientos, miles o millones de máquinas, se opta por generar aplicaciones propias que permitan minimizar el uso de recursos de la máquina donde se ejecutan las simulaciones. El elevado número de nodos ha requerido algunas simulaciones costosas en tiempo, por ejemplo del orden de semanas.

La elección del lenguaje de programación de tales aplicaciones se basa en que sea un lenguaje multiplataforma que permita ejecutar las simulaciones en cualquier máquina. Así, el lenguaje utilizado para las aplicaciones propias de esta tesis ha sido Java.

Las aplicaciones propias generadas han sido un simulador que permite evaluar en un sistema distribuido simulado métricas de flexibilidad estática de algoritmos que se ejecuten sobre una overlay con topología en hipercubo y un simulador que permite en un sistema emulado distribuido con topología en hipercubo medir tiempos de búsqueda y efectividad.

5.1.3 PlanetLab

PlanetLab es una red de investigación global que permite desarrollar nuevos servicios. Desde sus inicios en 2003, más de 1000 investigadores de las principales instituciones académicas y laboratorios industriales de investigación han utilizado PlanetLab para desarrollar nuevas tecnologías para el almacenamiento distribuido, mapeo de red, sistemas P2P, DHTs, y el procesamiento de consultas. En la presente tesis se ha utilizado PlanetLab como infraestructura que simula un entorno distribuido formado por máquinas geográficamente dispersas y sobre las cuales se han desplegado aplicaciones propias de la presente tesis.

5.2 Evaluación de escalabilidad

En sistemas distribuidos de gran escala la escalabilidad es una propiedad necesaria ya que cuando un sistema distribuido tiene esta propiedad, al aumentar de tamaño no se degrada considerablemente su funcionamiento y calidad habituales. En este apartado se describe la simulación realizada para evaluar la escalabilidad del los algoritmos presentados en esta tesis También en este capítulo se muestran y discuten los resultados.

5.2.1 Descripción del experimento

En esta simulación se ha evaluado la escalabilidad de los algoritmos v_d, v_a y t_{aux}. Las HOverlays consideradas han estado formadas entre 153 nodos y 29.491 nodos. Para ello, se han generado hipercubos *n*-dimensionales incompletos de dimensión 8 a 15, ocupados en un 60%, 70%, 80% y 90%. No se han considerado ocupaciones iguales o menores al 50% porque un hipercubo *n*-dimensional incompleto ocupado en un 50% o menos es un hipercubo (*n*-1)-dimensional incompleto.

En la simulación se ha evaluado cada HOverlay con una probabilidad de fallo (P_f) del 30% y con una probabilidad de satisfacer el recurso en los nodos (P_{sr}) del 25%, 50% y 75%. P_f puede interpretarse como el porcentaje de nodos no-vivos que pueden encontrarse en la overlay. P_{sr} puede interpretarse como el porcentaje de nodos vivos que disponen del servicio que el cliente solicita. Se han establecido estos valores considerando condiciones para las búsquedas con alta probabilidad de fallo y recursos "poco presentes", "bastante presentes" y "muy presentes" en el sistema distribuido.

Con P_f a 30%, para cada dimensión, para cada % de ocupación y para cada P_{sr}, se han ordenado aleatoriamente todos los nodos vivos del hipercubo *n*-dimensional incompleto y se ha ejecutado una búsqueda de servicio desde cada uno de estos nodos siguiendo ese orden para cada uno de los algoritmos evaluados. Después de esta primera iteración por cada uno de los nodos vivos, se ha ejecutado una segunda iteración de una búsqueda de servicio desde cada uno de esos nodos vivos siguiendo el mismo orden que en la primera iteración para cada uno de los algoritmos evaluados. Finalmente, para cada algoritmo evaluado, se ha calculado el porcentaje de la media de nodos vivos consultados respecto el total de nodos vivos. También se ha calculado el porcentaje de éxito para cada uno de los algoritmos, es decir, en cuantas búsquedas desde nodos vivos de HOverlay se ha encontrado el servicio solicitado.

Cabe notar que calcular los porcentajes anteriores en la primera iteración o en la segunda iteración para los algoritmos v_d y v_a es indistinto, se obtienen los mismos resultados y que en el objetivo de la primera iteración de búsquedas en el experimento sólo ha sido asegurar que las tablas t_{aux} de Algoritmo-t_{aux} no se encuentren vacías o prácticamente vacías en las búsquedas desde los primeros nodos vivos de la iteración, ya que en ese caso Algoritmo-t_{aux} es equivalente a Algoritmo-v_a.

Cabe notar también que en Algoritmo-t_{aux} la mejora en el porcentaje de la media de nodos vivos consultados respecto el total de nodos vivos con una única iteración o con dos es muy pequeña. Por ejemplo, en el experimento realizado se ha obtenido para el caso concreto del hipercubo *n*-dimensional incompleto de dimensión 15 ocupado en un 60%, un porcentaje del 85,97% con una iteración y un porcentaje del 85,95% con dos iteraciones. Finalmente para Algoritmo-t_{aux} no hay ninguna mejora en el porcentaje de éxito ya que es un 100% con una única iteración y con dos iteraciones.

5.2.2 Datos e interpretación del experimento

La figura 30 muestra la media de nodos vivos consultados en % respecto al total de nodos vivos en función de P_{sr} y del % de ocupación. El % de ocupación del hipercubo *n*-dimensional

incompleto (% occupp.) mostrado es 60%, 70%, 80% y 90% (filas) y el P_{sr} es 25%, 50% y 75% (columnas). Para cada una de las gráficas mostradas, la unidad en el eje ordenadas es %. El eje de abscisas indica la dimensión del hipercubo incompleto. Los valores exactos obtenidos se encuentran en el Anexo A.

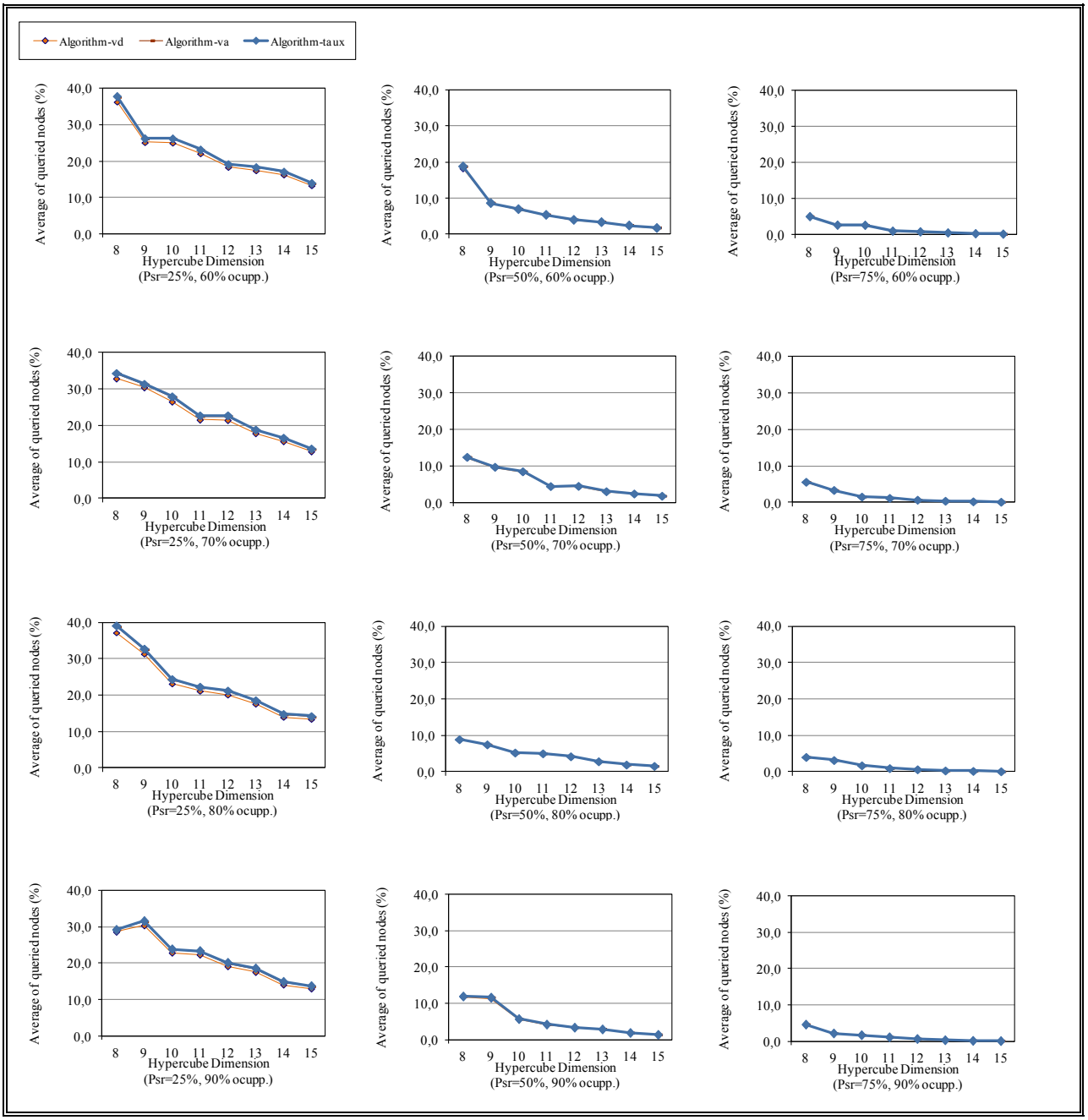

Figura 30: Escalabilidad-Porcentaje de la media de nodos vivos consultados (P_f=30%).

Fijado un % de ocupación (una fila), vemos que al aumentar P_{sr} disminuye el % de nodos consultados. La explicación de este resultado se encuentra en que, al aumentar P_{sr}, aumenta el número de nodos vivos que satisfacen la petición del cliente y, por tanto, se encuentra el nodo

que satisface esta petición antes, con lo que la búsqueda deja de propagarse por el hipercubo *n*-dimensional incompleto antes para una P_{sr} mayor.

Estableciendo una P_{sr} (una columna), no se ve ninguna relación con la ocupación del hipercubo *n*-dimensional incompleto.

Para cada gráfica de la figura 30, no se aprecian casi nunca diferencias para los tres algoritmos evaluados, pero en el Anexo A sí se observa menor % de nodos vivos consultados para v_d, después le sigue v_a y, finalmente, el % es igual al anterior o algo mayor para t_{aux} (aunque los valores difieren poco expresados en %). Por ejemplo, en la dimensión 15, para P_{sr}=25% y 70% es 12,85% para v_d, 13,48% para v_a y 13,62% para t_{aux} y en la misma dimensión 15, para P_{sr}=50% y 70% es 1,71% para v_d, 1,75% para v_a y 1,75% para t_{aux}.

Para la mayoría de las gráficas vemos que al aumentar la dimensión del hipercubo disminuye el % de nodos vivos consultados. La explicación de este resultado se encuentra en el hecho de que, al aumentar la dimensión, manteniendo el % y P_{sr}, aumenta el número total de nodos vivos que satisfacen el servicio que solicita el cliente y, por tanto, es más probable encontrar un nodo que satisfaga la petición, dejándose de propagar la búsqueda por el hipercubo incompleto antes para dimensiones mayores. Para las pocas ocasiones en que lo anterior no se cumple, hay que tener en cuenta que la ubicación de los nodos que satisfacen el servicio que solicita el cliente es aleatoria y por tanto, puede darse el caso de que haya que consultar bastantes nodos para encontrar uno que satisfaga la búsqueda del cliente.

En la figura 31 se muestra el % de veces (efectividad) que un servicio se encontró para las búsquedas realizadas desde nodos vivos. Para todas las gráficas la efectividad es del 100%, es decir, siempre se encuentra el servicio solicitado por el cliente.

Como conclusión obtenemos que el Algoritmo-t_{aux} es escalable, ya que fijada una P_{sr}, al aumentar la dimensión del hipercubo disminuye el porcentaje de nodos vivos consultados. Además, el algoritmo propuesto es escalable en términos de almacenamiento de datos, ya que un nodo sólo mantiene información de sus vecinos y no hay ningún nodo en el sistema que tenga información de todos los nodos. También, las búsquedas pueden iniciarse desde cualquier nodo del sistema distribuido y como mínimo para las P_{sr} superiores al 25% siempre se encuentra el servicio buscado aunque fallen el 30% de los nodos del sistema.

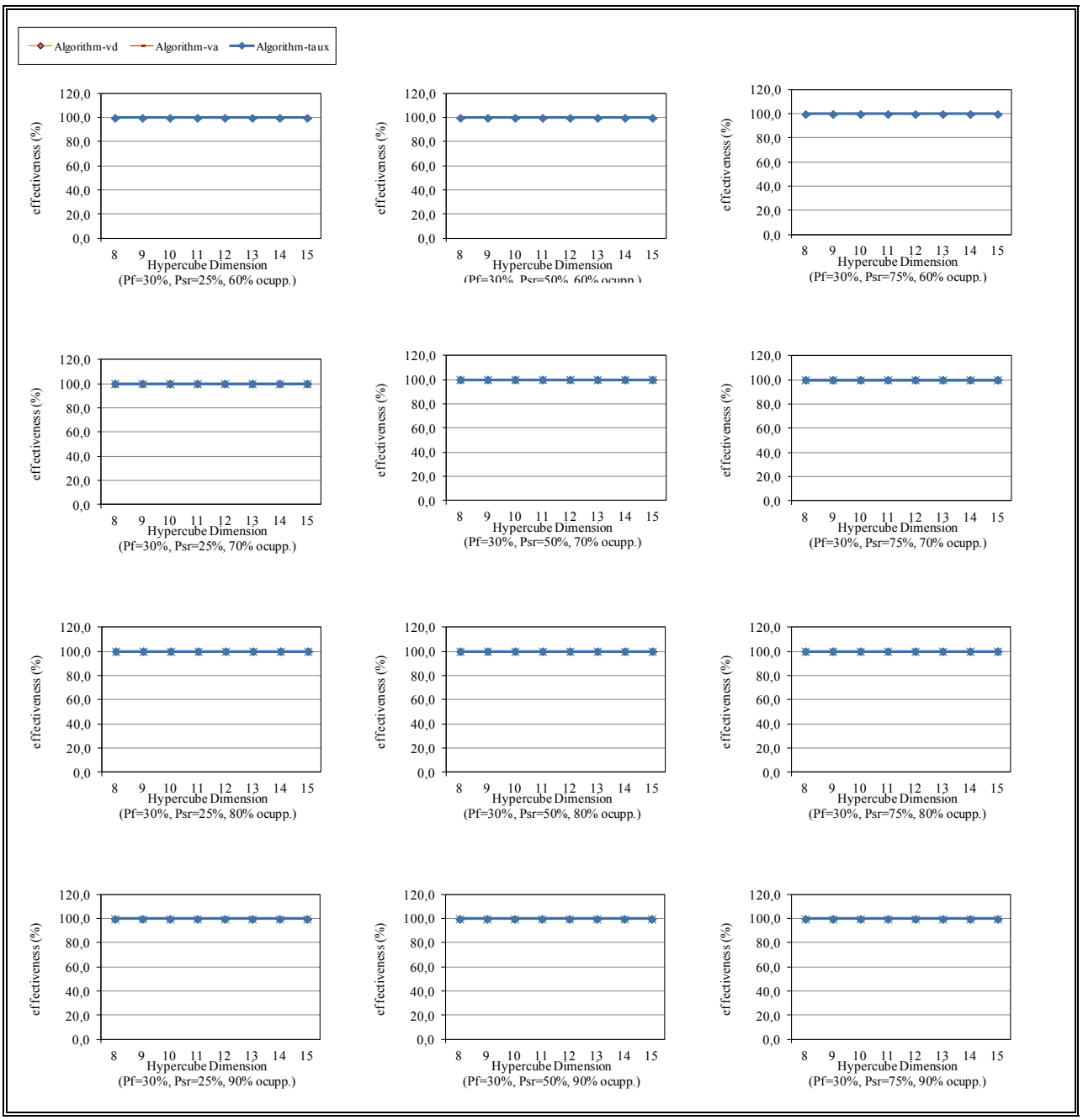

Figura 31: Escalabilidad-Porcentaje de veces que se encuentra un recurso (P_f=30%).

5.3 Flexibilidad estática en hipercubos completos

En este apartado se explica el método utilizado para evaluar cómo se comportan el algoritmo utilizado en HyperD, el utilizado por HaoRen et al. y Algoritmo-t_{aux} (propuesto en esta documento) bajo condiciones de fallo y, si son capaces o no, de localizar servicios eficientemente antes de que los algoritmos de recuperación ante fallos se ejecuten. Recordemos que se utiliza el término flexibilidad estática (*static resilience*) para referirnos a esta localización. En esta evaluación los hipercubos son hipercubos completos, es decir, hipercubos formados por $N = 2^n$

nodos (donde *N* es el número de nodos del hipercubo y *n* es la dimensión del hipercubo). También en este apartado se muestran los resultados obtenidos y se discuten éstos.

5.3.1 Comparativa con búsqueda en HyperD

En HyperD los hipercubos siempre son *n*-dimensionales completos. Cuando algún nodo tiene más de un único identificador lógico el hipercubo es parcial mientras que cuando todos los nodos tienen un único identificador lógico es un hipercubo *n*-dimensional completo lleno. En la comparativa de este apartado se utiliza un hipercubo parcial, que es el escenario más habitual. La comparativa cuando HyperD es un hipercubo *n*-dimensional completo lleno puede verse en el siguiente apartado a este ya que la búsqueda en HyperD es equivalente a la búsqueda con el algoritmo utilizado por HaoRen et al.

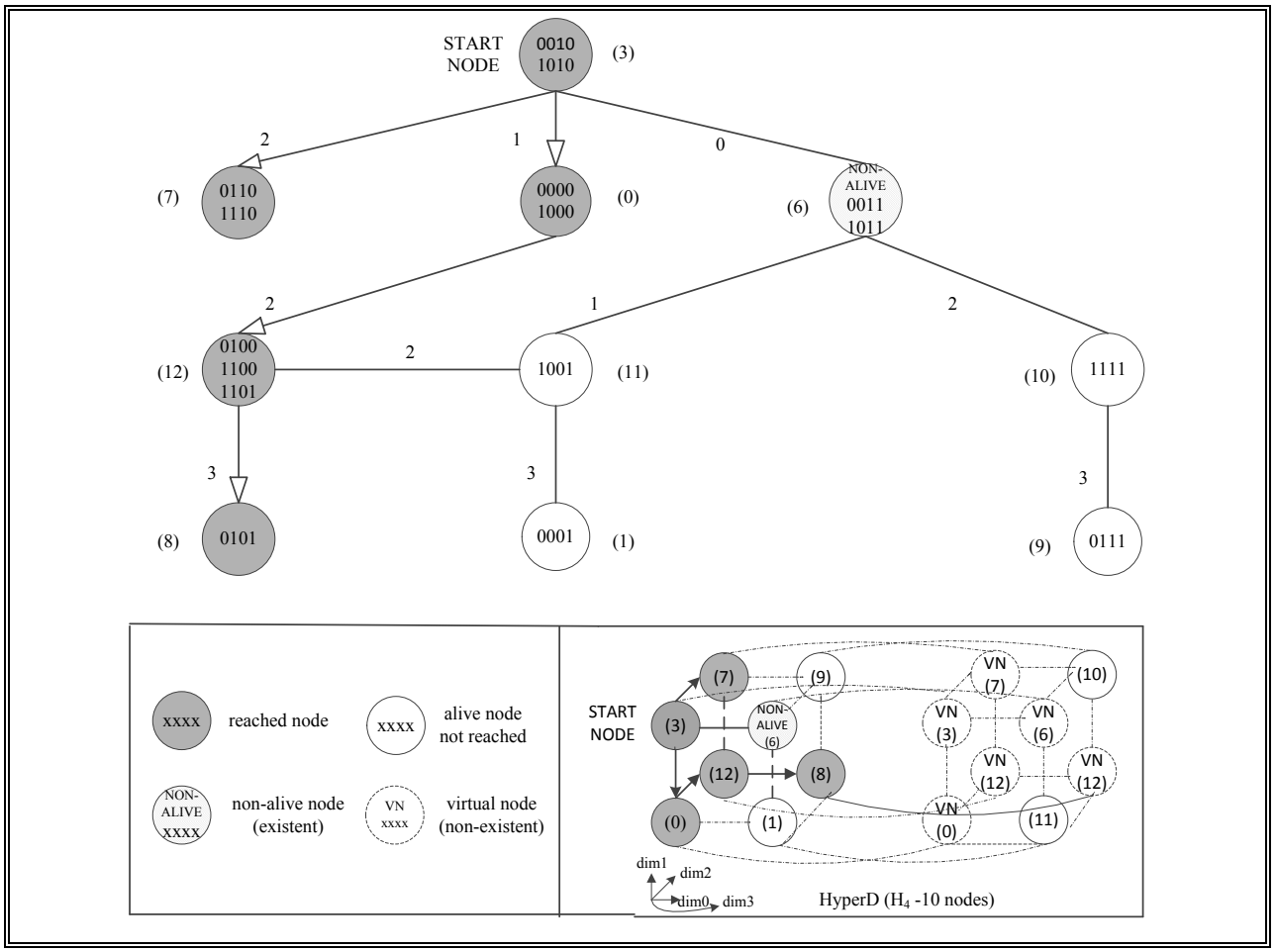

Figura 32: Ejemplo de búsqueda en HyperD con un nodo no-vivo.

Para realizar la comparativa se ha asumido que el sistema no dispone del servicio solicitado por el cliente, se ha iniciado una búsqueda desde un nodo vivo y se ha contabilizado cuantos nodos vivos no se han consultado. En la comparativa de este apartado se ha utilizado como hipercubo el ejemplo extraído de la figura 3 del artículo de los autores de HyperD [8]. Dicho ejemplo es un hipercubo 4-dimensional completo parcial formado por 10 nodos. Para representar

la evolución de la búsqueda que realiza el algoritmo se utiliza una representación en árbol. Los nodos del árbol representan nodos del hipercubo. Cuando un nodo envía la búsqueda a uno de sus nodos vecinos lo indicamos con una flecha que va del nodo emisor al nodo receptor. El algoritmo se ejecuta en varios pasos siendo el número de flechas que hay hasta el nodo inicial el número de pasos en el que el nodo es accedido. La figura 32 muestra la búsqueda iniciada desde el nodo vivo marcado como (3) con identificadores lógicos 0010/1010 donde el nodo marcado como (6) con identificadores lógicos 0011/1011 es no-vivo. En esas condiciones se consultan con el algoritmo de búsqueda en HyperD 5 de 9 nodos vivos. Los nodos no consultados son los nodos marcados como (11), (10), (1) y (9).

Para la búsqueda utilizando el algoritmo propuesto en este documento se ha considerado el caso peor, es decir, que la tabla auxiliar t_{aux} está vacía para todos los nodos. La figura 33 muestra la misma búsqueda que la figura 32 utilizando el Algoritmo-t_{aux}. Cabe notar que en este caso particular, Algoritmo-t_{aux} sólo reordena el vector v_d ya que no necesita usar el vector v_a ni las tablas t_{aux}. En esas condiciones con Algoritmo-t_{aux} se consultan 9 de 9 nodos vivos.

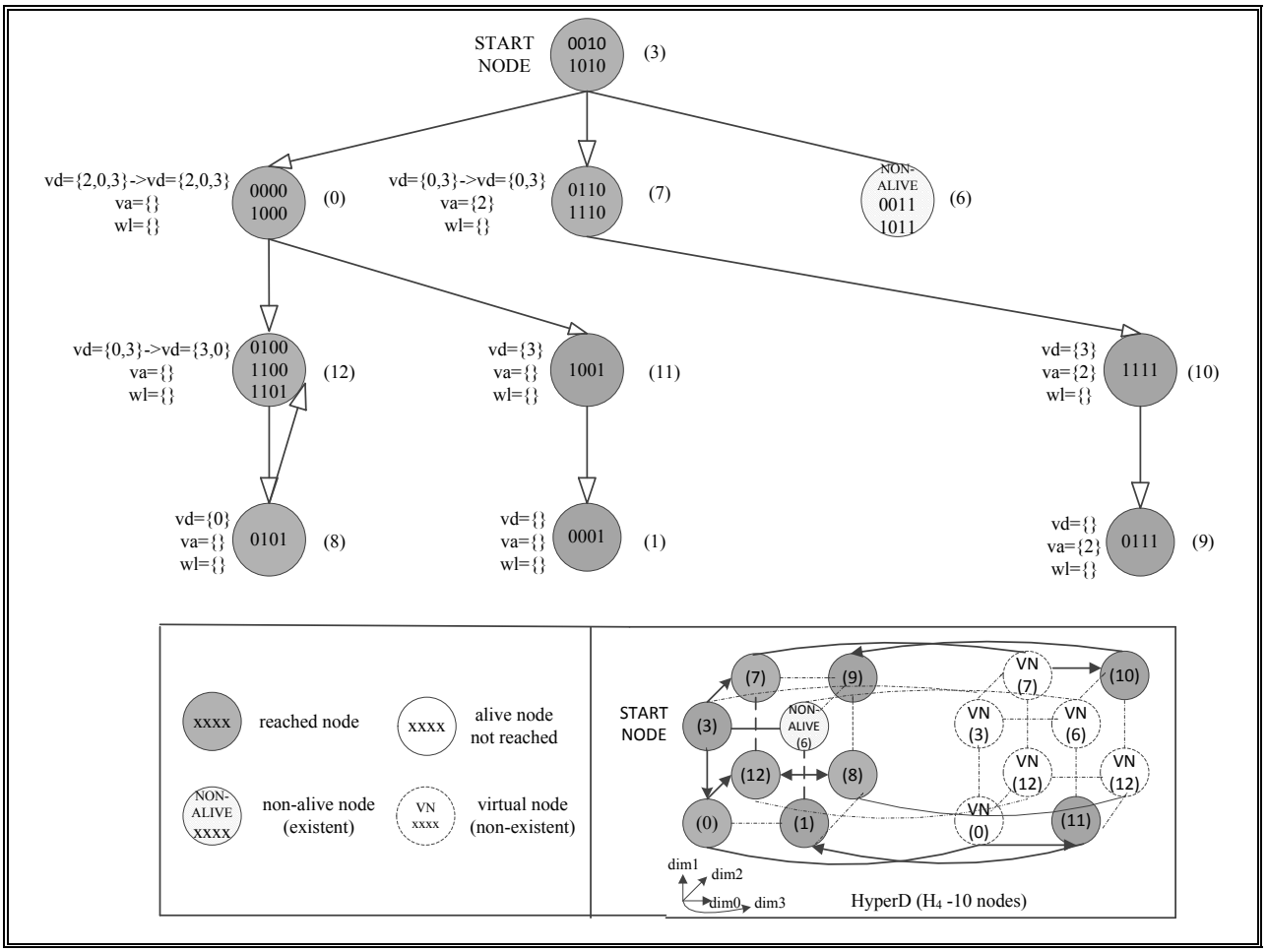

Figura 33: Ejemplo de búsqueda de Algoritmo-t_{aux} con un nodo no-vivo.

En las condiciones de la comparativa realizada, la flexibilidad estática de Algoritmo-t_{aux} es mucho mayor ya que ofrece una garantía total de localización, consultando si es necesario a todos los nodos vivos (el servicio solicitado por el cliente no se encuentra). La mejora es de 44,44% con respecto al algoritmo de búsqueda de HyperD en ese hipercubo 4-dimensional parcial.

5.3.2 Comparativa con búsqueda propuesta por HaoRen et al.

Para realizar la comparativa se han evaluado los algoritmos v_d, v_a, t_{aux} y el algoritmo utilizado por HaoRen et al. en hipercubos completos de dimensiones 14, 17 y 20, lo que nos permite compararlos en sistemas distribuidos formados por 16.384, 131.072 y 1.048.576 nodos. Cabe decir que para los casos en que los nodos de HyperD tienen uno y sólo un identificador lógico el hipercubo *n*-dimensional utilizado por HaoRen et al. es igual al hipercubo *n*-dimensional completo y que además en esos casos ambos algoritmos de búsqueda son iguales. Así, sólo para esos casos, esta comparativa también aplica a la búsqueda en HyperD. La probabilidad de este escenario en HyperD es baja (aproximadamente $1/2^n$ donde *n* es la dimensión del hipercubo) y decrece a medida que aumenta la dimensión (redes de gran escala).

5.3.2.1 Descripción del experimento

Para que la localización del servicio no influya en los resultados del experimento, se ha asumido que el sistema no dispone del servicio solicitado por el cliente y a todos los nodos del hipercubo se les ha asignado una probabilidad de fallo P_f. P_f puede interpretarse como el porcentaje de nodos no-vivos presentes en la HOverlay. Se ha ejecutado la simulación para valores de P_f entre 0 y 50%. Dada una P_f, se ha ejecutado una búsqueda de servicio *P* desde un nodo vivo y se ha contabilizado cuantos nodos vivos no se han consultado (*failed paths*) para cada uno de los algoritmos comparados. Posteriormente se ha escogido otro nodo vivo, lo que es equivalente a variar la distribución de nodos no-vivos utilizando una distribución random uniforme, y se ha vuelto a ejecutar la búsqueda de servicio *P*. Se han ejecutado un total de 20 búsquedas. Después de esta primera iteración de 20 búsquedas, se ha ejecutado una segunda iteración de las mismas 20 búsquedas de servicio desde cada uno de los nodos vivos de la primera iteración para cada uno de los algoritmos evaluados. Finalmente, para cada algoritmo evaluado, se ha calculado el porcentaje de la media de *failed paths* respecto al total de nodos vivos.

Cabe notar que calcular los porcentajes anteriores en la primera iteración o en la segunda iteración para los algoritmos de HaoRen, v_d y v_a es indistinto, se obtienen los mismos resultados y que en el objetivo de la primera iteración de búsquedas en el experimento sólo ha sido asegurar que las tablas t_{aux} de Algoritmo-t_{aux} no se encuentren vacías o prácticamente vacías, ya que en ese caso Algoritmo-t_{aux} es equivalente a Algoritmo-v_a.

Cabe notar también que en Algoritmo-t_{aux} la mejora en el porcentaje calculado con una única iteración o con dos es muy pequeña. Por ejemplo, en el experimento realizado se ha obtenido para el caso concreto del hipercubo *n*-dimensional completo de dimensión 20 con probabilidad de fallo $P_f = 10\%$, un porcentaje del 0,23 % con una iteración y un porcentaje del 0,21 % con dos iteraciones.

A diferencia del Algoritmo-v_a, el Algoritmo de HaoRen et al. no reordena el vector v_d y no utiliza el vector v_a. El Algoritmo-v_d reordena el vector v_d pero no usa el vector v_a. Finalmente el

Algoritmo-t_{aux} continúa en algunos casos la búsqueda cuando los demás algoritmos no pueden (utilizando una tabla auxiliar). Cómo se generan y utilizan los vectores y la tabla se ha explicado anteriormente.

5.3.2.2 Datos e interpretación del experimento

En la figura 34 se muestra gráficamente al variar P_f el porcentaje de la media de nodos vivos no consultados (*failed paths*). La unidad del eje de ordenadas es % y la del eje de abscisas indica la probabilidad de fallo. Dado que la gráfica muestra el porcentaje de nodos no consultados, el algoritmo con porcentaje más alto será el peor y el algoritmo con porcentaje más bajo el mejor de los evaluados. Visualmente las gráficas de la figura 34 se agrupan en 3 grupos. De arriba a abajo, el primer grupo, formado por las 3 gráficas superiores corresponden al algoritmo de búsqueda utilizado por HaoRen et al., el segundo grupo, formado por 3 gráficas que se solapan visualmente en una sola línea corresponden al Algoritmo-v_d y el tercer grupo, formado por 6 gráficas, que se solapan visualmente en una sola línea, corresponden a los algoritmos v_a y t_{aux}. Los valores exactos obtenidos en el experimento se detallan en el anexo B.

Así, la gráfica de la figura 34 muestra que los algoritmos v_d, v_a y t_{aux} ofrecen sustancialmente mejor flexibilidad estática que el Algoritmo utilizado por HaoRen et al. Los algoritmos t_{aux} y v_a son además mejores que v_d. Gráficamente no se aprecia que t_{aux} sea mejor que v_a, pero en los valores exactos mostrados en el anexo B puede verse una mejora de t_{aux} con respecto a v_a. Cabe esperar también, que esa mejora aumente a medida que se realicen más búsquedas en HOverlay (en esta simulación se han realizado sólo 20), ya que la tabla auxiliar que utiliza t_{aux} para continuar las búsquedas cuando v_a no puede, contendrá más nodos y será más probable continuar la propagación si es necesario.

De los resultados se observa que al algoritmo utilizado por HaoRen et al. es el único que le afecta la dimensión del hipercubo. Como en esta simulación sólo se realizan 20 búsquedas desde nodos vivos escogidos aleatoriamente y el estado de todos los nodos es aleatorio, no podemos determinar con exactitud cuál es el comportamiento al aumentar la dimensión del hipercubo (más adelante confirmaremos que a este algoritmo le afecta la dimensión del hipercubo).

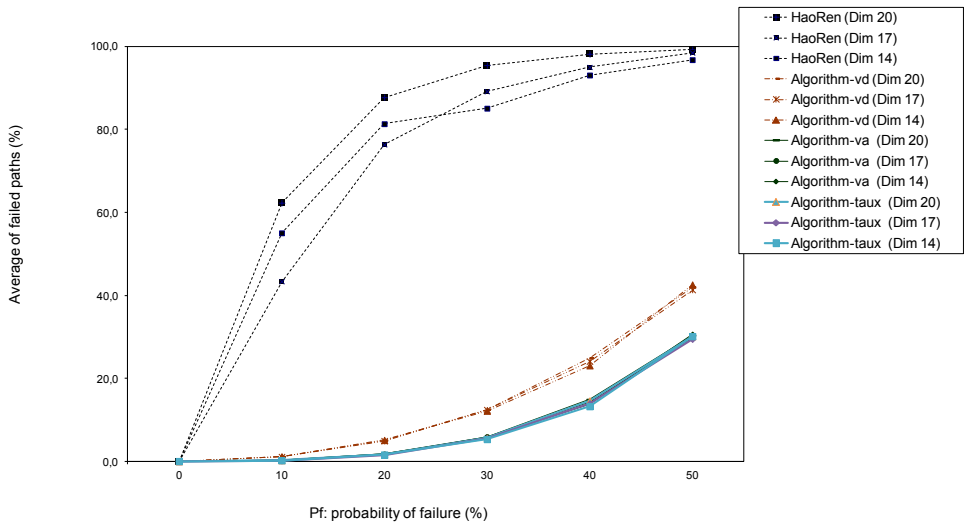

Figura 34: Porcentaje de la media de *failed paths* en hipercubos completos.

A todos los algoritmos les afecta P_f, pero al algoritmo utilizado por HaoRen et al. le afecta mucho más. Por ejemplo, al incrementar el valor de P_f de un 10% a un 20% en el hipercubo 20-dimensional completo de la simulación, se produce un incremento en el porcentaje de la media de *failed paths* del 25,44% para el Algoritmo propuesto por HaoRen et al., del 3,88% para v_d, del 1,48% para v_a y del 1,37% para t_{aux}. Este hecho también se observa en la tabla 2 que muestra la media y la varianza de los datos de la gráfica de la figura 34.

Algorithm (hypercube's dimension)	*Average*	*Variance*
HaoRen (Dim. 14)	68,56	1343,77
HaoRen (Dim. 17)	67,08	1481,23
HaoRen (Dim. 20)	73,83	1497,35
Algorithm-vd (Dim. 14)	14,01	268,64
Algorithm-vd (Dim. 17)	13,98	259,87
Algorithm-vd (Dim. 20)	14,26	269,48
Algorithm-va (Dim. 14)	8,69	142,10
Algorithm-va (Dim. 17)	8,67	136,13
Algorithm-va (Dim. 20)	8,86	142,06
Algorithm-taux (Dim. 14)	8,41	137,86
Algorithm-taux (Dim. 17)	8,42	133,75
Algorithm-taux (Dim. 20)	8,69	140,05

Tabla 2: Media y varianza del porcentaje de la media de *failed paths* de la figura 34.

Del experimento podemos extraer también la conclusión de que los algoritmos v_d, v_a y t_{aux} son capaces de localizar eficientemente servicios buscados en condiciones de fallo ya que dejan de consultar si es necesario (no se encuentra) porcentajes muy bajos de nodos vivos (que pueden tener o no el servicio buscado).

Los resultados confirman que la flexibilidad estática de los algoritmos v_d, v_a y t_{aux} en hipercubos completos es eficiente para los entornos distribuidos de gran escala donde la P_f no sea extremadamente alta, ya que ofrecen garantías de localización elevadas consultando, si es necesario (no se encuentra el servicio solicitado por el cliente), a casi todos los nodos. Además, el número de nodos que tenga el sistema no parece afectar a las garantías de la búsqueda ya que se obtiene prácticamente la misma gráfica para las 3 dimensiones evaluadas. De todos los algoritmos evaluados, t_{aux} es el que presenta mejores prestaciones al aumentar la P_f.

5.4 Flexibilidad estática en hipercubos incompletos

En este apartado se explica el método utilizado para evaluar cómo se comportan el algoritmo propuesto por HaoRen et al. y los presentados en esta tesis (v_d, v_a, t_{aux}) bajo condiciones de fallo y, si son capaces o no, de localizar servicios eficientemente antes de que los algoritmos de recuperación ante fallos se ejecuten. Recordemos que se utiliza el término flexibilidad estática (*static resilience*) para referirnos a esta localización. A diferencia del apartado anterior, en esta evaluación los hipercubos son hipercubos *n*-dimensionales incompletos, es decir, no están formados por $N = 2^n$ nodos (donde N el número de nodos del hipercubo y n es la dimensión del hipercubo). En este apartado no aplica el comparar con el algoritmo de búsqueda utilizado en HyperD ya que en HyperD, el hipercubo siempre es un hipercubo *n*-dimensional completo. Después de describir el experimento, se muestran los resultados obtenidos y se discuten éstos.

5.4.1 Descripción del experimento

Las simulaciones realizadas cubren hasta 65.000 nodos aproximadamente utilizando hipercubos *n*-dimensionales incompletos donde su porcentaje de ocupación y su dimensión varían. El valor de n varía entre 8 y 16. Para cada dimensión, se considera que el hipercubo (de HOverlay) está ocupado en un 60%, 70%, 80% o 90%. Por ejemplo, dado un hipercubo de *n*-dimensional con un 60% de ocupación, los nodos cuyos identificadores van de 0 a $((2^n-1)*60/100)-1$ formarán HOverlay. Los nodos cuyos identificadores van de $((2^n-1)*60/100)$ a (2^n-1) serán considerados como nodos no-vivos por los algoritmos evaluados. No se consideran ocupaciones iguales o menores al 50% porque un hipercubo *n*-dimensional ocupado en un 50% o menos es un hipercubo (n-1)-dimensional.

Para que la localización del servicio no influya en los resultados del experimento, se ha asumido que el sistema no dispone del servicio solicitado por el cliente y una probabilidad de fallo, llamada P_f se asigna a cada nodo que forma parte de HOverlay. P_f puede verse como el porcentaje de nodos no-vivos que se encuentran en la overlay. Todos los nodos tienen la misma P_f.

La simulación se ejecuta para los cuatro algoritmos evaluados para valores de P_f iguales a 10%, 20% y 30%. Dada un P_f, un nodo vivo es escogido de HOverlay y una petición de búsqueda

de servicio *P* se inicia en este nodo. La simulación de los 4 algoritmos se repite para cada uno de los nodos vivos presentes en HOverlay y en cada búsqueda de servicio se elige un nuevo nodo vivo inicial. Después de esta primera iteración por cada uno de los nodos vivos, se ha ejecutado una segunda iteración de una búsqueda de servicio desde cada uno de los nodos vivos para cada uno de los algoritmos evaluados. Finalmente, para cada algoritmo evaluado, se calcula el porcentaje de la media de nodos vivos que no se han consultados (*failed paths*) sobre el total de nodos vivos.

Cabe notar que calcular el porcentaje anterior en la primera iteración o en la segunda iteración para los algoritmos de HaoRen, v_d y v_a es indistinto, se obtienen los mismos resultados y que en el objetivo de la primera iteración de búsquedas en el experimento sólo ha sido asegurar que las tablas t_{aux} de Algoritmo-t_{aux} no se encuentren vacías o prácticamente vacías en las búsquedas desde los primeros nodos vivos de la iteración, ya que en ese caso Algoritmo-t_{aux} es equivalente a Algoritmo-v_a.

Cabe notar también que en Algoritmo-t_{aux} la mejora en el porcentaje de la media de nodos vivos consultados respecto el total de nodos vivos con una única iteración o con dos es muy pequeña. Por ejemplo, en el experimento realizado se ha obtenido para el caso concreto del hipercubo *n*-dimensional incompleto de dimensión 16 ocupado en un 60% y con probabilidad de fallo P_f = 10%, se obtiene un porcentaje del 0,81% con una iteración y un porcentaje del 0,46% con dos iteraciones.

Dada una dimensión, un porcentaje de ocupación y una P_f, todos los algoritmos evaluados (v_d, v_a, t_{aux} y el algoritmo propuesto por HaoRen et al.) tienen el mismo escenario, con la misma localización de nodos no-vivos, para poder realizar una comparación justa.

El experimento permite comparar la flexibilidad estática de los algoritmos v_d, v_a, t_{aux} y del algoritmo propuesto por HaoRen et al. en hipercubos incompletos.

A diferencia del Algoritmo-v_a, el Algoritmo de HaoRen et al. no reordena el vector v_d y no utiliza el vector v_a. El Algoritmo-v_d reordena el vector v_d pero no usa el vector v_a. Finalmente el Algoritmo-t_{aux} continúa en algunos casos la búsqueda cuando los demás algoritmos no pueden (utilizando una tabla auxiliar). Cómo se generan y utilizan los vectores y la tabla se ha explicado anteriormente.

5.4.2 Datos e interpretación del experimento

La figura 35 muestra, para los algoritmos presentados en esta tesis y para el algoritmo propuesto por HaoRen et al., el porcentaje de la media de nodos vivos no consultados (*failed paths*) sobre el total de nodos vivos, considerando la probabilidad de fallo (P_f) (columnas) y el porcentaje de ocupación (filas). Para un mayor detalle de los algoritmos presentados en esta tesis la figura 36 muestra los mismos datos que la figura 35 sin mostrar los datos obtenidos para el algoritmo propuesto por HaoRen et al. Para cada una de las gráficas, la unidad en el eje de ordenadas es % y el eje de abscisas indica la dimensión del hipercubo. Dado que las gráficas muestran el porcentaje de la media de nodos vivos no consultados, el algoritmo con porcentaje más alto será el peor y el algoritmo con porcentaje más bajo será el mejor de los evaluados. Los valores exactos obtenidos se encuentran en el anexo C.

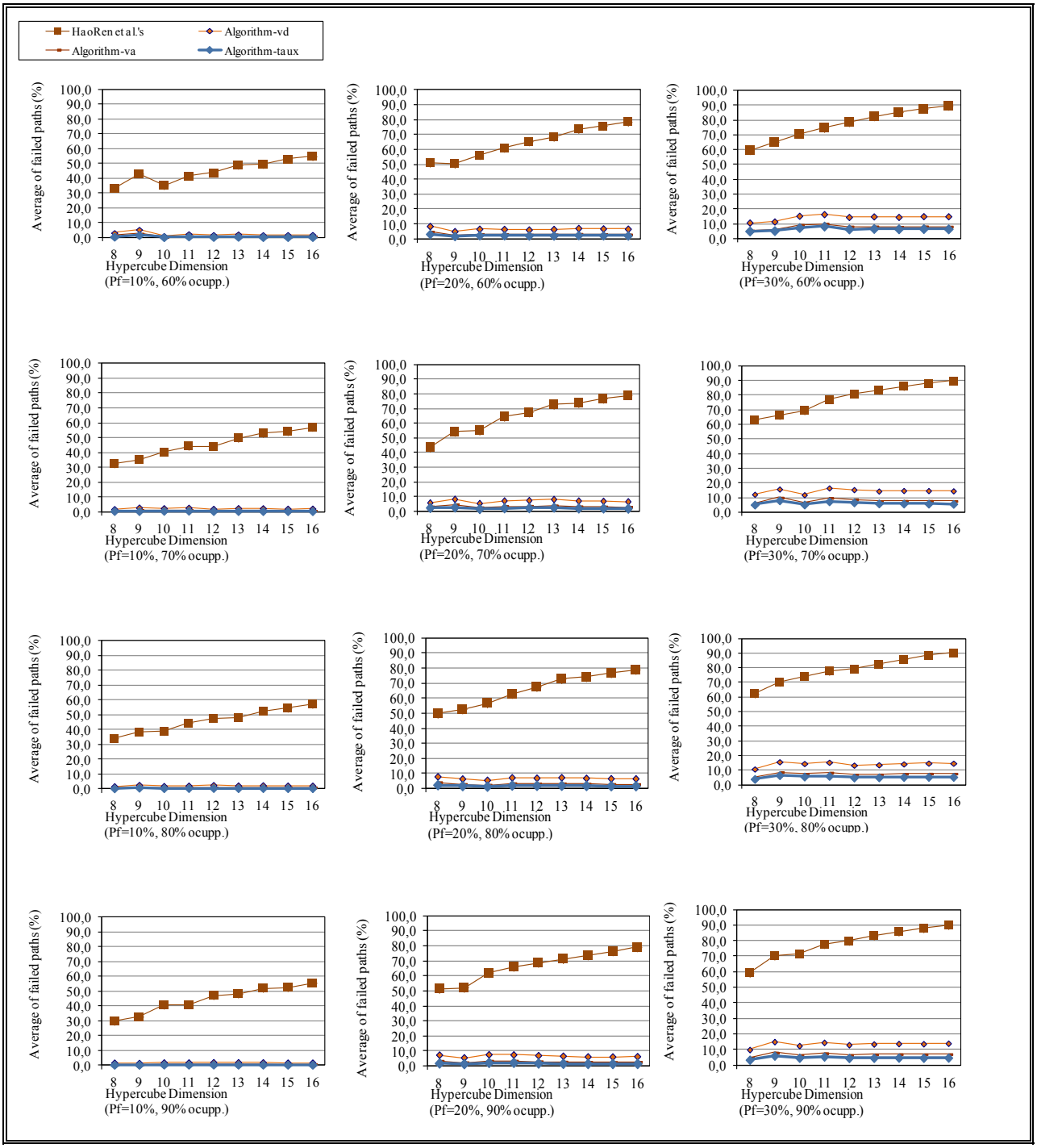

Figura 35: Porcentaje de la media de *failed paths* en hipercubos incompletos (1).

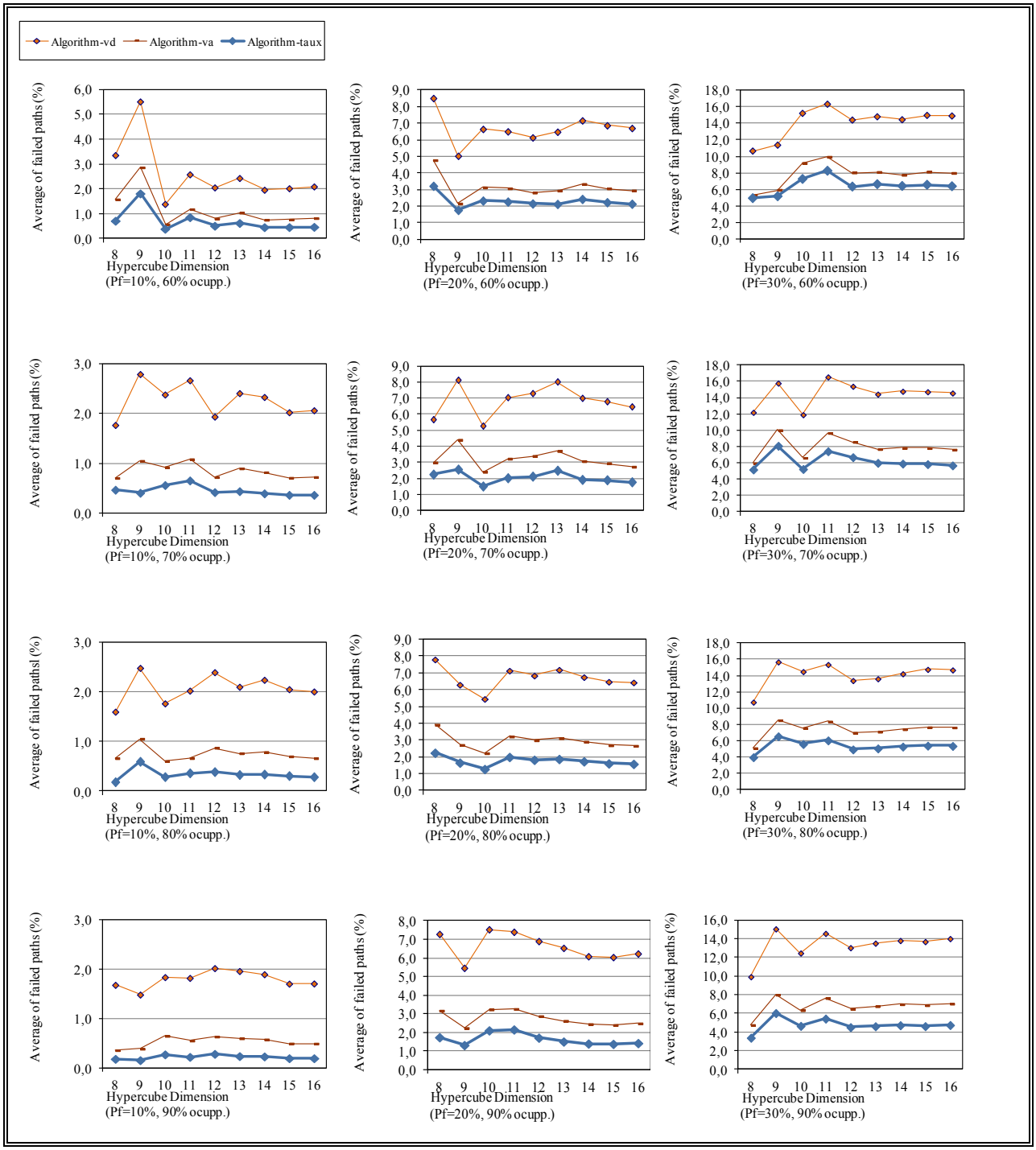

Figura 36: Porcentaje de la media de *failed paths* en hipercubos incompletos (2).

En las tablas 3 y 4 se muestra la media y la varianza respectivamente de los datos de la gráfica de la figura 35.

Algorithm(% occup.)	*Average (Pf = 10%)*	*Average (Pf = 20%)*	*Average (Pf = 30%)*
HaoRen (60% occup.)	44,89	64,50	77,02
HaoRen (70% occup.)	45,51	65,11	78,23
HaoRen (80% occup.)	46,07	65,77	79,01
HaoRen (90% occup.)	44,26	66,77	78,52
Algorithm-vd (60% occup.)	2,60	6,67	14,11
Algorithm-vd (70% occup.)	2,27	6,84	14,51
Algorithm-vd (80% occup.)	2,07	6,72	14,14
Algorithm-vd (90% occup.)	1,79	6,61	13,34
Algorithm-va (60% occup.)	1,14	3,13	7,78
Algorithm-va (70% occup.)	0,85	3,19	8,00
Algorithm-va (80% occup.)	0,75	2,95	7,40
Algorithm-va (90% occup.)	0,53	2,74	6,77
Algorithm-taux (60% occup.)	0,69	2,29	6,46
Algorithm-taux (70% occup.)	0,45	2,05	6,25
Algorithm-taux (80% occup.)	0,34	1,76	5,38
Algorithm-taux (90% occup.)	0,23	1,63	4,76

Tabla 3: Media del porcentaje de la media de *failed paths* de la figura 35.

Algorithm(% occup.)	*Variance (Pf = 10%)*	*Variance (Pf = 20%)*	*Variance (Pf = 30%)*
HaoRen (60% occup.)	55,21	110,79	106,47
HaoRen (70% occup.)	72,90	144,80	96,50
HaoRen (80% occup.)	62,83	116,46	82,45
HaoRen (90% occup.)	80,67	98,97	97,96
Algorithm-vd (60% occup.)	1,49	0,84	3,44
Algorithm-vd (70% occup.)	0,12	0,91	2,35
Algorithm-vd (80% occup.)	0,08	0,44	2,13
Algorithm-vd (90% occup.)	0,03	0,51	2,22
Algorithm-va (60% occup.)	0,50	0,48	2,05
Algorithm-va (70% occup.)	0,02	0,34	1,70
Algorithm-va (80% occup.)	0,02	0,22	0,98
Algorithm-va (90% occup.)	0,01	0,16	0,83
Algorithm-taux (60% occup.)	0,20	0,15	0,98
Algorithm-taux (70% occup.)	0,01	0,11	0,98
Algorithm-taux (80% occup.)	0,01	0,08	0,52
Algorithm-taux (90% occup.)	0,00	0,10	0,51

Tabla 4: Varianza del porcentaje de la media de *failed paths* de la figura 35.

Cabe notar que aunque en HOverlay la secuencia de nodos (por orden) que forman la overlay no sigue un código de Gray sino uno decimal, esto no afecta a los resultados obtenidos en este experimento para el algoritmo propuesto por HaoRen et al. ya que vemos que para todas las ocupaciones evaluadas (60%, 70%, 80% y 90%) los valores de la figura 35 para las dimensiones 13, 14, 15 y 16 están en torno a los valores obtenidos para hipercubos 14 y 17-dimensionales completos de la figura 34 (comparativa para hipercubos completos).

Todos los algoritmos exhiben un resultado en común que puede observarse en la figuras 35 y 36. El porcentaje de la media de nodos vivos no consultados no parece depender de lo incompleto que esté el hipercubo (% ocupp.), pero sí depende de la probabilidad de fallo de los nodos (P_f). Al aumentar P_f aumenta el número de veces que un nodo no puede propagar la búsqueda de servicios porque los vecinos a los que debe propagar están en estado no-vivo.

Si observamos el resultado de la simulación, vemos que los algoritmos v_d, v_a y t_{aux} ofrecen sustancialmente mejor flexibilidad estática que el algoritmo propuesto por HaoRen et al. De la misma forma, t_{aux} ofrece mejor flexibilidad estática que v_a y v_a mejor flexibilidad estática que v_d. Reordenando el vector v_d la flexibilidad estática de t_{aux}, v_a y v_d mejora sustancialmente, ya que se consultan más nodos vivos. Además, utilizando el vector v_a, el Algoritmo-v_a consulta más nodos utilizando rutas alternativas. Finalmente, t_{aux} utilizando su tabla auxiliar consulta aún más nodos que el resto de algoritmos evaluados.

Al aumentar la dimensión del hipercubo el comportamiento del algoritmo utilizado por HaoRen et al. difiere del comportamiento de los algoritmos propuestos en esta tesis. El porcentaje de la media de nodos vivos no consultados se incrementa para el algoritmo utilizado por Haoren et al., mientras que se mantiene más o menos constante para los otros tres algoritmos. Además en el algoritmo utilizado por HaoRen et al. el porcentaje se mueve entre el 30% y el 90%, mientras que para el resto (v_d, v_a y t_{aux}) no supera el 16%. De nuevo, reordenar el vector v_d limita el efecto de los nodos no-vivos en la propagación de la petición de búsqueda de servicios.

El incremento de la probabilidad de fallo le afecta mucho más al algoritmo utilizado por HaoRen et al. que a los algoritmos descritos en esta tesis (v_d, v_a y t_{aux}). Por ejemplo, al incrementar el valor de P_f de un 10% a un 20% en el hipercubo 16-dimensional incompleto con un 80% de ocupación de la simulación, se produce un incremento en el porcentaje de la media de nodos vivos no consultados del 21,89% para el algoritmo utilizado por HaoRen et al., un incremento del 4,44% para v_d, un incremento del 2,01% para v_a y un incremento del 1,31% para t_{aux}. Cabe destacar que este resultado se obtiene en hipercubos *n*-dimensionales incompletos (presente experimento) y ya se había obtenido en hipercubos *n*-dimensionales completos (experimento detallado con anterioridad).

Como en el experimento para hipercubos *n*-dimensionales completos, en el presente experimento (hipercubos *n*-dimensionales incompletos), los resultados confirman que la flexibilidad estática de los algoritmos v_d, v_a y t_{aux} es eficiente para los entornos distribuidos donde

la P_f no sea extremadamente alta, ya que ofrecen garantías de localización elevadas consultando si es necesario (el servicio solicitado por el cliente no se encuentra) a casi todos los nodos del sistema distribuido independientemente del número de nodos que tenga el sistema. De todos los algoritmos evaluados, t_{aux} es el que presenta mejores prestaciones al aumentar la P_f.

5.5 Tiempos de búsqueda por simulación

En este apartado se describe la simulación realizada para medir el tiempo de búsqueda de servicios en HOverlay. La simulación se ha realizando utilizando tiempos de transferencia entre máquinas obtenidos del proyecto Reliable Service Infrastructure in Donation-based Grid Environments (RIDGE) [48]. En este proyecto se obtuvieron tiempos de transferencia de datos durante un periodo de tiempo de 10 meses, de julio de 2007 a abril de 2008, en PlanetLab. Como ya se ha dicho, PlanetLab es una red de investigación global. Puede verse como un conjunto de máquinas distribuidas a lo largo del planeta, sobre las que se pueden instalar y ejecutar aplicaciones distribuidas en condiciones reales. Actualmente, PlanetLab dispone de 1078 máquinas que pertenecen a 530 instituciones diferentes. Los resultados del proyecto RIDGE se encuentran disponibles a través de un fichero de texto en su página web.

5.5.1 Descripción del experimento

En esta simulación se ha evaluado el tiempo de búsqueda de servicios en HOverlay para los algoritmos v_d, v_a, t_{aux} y el algoritmo utilizado por HaoRen et al. Las HOverlays consideradas han estado formadas por 614, 768 y 921 nodos. Para ello, se han generado hipercubos 10-dimensionales incompletos ocupados en un 60%, 75% y 90 %. No se han considerado ocupaciones iguales o menores al 50% porque un hipercubo *n*-dimensional ocupado en un 50% o menos es un hipercubo (*n*-1)-dimensional.

En la simulación se ha evaluado cada HOverlay con una probabilidad de fallo (P_f) en los nodos del 30% y con una probabilidad de satisfacer la petición (P_{sr}) en los nodos del 1%. P_f puede interpretarse como el porcentaje de nodos no-vivos que pueden encontrarse en la overlay. P_{sr} puede interpretarse como el porcentaje de nodos vivos que dispone del servicio que el cliente busca. Se han establecido estos valores considerando condiciones extremas para las búsquedas de servicios, alta probabilidad de fallo de los nodos y muy pocos nodos que dispongan del servicio buscado.

A cada nodo presente en la overlay se le ha asignado un tiempo del fichero disponible en el proyecto RIDGE. La asignación de tiempos ha sido aleatoria, de forma que, para al nodo con identificador 0 de la overlay se le ha asignado el tiempo del fichero almacenado en una posición aleatoria, al nodo con identificador 1 se le ha asignado el siguiente tiempo después de esta posición aleatoria y así sucesivamente. Como los datos almacenados en el fichero no están ordenados, la asignación ha sido aleatoria. Este tiempo se ha interpretado como el tiempo que tarda el mensaje de búsqueda en llegar a este nodo desde otro nodo más el tiempo que necesita este nodo para ejecutar el algoritmo evaluado.

Una vez generado el hipercubo 10-dimensional incompleto (con una P_f de 30% y una P_{sr} de 1%, en cada nodo), para cada % de ocupación, se han ordenado aleatoriamente los nodos vivos del hipercubo incompleto, se ha ejecutado una búsqueda desde cada uno de los nodos vivos

siguiendo ese orden para cada uno de los algoritmos evaluados. Para cada búsqueda se ha almacenado el tiempo menor que ha tardado la búsqueda en encontrar un nodo que satisface el servicio que solicita el cliente. Este tiempo puede verse como el tiempo menor que transcurre desde que el cliente envía el mensaje de búsqueda hasta que se encuentra un nodo en HOverlay que satisface la búsqueda.

También se ha calculado el porcentaje de éxito para cada uno de los algoritmos, es decir, en cuantas búsquedas desde el cliente se ha encontrado el servicio.

5.5.2 Datos e interpretación del experimento

Para representar los datos se utiliza el percentil. Un percentil, $y\% = v$ significa que en $y\%$ de las búsquedas de servicio, el tiempo de búsqueda obtenido está por debajo del tiempo v. Así, un percentil del 50% muestra el tiempo de búsqueda suficiente para encontrar el servicio en el 50% de las búsquedas realizadas y un percentil del 100% muestra el tiempo de búsqueda máximo obtenido.

La figura 37 muestra el tiempo de las búsquedas de las que se ha obtenido respuesta en función del % de ocupación y del percentil. Los percentiles mostrados son 25%, 50%, 75% y 100% (filas) y el % de ocupación del hipercubo 10-dimensional (% occupp.) es 60%, 75% y 90% (columnas). Para cada una de las gráficas mostradas, la unidad de tiempo en el eje de ordenadas es el milisegundo. El eje de abscisas indica cada uno de los 4 algoritmos evaluados: algoritmo utilizado por HaoRen et al., v_d, v_a y t_{aux}. Dado que las gráficas muestran el tiempo de búsqueda, el o los algoritmos con tiempos menores son los mejores de los que se evalúan. Los valores exactos obtenidos se encuentran en el anexo D.

Como ejemplo, el primer valor de la primera gráfica (337,5 msecs.) significa que del total de respuestas recibidas (servicios encontrados) para el algoritmo utilizado por HaoRen et al., el 25% de las respuestas obtenidas no tardaron más de 337,5 msecs.

Estableciendo un % de ocupación (columna) vemos que al incrementar el percentil, incrementa el tiempo obtenido. La explicación de ello es que aumentando el percentil estamos añadiendo casos en los que el tiempo de búsqueda es cada vez mayor.

De las gráficas obtenidas se observa que el algoritmo utilizado por HaoRen et al. no siempre es peor que los algoritmos propuestos v_d, v_a y t_{aux}. Debe tenerse en cuenta que la comparación del algoritmo de HaoRen et al. con el resto de algoritmos evaluados no puede realizarse utilizando las gráficas mostradas en la figura 37. La razón se encuentra en que estas gráficas muestran el tiempo de las búsquedas de las que se ha obtenido respuesta y, en el caso del algoritmo de HaoRen et al., no siempre se encuentra el servicio buscado. La figura 38 muestra el porcentaje de veces que se encuentra un servicio para las búsquedas realizadas desde nodos vivos (efectividad). Los valores exactos obtenidos se encuentran en el anexo D.

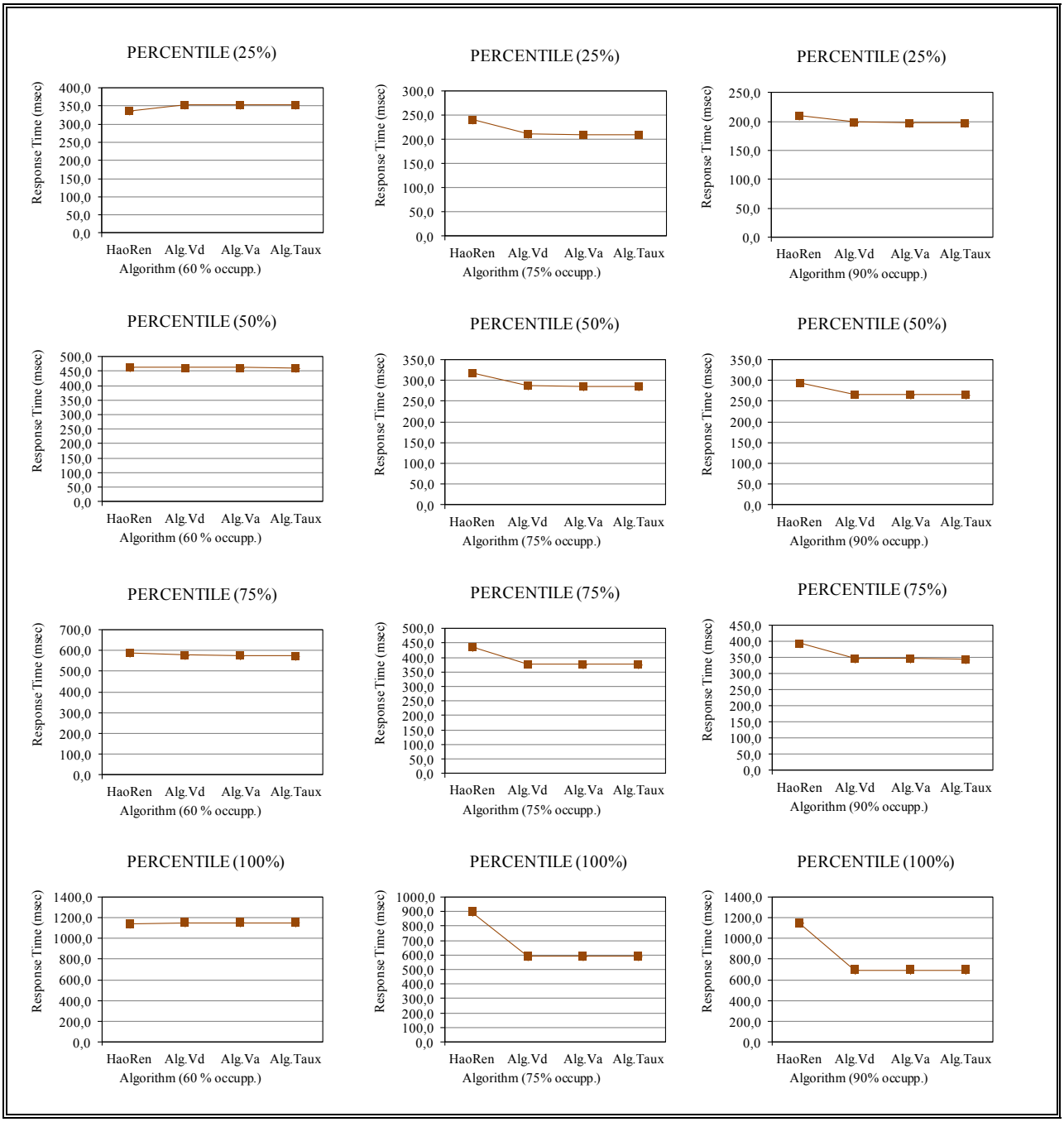

Figura 37: Tiempo de búsqueda (P_f=30%)(P_{sr}=1%).

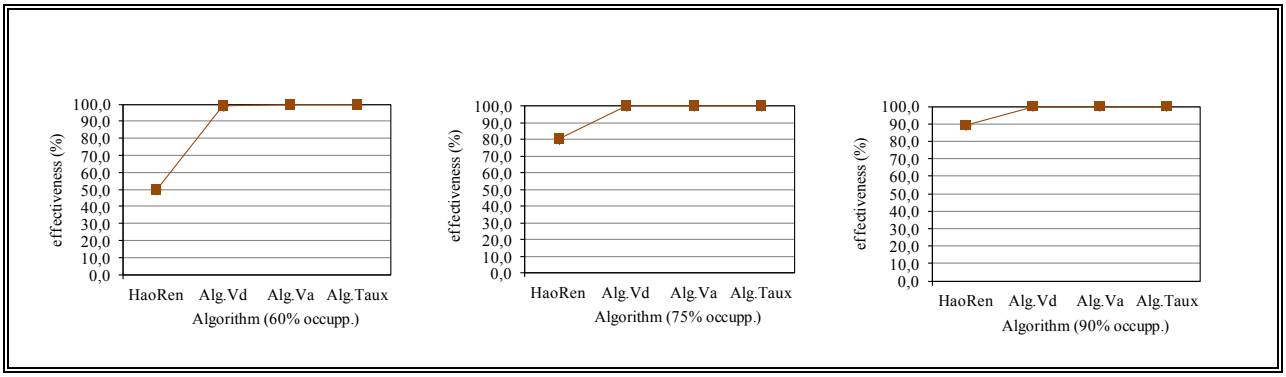

Figura 38: Porcentaje de veces que se encuentra un servicio buscado (P_f=30%)(P_{sr}=1%).

Si observamos el porcentaje de veces que se encuentra un servicio buscado en la figura 38, vemos que la efectividad del algoritmo utilizado por HaoRen et al. es claramente inferior al resto de los algoritmos. Para los algoritmos v_d, v_a y t_{aux} se encuentra en un 100% de los casos el servicio buscado mientras que para el Algoritmo utilizado por HaoRen et al. esto no sucede.

Concluimos de esta simulación que el algoritmo utilizado por HaoRen et al. no asegura que se encuentre un servicio en HOverlay, aunque dicho servicio esté presente, al menos para las condiciones de la simulación realizada: una P_f = 30% y una P_{sr} = 1% en un hipercubo 10-dimensional incompleto con % de ocupación del 60%, 75% y 90%. En cambio, los algoritmos v_d, v_a y t_{aux} encuentran el servicio en estas condiciones en un 100% de los casos.

5.6 Tiempos de búsqueda en PlanetLab

En este apartado se presenta el sistema distribuido real implementado para el servicio de descubrimiento descrito en esta tesis y el experimento realizado en él. El sistema distribuido utiliza PlanetLab como infraestructura. Como ya se ha dicho, PlanetLab es una red de investigación global que permite desarrollar nuevos servicios. Desde sus inicios en 2003, más de 1000 investigadores de las principales instituciones académicas y laboratorios industriales de investigación, han utilizado PlanetLab para desarrollar nuevas tecnologías para el almacenamiento distribuido, mapeo de red, sistemas P2P, DHTs, etc.

PlanetLab actualmente dispone de 1093 nodos en 533 lugares. De los nodos disponibles se eligieron 150 nodos distribuidos a lo largo de Europa para crear HOverlay. Así se obtuvo un hipercubo 8-dimensional incompleto con una ocupación del 59%.

Tal y como hemos comentado anteriormente, cada nodo (o BIS) en HOverlay tiene un identificador. El identificador de cada BIS fue la posición en la que la máquina aparecía en la lista de nodos de PlanetLab en el momento en que se creó la overlay (noviembre 2011). Dado que dicha lista de nodos no sigue ningún orden aparente y que 2 identificadores en decimal seguidos en HOverlay no son necesariamente vecinos (por ejemplo el nodo con identificador 7(0111) y nodo con identificador 8(1000) no son vecinos en HOverlay), la overlay creada fue aleatoria sin tener en cuenta ningún criterio geográfico que permitiera (a priori) optimizar el tiempo de búsqueda. La lista de nodos utilizados con el identificador de cada uno de ellos se encuentra en el anexo E.

En HOverlay cada nodo almacena el estado de sus vecinos. El estado de un vecino se actualizó cada 10 segundos enviando un mensaje al nodo vecino. Si el nodo vecino contestaba al mensaje, su estado se actualizaba como vivo. En otro caso, el estado del vecino se consideraba no-vivo.

Entre los nodos escogidos no fue posible desplegar el servicio de descubrimiento en 37 de ellos; obteniendo una tasa inicial de fallo del 24,67%. Durante el experimento, no se obtuvo ninguna respuesta de un total de 7 nodos; 3 de ellos interrumpieron su ejecución y el resto parece que generaban *timeouts* en las respuestas. Así, el total de nodos no-vivos fue de 44 y la tasa final de nodos en estado no-vivo en el momento del experimento del 29,33%.

Una vez puesto en marcha HOverlay, si un nodo recibía un mensaje de búsqueda, el Algoritmo-v_a se ejecutaba. HOverlay y el Algoritmo-v_a se implementaron utilizando el protocolo de transporte TCP. Pruebas utilizando UDP en vez de TCP, mostraron que las pérdidas de paquetes eran tan altas que los clientes en pocas ocasiones recibían respuestas de sus búsquedas.

5.6.1 Descripción del experimento

En este apartado se describe el experimento realizado sobre el sistema distribuido implementado en PlanetLab.

Un cliente se conecta a cada uno de los nodos de HOverlay y envía una búsqueda de servicio con una probabilidad x de ser satisfecha en los nodos de la overlay. Esto significa que aproximadamente un x% de los nodos de HOverlay tienen el servicio solicitado y, por tanto, pueden satisfacer la búsqueda. Los nodos que satisfacen la búsqueda fueron escogidos aleatoria y uniformemente. Las probabilidades de satisfacer la búsqueda o petición (P_{sr}) utilizadas en la evaluación fueron 1%, 5% y 10%.

Para cada una de las búsquedas realizadas (una por nodo de la overlay), se calculó el tiempo transcurrido en el cliente entre el envío de la búsqueda y la primera respuesta recibida y se guardaron los datos. Cuando un nodo de la overlay recibía una búsqueda y el servicio solicitado estaba disponible en el nodo, éste enviaba una respuesta directa al cliente que le indicaba la disponibilidad del servicio. Para las búsquedas de las que no se recibió ninguna respuesta no se pudieron almacenar datos.

El experimento anterior se realizó el mismo día, 5 veces en 3 slots de tiempo diferentes. Uno fue a las 08:00 a.m. CET, otro a las 13:00 p.m. CET y un tercero a las 21:00 p.m. CET. Así, se ejecutaron un total de 2250 búsquedas desde nodos distribuidos geográficamente por Europa.

5.6.2 Datos e interpretación del experimento

Para representar los datos se utiliza el percentil. Un percentil, $y\% = v$ significa que en y% de las búsquedas de servicio, el tiempo de búsqueda obtenido está por debajo del tiempo v. Así, un percentil del 50% muestra el tiempo de búsqueda suficiente para encontrar el servicio en el 50% de las búsquedas realizadas y un percentil del 100% muestra el tiempo de búsqueda máximo.

La figura 39 muestra el tiempo de las búsquedas de las que se ha obtenido respuesta, en función de la probabilidad de satisfacer la búsqueda (P_{sr}) y del percentil. Los percentiles mostrados son 25%, 50%, 75% y 100% (filas) y la probabilidad de satisfacer la búsqueda (P_{sr}) en los nodos es 1%, 5% y 10% (columnas). Para todas las gráficas, la unidad de tiempo en el eje de

ordenadas es el milisegundo. El eje de abscisas indica el número del experimento ejecutado (1 a 5 son los 5 experimentos para el primer slot de tiempo, 6 a 10 los 5 experimentos del segundo slot de tiempo, y 11 a 15 los 5 experimentos para el tercer slot de tiempo). Dado que las gráficas muestran tiempo de búsqueda, un tiempo bajo es mejor que uno alto. Los valores exactos obtenidos se encuentran en el anexo F.

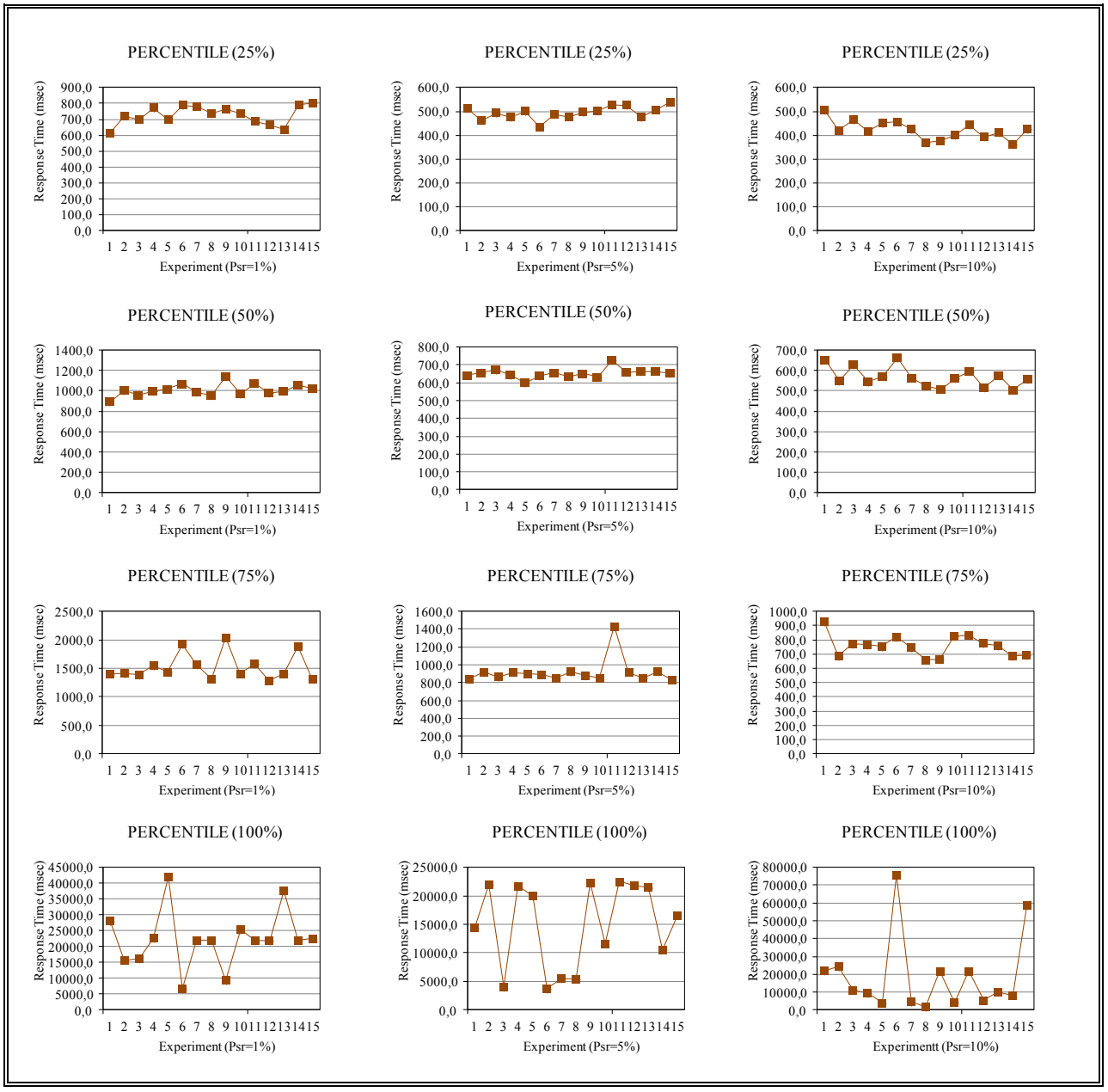

Figura 39: Tiempo de búsqueda en PlanetLab.

Como ejemplo, el primer valor de la primera gráfica (612 milisegundos), significa que del total de las respuestas recibidas de las búsquedas del primer experimento de los 5 del slot de tiempo de las 08:00 a.m. CET, el 25% de las búsquedas no tardaron más de 612 milisegundos.

Fijando un percentil (una fila), vemos que a medida que P_{sr} se incrementa, el tiempo de búsqueda mostrado desciende en la mayoría de los casos. La explicación de este resultado se encuentra en que para una búsqueda con un P_{sr} mayor es más probable que un nodo que satisfaga la búsqueda y, por tanto, es más probable encontrar un nodo antes.

Estableciendo una P_{sr} (columna) vemos que al incrementar el percentil incrementa el tiempo obtenido para la mayoría de las muestras. La explicación de ello es que aumentando el percentil estamos añadiendo casos en los que el tiempo de búsqueda es cada vez mayor.

Para algunas gráficas vemos que se obtienen valores de tiempo bastante dispares en diferentes experimentos. Con respecto a este punto, debe tenerse en cuenta el hecho de que estamos ejecutando las búsquedas en nodos de PlanetLab que ejecutan también procesos de otros usuarios que no se controlan. Así, el tiempo de búsqueda puede verse afectado por otros procesos que se ejecutan en los nodos. También, el tiempo de búsqueda se ve afectado por las condiciones de la red en el momento en el que se realizan los experimentos y estas condiciones son impredecibles.

Por otro lado vemos que para el percentil 100% es para el que se obtienen las mayores diferencias entre los valores obtenidos para distintos experimentos. La explicación de esto radica en el hecho de que en este caso estamos considerando todos los tiempos de búsqueda de las búsquedas de las que se ha obtenido respuesta. Así, sólo con que un nodo esté sobrecargado, o la comunicación entre dos nodos o entre un nodo y el cliente sea lenta, el tiempo mostrado se ve incrementado considerablemente.

En las figuras 40, 41 y 42 se muestran los tiempos de búsqueda en PlanetLab utilizando funciones de distribución acumulada para P_{sr} = 1%, 5% y 10% respectivamente. Para cada una de las figuras, la primera gráfica son las funciones de distribución acumulada de los experimentos 1 a 5 que son los 5 experimentos para el primer slot de tiempo, la segunda gráfica son las funciones de distribución acumulada de los experimentos 6 a 10 que son los 5 experimentos del segundo slot de tiempo y la tercera gráfica son las funciones de distribución de los experimentos 11 a 15 que son los 5 experimentos para el tercer slot de tiempo. El eje de abscisas indica el tiempo de búsqueda expresado en segundos y el eje de ordenadas indica la probabilidad (en tanto por uno). Por ejemplo, si para 1 segundo en el eje de abscisas el eje de ordenadas toma el valor 0,8, significa que en el 80% de los casos la búsqueda no tardó más de 1 segundo.

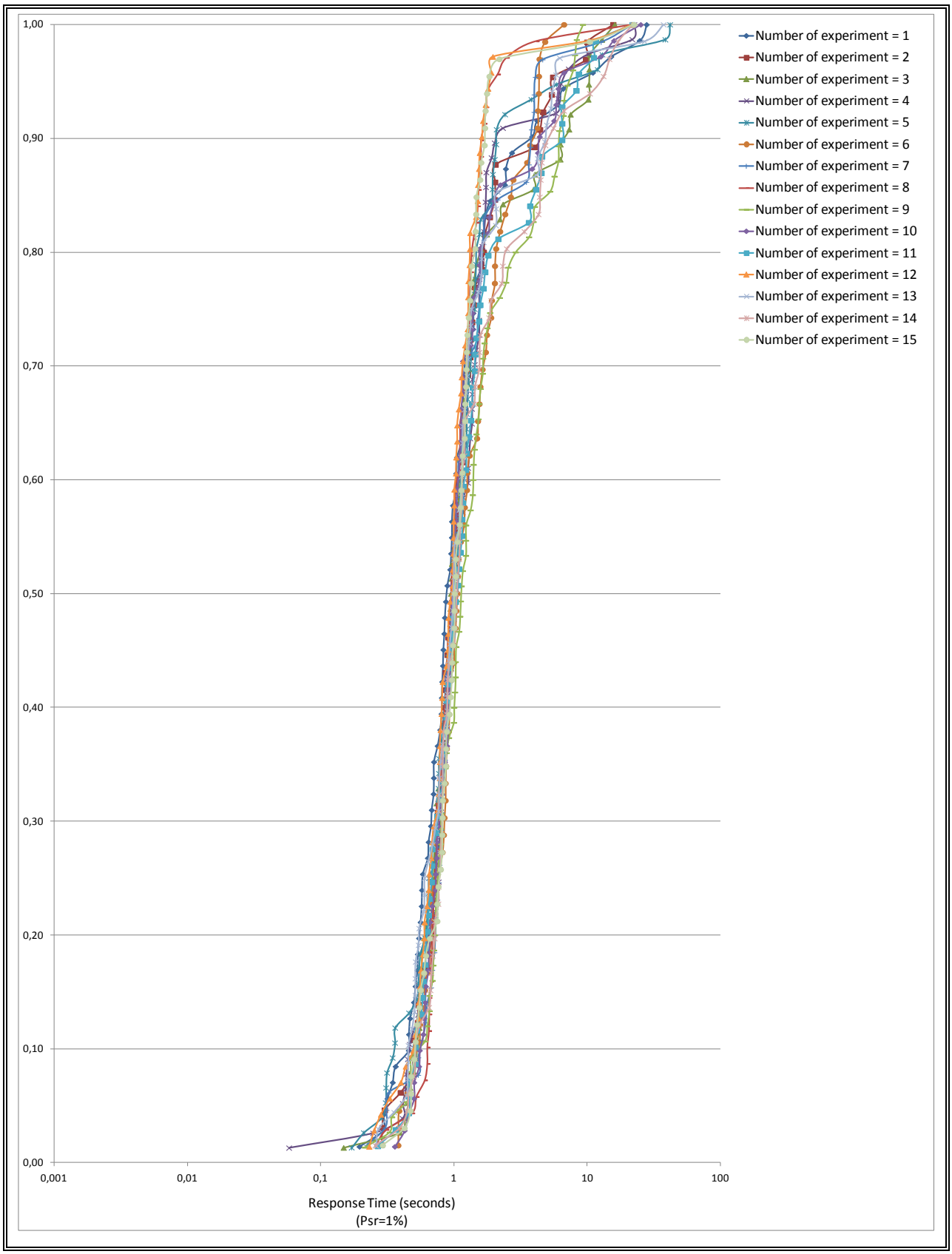

Figura 40: Funciones de distribución acumulada del tiempo de búsqueda (P_{sr} = 1%).

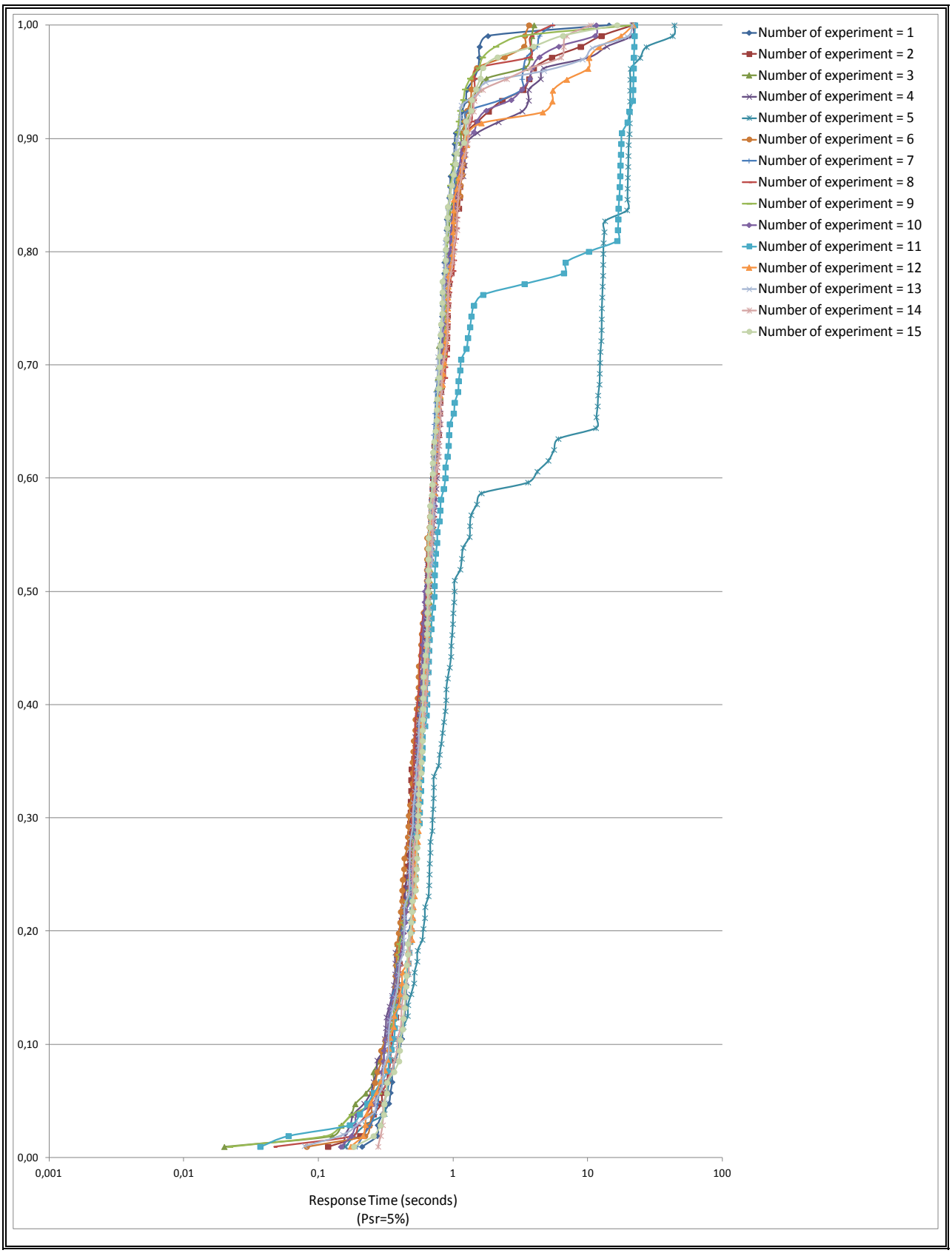

Figura 41: Funciones de distribución acumulada del tiempo de búsqueda (P_{sr} = 5%).

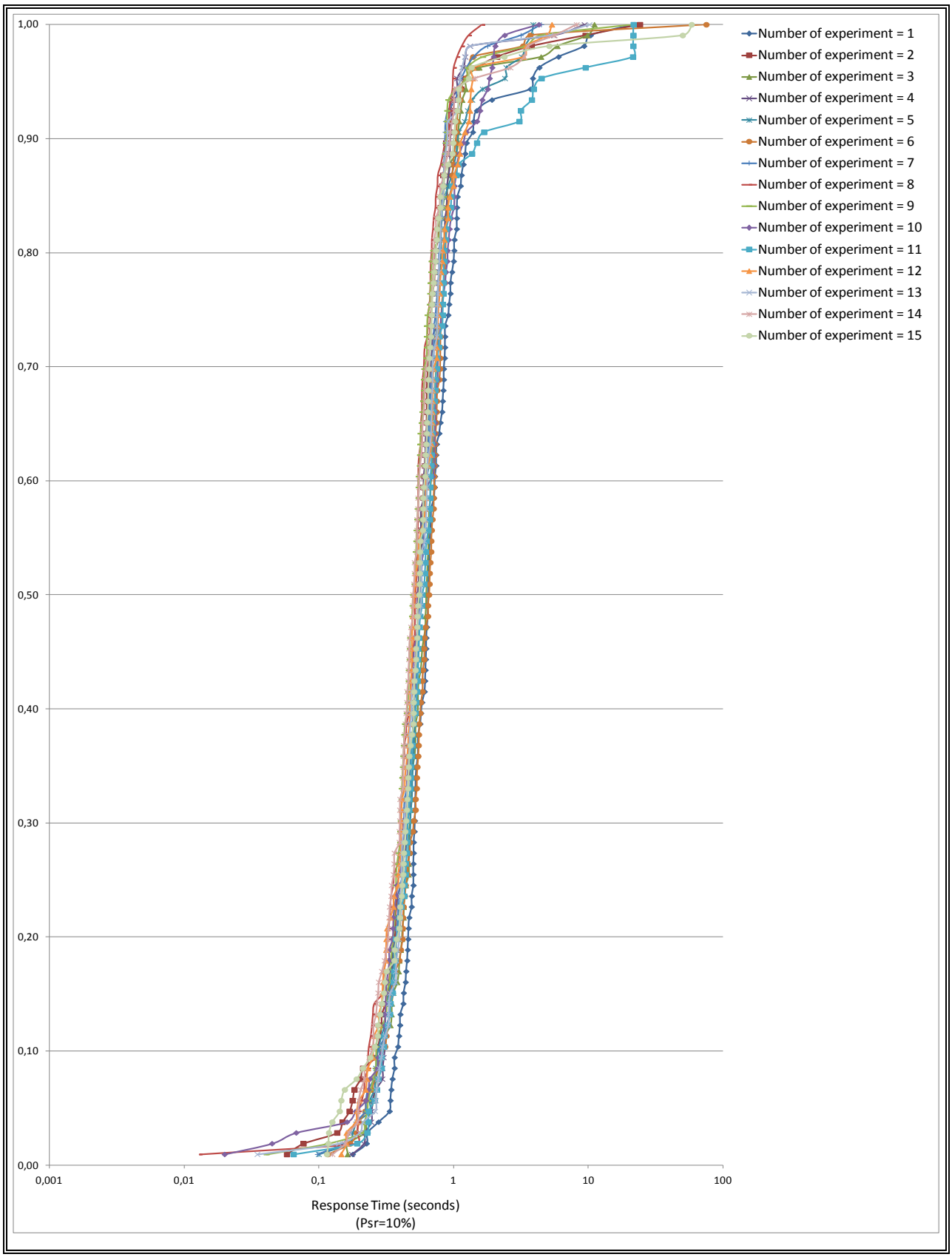

Figura 42: Funciones de distribución acumulada del tiempo de búsqueda (P_{sr} = 10%).

Es importante destacar que no se obtuvieron respuestas para algunas búsquedas. En la figura 43 se muestra el porcentaje de veces que se encontró el servicio para las búsquedas realizadas desde nodos – a priori – vivos (es decir, no tenemos en cuenta en los cálculos las búsquedas realizadas desde nodos que ya sabemos que son no-vivos). Para todas las gráficas, la unidad de tiempo en el eje de ordenadas es %. El eje de abscisas indica el número del experimento ejecutado (1 a 5 son los 5 experimentos para el primer slot de tiempo, 6 a 10 los 5 experimentos del segundo slot de tiempo, y 11 a 15 los 5 experimentos para el tercer slot de tiempo). Dado que las gráficas muestran el porcentaje de veces que se encontró el servicio, un porcentaje más alto es mejor que un porcentaje menor. Los valores exactos obtenidos se encuentran en el anexo F.

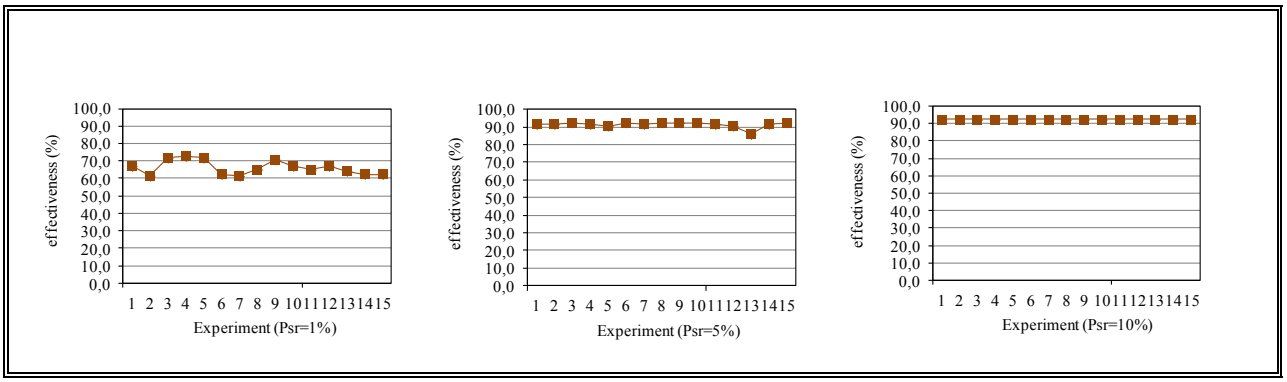

Figura 43: Porcentaje de veces que se encontró el servicio en PlanetLab.

En la figura 43 vemos que a medida P_{sr} aumenta, aumenta también el porcentaje de veces que se encontró el servicio. Para el caso de P_{sr} = 10% es mayor del 90%. Esto se debe a que cuando P_{sr} aumenta, el número de nodos que satisfacen la búsqueda del cliente también aumenta. Cabe destacar que cuando P_{sr} es del 1% los valores mostrados parecen bajos. Pero hay que tener en cuenta que, para un hipercubo 8-dimensional incompleto (usado aquí), cuando P_{sr} es del 1%, sólo alrededor de 1 nodo del hipercubo satisface la búsqueda (quizás ninguno). De estos resultados podemos presuponer que un incremento de P_{sr} incrementa el porcentaje de veces que se encuentra el servicio en una búsqueda – probablemente siendo alrededor del 100% para valores de P_{sr} mayores que 10%. También es probable que, fijado P_{sr}, un incremento en la dimensión del hipercubo incremente la efectividad de la búsqueda, dado que se incrementa el número de nodos que disponen del servicio buscado.

Particularizando para el percentil 50%, que considera el 50% de las búsquedas de las que se ha obtenido respuesta, observamos que para cualquiera de las 3 probabilidades analizadas de satisfacer la búsqueda en un nodo (P_{sr}=1%, P_{sr}=5% y P_{sr}=10%), el tiempo de búsqueda es menor de 1200 milisegundos. Si comparamos estos tiempos de búsqueda con el tiempo de 462 milisegundos, obtenido en un experimento anterior (tiempos de búsqueda por simulación para 614 nodos en el que se ejecutaron aproximadamente 429 búsquedas (614 * 0,7) ya que la probabilidad de fallo (P_f) en ese experimento es del 30%), vemos que el tiempo que ha tardado en el presente experimento un mensaje de búsqueda en llegar a un nodo desde otro nodo más el tiempo que ha necesitado este nodo para ejecutar el Algoritmo-v_a ha debido ser superior a los

tiempos de transferencia entre máquinas obtenidos del proyecto RIDGE que recordemos son los que se utilizaron en el experimento de tiempos de búsqueda por simulación (disponibles en Internet).

Siguiendo con el caso particular del percentil 50%, si ahora comparamos el tiempo de búsqueda de 1144 milisegundos para P_{sr}=1%, 729 milisegundos para P_{sr}=5% y 662 milisegundos para P_{sr}=10%) con el tiempo de búsqueda de 285 milisegundos del experimento que muestra el artículo [42] en su figura 16 que realizó 2880 búsquedas en 180 nodos con un tiempo de retransmisión típico (*typical relay time*) de 60 milisegundos vemos que los tiempos de búsqueda son los esperados y que resultan satisfactorios en un entorno distribuido. En la tabla 5 se muestran el número de nodos, el número de búsquedas, el tiempo de retransmisión típico y los tiempos de búsqueda para el percentil 50% del experimento del artículo referenciado, del actual experimento y del experimento de tiempos de búsquedas por simulación del presente documento.

	CHORD	PLANETLAB	SIMULATION
Number of nodes	180	150	614
Number of lookups	2880	2250	429 (Pf=30%)
Typical delay time (msecs.)	60	not meassured	>60
Time of lookup (msecs.) PERCENTILE(50%)	285	662 (Psr=10%) 729 (Psr=5%) 1144 (Psr=1%)	462 (Psr=1%)

Tabla 5: Comparativa de datos entre experimentos.

En este experimento se ha evaluado v_a y no t_{aux} pero esto nos permite obtener los tiempos de búsqueda peores para t_{aux} (contenido de las tablas vacías). Tampoco se ha evaluado v_d pero hay que tener en cuenta que debido a las condiciones impredecibles en el momento de ejecutar las búsquedas del estado de las máquinas de PlanetLab y de la red, desplegar v_d en un entorno distribuido real no nos hubiera permitido comparar de forma justa v_a y v_d. La mejora que se obtiene en v_a con respecto a v_d puede verse en un experimento presentado anteriormente (tiempos de búsqueda por simulación).

Se concluye de esta evaluación en PlanetLab que aunque se ha evaluado el servicio de descubrimiento en un entorno no dedicado exclusivamente a ejecutar nuestras búsquedas, se obtienen tiempos de búsqueda satisfactorios. A priori, en un entorno dedicado y ejecutando búsquedas desde nodos geográficamente cercanos al cliente (asumiendo que se obtienen tiempos de respuesta más bajos para nodos geográficamente cercanos), el cliente debería obtener tiempos de búsqueda menores. El porcentaje de veces que se encuentra el servicio en las búsquedas también es satisfactoria, mayor del 90%, incluso cuando el número de nodos que disponen del servicio solicitado por el cliente es muy bajo.

VI. CONCLUSIONES Y TRABAJO FUTURO

El trabajo presentado en esta tesis propone un servicio de descubrimiento basado en una overlay en hipercubo y un algoritmo de búsqueda tolerante a fallos para sistemas distribuidos de gran escala. Por gran escala se entiende sistemas formados por cientos, miles o millones de máquinas y por fallos se entiende máquinas que no son accesibles a través de la red, máquinas que están apagadas, máquinas sobrecargadas, etc. Cabe notar que el servicio de descubrimiento de servicios o recursos es crítico en sistemas distribuidos, permitiendo que un cliente pueda localizar un servicio que necesita. En estos sistemas lo normal es que haya fallos en sus componentes y por tanto, es necesario contemplar que estos fallos se producen mientras se realizan búsquedas de servicios, hecho que hemos constatado en una de las evaluaciones del servicio de descubrimiento propuesto escogiendo 150 máquinas distribuidas geográficamente a lo largo de Europa de forma aleatoria sin conocer si se encontraban en estado de fallo o no, resultando que un 24,67% de ellas fallaron o bien porque ya estaban en fallo o porque fallaron durante la evaluación del servicio.

Una overlay en hipercubo permite que el número de saltos para localizar un servicio crezca de forma logarítmica al aumentar el número de nodos presentes en el hipercubo (N) y que por tanto, el tiempo de búsqueda no se degrade considerablemente al aumentar N. También permite que la cantidad de conexiones que un nodo debe mantener con sus vecinos sea acotada creciendo también de forma logarítmica al aumentar N ($n = \log N$ donde n es el número de conexiones que un nodo debe mantener y N el número de nodos presentes). Además, la topología en hipercubo es robusta por carecer de puntos de fallo únicos, ya que cualquier nodo puede iniciar una búsqueda sin que haya ningún punto de información global. Finalmente, el hipercubo permite con su redundancia intrínseca de caminos alternativos para llegar de un nodo a otro nodo, que los fallos de los nodos no degraden la búsqueda considerablemente.

En este trabajo no se considera el mantenimiento de la overlay, es decir, se ha considerado que el hipercubo estaba ya formado y estable. En cualquiera caso, con el objetivo de presentar un servicio de descubrimiento completo, se han mostrado posibles soluciones propuestas por otros autores para abordar esta temática y que pueden aplicarse al servicio de descubrimiento de este trabajo.

La overlay en hipercubo se llama HOverlay y soporta cualquier número de nodos, es decir, el hipercubo que forma no es necesariamente completo y por tanto formado exactamente por $N = 2^n$ nodos (donde n es la dimensión del hipercubo y N el número de nodos del hipercubo). Además en los experimentos realizados no afecta lo incompleto que esté el hipercubo.

El algoritmo de búsqueda tolerante a fallos propuesto se llama Algoritmo-t_{aux} y utiliza la redundancia de la topología en hipercubo para evitar que los nodos en fallo, llamados nodos no-vivos, impidan consultar nodos vivos cuando el servicio no está disponible en un nodo consultado. Además aprende de las búsquedas realizadas anteriormente, lo que le permite en búsquedas futuras poder consultar (si es necesario) aún más nodos vivos.

Para presentar Algoritmo-t_{aux} se presentan también 2 algoritmos más llamados Algoritmo-v_d y Algoritmo-v_a ya que el algoritmo final propuesto es el resultado de un proceso de mejora que se inició con Algoritmo-v_d y que continuó con Algoritmo-v_a. Se presentan los algoritmos previos porque cada uno de ellos introduce una nueva mejora al algoritmo final propuesto. Algoritmo-v_d

utiliza un vector llamado v_d (vector de dimensiones), Algoritmo-v_a usa el vector v_d anterior y un nuevo vector llamado v_a (vector añadido) y finalmente, Algoritmo-t_{aux} utiliza los vectores v_d y v_a anteriores más una tabla que se llama t_{aux} (tabla auxiliar). Diferenciar cada uno de los algoritmos permite al evaluarlos sobre el mismo escenario cuantificar la mejora que introduce cada uno de ellos al Algoritmo-t_{aux}.

Algoritmo-t_{aux} resuelve con la tabla t_{aux} casos poco frecuentes pero devastadores en una overlay (por ejemplo un fallo en un segmento de red), con el vector v_a casos frecuentes y con el vector v_d casos muy frecuentes. En este capítulo y a partir de aquí para referirnos a Algoritmo-v_d, Algoritmo-v_a, Algoritmo-t_{aux} se utilizan los términos v_d, v_a y t_{aux} cuando en el contexto queda claro que nos referimos a algoritmos.

En sistemas distribuidos de gran escala la escalabilidad es una propiedad necesaria ya que cuando un sistema distribuido tiene esta propiedad, al aumentar de tamaño no se degrada considerablemente su funcionamiento y calidad habituales. Para evaluar la escalabilidad de v_d, v_a y t_{aux} se ha calculado el tanto por ciento de nodos que se consultan para distintas probabilidades de satisfacer el servicio solicitado por el cliente obteniendo por ejemplo que cuando el 25% de los nodos tienen el servicio solicitado sólo se consulta el 40% aproximadamente de nodos para un 8-dimensional hipercubo incompleto (orden de 256 nodos) y el 14% para un 15-dimensional hipercubo incompleto (orden de 32768 nodos).

Por otro lado, dado que un sistema distribuido de gran escala puede contener cualquier número de nodos, los algoritmos v_d, v_a y t_{aux} se han evaluado tanto en hipercubos completos como en hipercubos incompletos, cubriendo hasta un millón de nodos aproximadamente para hipercubos completos, y hasta dos cientos treinta y cinco mil nodos aproximadamente para hipercubos incompletos. La evaluación de los algoritmos v_d, v_a nos permite cuantificar la mejora que introducen al Algoritmo-t_{aux}. Además v_d, v_a y t_{aux} se han comparado con 2 búsquedas propuestas por otros autores: búsqueda propuesta por HaoRen et al. y búsqueda en HyperD.

También se ha evaluado la propuesta de esta tesis en términos de tiempos de búsqueda. Además el tiempo de búsqueda se ha obtenido por simulación en sistemas formados por alrededor de mil nodos y en un entorno real distribuido geográficamente. Para el entorno real se ha implementado un sistema formado por 150 máquinas distribuidas geográficamente a lo largo de Europa donde se ejecutaban procesos de otros usuarios que no se controlaban y se han ejecutado búsquedas en condiciones de tráfico real. Las máquinas se escogieron de forma aleatoria sin conocer si se encontraban en estado de fallo o no y un 24,67% de ellas fallaron o bien porque ya estaban en fallo o porque fallaron durante las búsquedas, lo que constata el hecho de que los algoritmos a emplear en sistemas distribuidos deben ser algoritmos tolerantes a fallos.

En relación con los sistemas de descubrimiento P2P estructurados basados en DHTs, las DHTs son escalables y no presentan un único punto de fallo, como el sistema de descubrimiento presentado en esta tesis. También como en las DHTs, en este trabajo cada nodo de la overlay que se propone mantiene información de enrutado de O(log N) nodos, los datos se encuentran distribuidos entre los nodos y una búsqueda de servicios se resuelve en (O(log N) o (O(log N)+1) pasos como máximo. A diferencia de las DHTs, en el sistema que se propone los datos no están previamente organizados entre los nodos por lo que las actualizaciones de datos generan menos

overhead en el sistema. Actualizar un dato en el sistema de descubrimiento propuesto envía un único mensaje, mientras que en una DHT como Chord se envían 2* O((log N)^2) mensajes. Al no estar los datos previamente organizados en la overlay que se propone en esta tesis no se aplican algoritmos de búsqueda basados en enrutamiento y esto hace que a diferencia de las DHTs, donde una búsqueda implica O(log N) mensajes, en el sistema de descubrimiento propuesto una búsqueda implica entre 0 y N mensajes. Así, no organizar previamente los datos entre los nodos, como hace el sistema de descubrimiento propuesto en este trabajo, genera menos *overhead* en la actualización de datos y mayor *overhead* en la búsqueda. Por otro lado, independientemente de la decisión que se tome de organizar previamente o no los datos entre los nodos, el algoritmo de búsqueda propuesto puede aplicarse a sistemas de descubrimiento P2P estructurados basados en DHTs como Chord para realizar búsquedas que no pueden resolverse, utilizando enrutamiento; por ejemplo, buscar en un sistema de almacenamiento de ficheros información de todos los ficheros con una determinada extensión.

Como resultado de las evaluaciones realizadas se ha obtenido que:

- Fijada una probabilidad de satisfacer el servicio en los nodos, disminuye el % de nodos consultados ya que hay más nodos que tienen el servicio y por tanto este se encuentra antes, siendo escalable en ese sentido. Además, el algoritmo propuesto es escalable en términos de almacenamiento de datos, ya que un nodo sólo mantiene información de sus vecinos y no hay ningún nodo en el sistema que tenga información de todos los nodos. También, las búsquedas pueden iniciarse desde cualquier nodo del sistema distribuido y como mínimo para probabilidades de satisfacer el recurso en los nodos superiores al 25% siempre se encuentra el servicio buscado aunque fallen el 30% de los nodos del sistema.
- La búsqueda utilizando Algoritmo-t_{aux} cuando en el sistema se producen fallos tiene una tolerancia a fallos mucho mayor que la búsqueda propuesta por HaoRen. Por ejemplo, en experimentos realizados, en un sistema formado por 131.072 nodos y donde fallen el 10% de estos nodos, si intentamos consultar el mayor número de nodos posibles, el algoritmo propuesto por HaoRen et al. deja de consultar un 43,38% de los nodos, v_d un 1,06%, v_a un 0,25% y t_{aux} un 0,19%. Si ahora aumentamos el número de nodos a 1.048.576 nodos manteniendo que fallen el 10% de estos nodos, el algoritmo propuesto por HaoRen et al. no consulta un 62,32% de los nodos, v_d un 1,14%, v_a un 0,26% y t_{aux} un 0,21%. Ahora, si en el sistema anterior formado por 1.048.576 nodos aumentamos de un 10% a un 20% el % de nodos en fallo, el algoritmo propuesto por HaoRen et al. deja de consultar un 87,75% de los nodos, v_d un 5,02%, v_a un 1,75% y t_{aux} un 1,58%.
- La búsqueda utilizando Algoritmo-t_{aux}, cuando en el sistema se producen fallos y en escenarios donde hay pocos nodos vivos que dispongan del servicio o recurso que el cliente busca, es capaz de encontrar el recurso en un 100% de los casos para las simulaciones realizadas. En cambio, esto no sucede así para la búsqueda propuesta por HaoRen.
- Para el caso particular de HyperD cuando en el sistema se introducen fallos, aplicando v_a (peor caso para t_{aux}) se obtiene el 100% de los servicios presentes para el ejemplo mostrado en su artículo [8] frente a un 55,5% aplicando la búsqueda propuesta en ese artículo.

- Los tiempos de búsqueda obtenidos para un conjunto de 150 máquinas distribuidas a lo largo de Europa con alrededor del 30% de las máquinas en estado no-vivo en el momento de realizar las búsquedas son alentadores, ya que por ejemplo en el 75% de las búsquedas realizadas con éxito el tiempo transcurrido en el cliente desde el envío de la búsqueda hasta obtener la localización de un nodo que satisficiera el servicio solicitado, considerando que sólo un 1% de los nodos tiene el servicio, no ha superado prácticamente los 2 segundos.

Como conclusión el servicio de descubrimiento basado en hipercubos para sistemas distribuidos de gran escala presentado en esta tesis es tolerante a fallos y ha obtenido tiempos de búsqueda alentadores. Cabe recordar que hay que tener en cuenta que las máquinas distribuidas a lo largo de Europa donde se han ejecutado las búsquedas no eran máquinas dedicadas sino que ejecutaban procesos de usuarios distribuidos a lo largo del planeta, por lo que es asumible que se puedan obtener fácilmente tiempos de búsqueda menores y por tanto mejores, por ejemplo, en un entorno dedicado al servicio de descubrimiento. Además el algoritmo propuesto permite que propuestas como la de HaoRen et al. o la de HyperD mejoren sus prestaciones adaptándose a los fallos que se generan en el sistema distribuido.

Como trabajo futuro quedan abiertas distintas líneas como pueden ser:

- Implementar completamente el sistema de descubrimiento presentado en el mismo entorno real que se ha utilizado en este trabajo (PlanetLab) o en otro y obtener datos durante intervalos de tiempo mayores. En este caso sólo habría que elegir alguna técnica de mantenimiento de la overlay ya propuesta o diseñar una nueva.
- Limitar la propagación de la búsqueda de servicios en HOverlay. Por ejemplo, se puede realizar una búsqueda del servicio en sólo una parte de HOverlay y si no se obtienen los resultados deseados realizar una nueva búsqueda en otra parte de HOverlay.
- Extender el presente trabajo añadiendo al servicio de descubrimiento presentado la fase de selección, la última fase del descubrimiento de servicios, donde se selecciona entre los servicios obtenidos aquel que utilizará el cliente. Por ejemplo, el cliente puede seleccionar de entre todas las respuestas recibidas durante un intervalo de tiempo el servicio que está más cerca geográficamente de él, el servicio más barato, etc.

.

BIBLIOGRAFÍA

[1] The European Grid Infrastructure home page, http://www.egi.eu/

[2] Berman, F., Hey, A.J. and Fox, G.C. 2003. GRID Computing, John Wiley & Sons Ltd, Chichester, England.

[3] Ian Foster, Carl Kesselman, and Steven Tuecke. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl*. 15, 3 (August 2001), 200-222. DOI=10.1177/109434200101500302 http://dx.doi.org/10.1177/109434200101500302

[4] Ian Foster, Yong Zhao, Ioan Raicu Shiyong Lu. "Cloud Computing and Grid Computing 360-Degree Compared"

[5] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. 2003. The impact of DHT routing geometry on resilience and proximity. *In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (SIGCOMM '03). ACM, New York, NY, USA, 381-394. DOI=10.1145/863955.863998 http://doi.acm.org/10.1145/863955.863998

[6] Hao Ren, Zhiying Wang, and Zhong Liu. 2006. A Hyper-cube based P2P Information Service for Data Grid. *In Proceedings of the Fifth International Conference on Grid and Cooperative Computing* (GCC '06). IEEE Computer Society, Washington, DC, USA, 508-513. DOI=10.1109/GCC.2006.9 http://dx.doi.org/10.1109/GCC.2006.9

[7] Andi Toce, Abbe Mowshowitz, Paul Stone, Patrick Dantressangle and Graham Bent. September 2011. HyperD: A Hypercube Topology for Dynamic Distributed Federated Databases. ACITA 2011.

[8] Andi Toce, Abbe Mowshowitz, Akira Kawaguchi, Paul Stone, Patrick Dantressangle and Graham Bent. September 18, 2012. HyperD: Analysis and Performance Evaluation of a Distributed Hypercube Database. ACITA 2012.

[9] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor and R. Wolski. March 2000. Revised January 2002. A Grid monitoring architecture. Global Grid Forum Performance Working Group.

[10] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor and R. Wolski. 2002. A Grid Monitoring Architecture. Open Grid Forum. GFD.7. INFO.

[11] A. W. Cooke, A. J. G. Gray, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, L. Field, S. M. Fisher, S. Hicks, J. Leake, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny and J. Ryan. 2004. The relational grid monitoring architecture: Mediating information about the grid. *Journal of Grid Computing* (2004) 2: 323–339

[12] Ian Foster. 2005. Globus toolkit version 4: software for service-oriented systems. *In Proceedings of the 2005 IFIP international conference on Network and Parallel Computing* (NPC'05), Hai Jin, Daniel Reed, and Wenbin Jiang (Eds.). Springer-Verlag, Berlin, Heidelberg, 2-13. DOI=10.1007/11577188_2 http://dx.doi.org/10.1007/11577188_2

[13] LDAP, http://en.wikipedia.org/wiki/LDAP

[14] Open Grid Services Infrastructure (OGSI) standard version 1.0. 2003. Global Grid Forum. draft-ggf-ogsi-gridservice-33_2003-06-27.pdf

[15] Booth D, Hass H, McCabe F et al. 2003. Web Services Architecture, W3C, Working Draft, Retrieved from http://www.w3.org/TR/2003/WD-ws-arch-20030808/

[16] Schopf J M, Raicu I, Pearlman L et al. 2006. Monitoring and Discovery in a Web Services Framework: Functionality and Performance of Globus Toolkit MDS4. Technical Report, Mathematics and Computer Science Division, Argonne National Laboratory.

[17] R. Ahmed, N. Limam, J. Xiao, Y. Iraqi, and R. Boutaba. 2007. Resource and service discovery in large-scale multi-domain networks. *Commun. Surveys Tuts*. 9, 4 (October 2007), 2-30.DOI=10.1109/COMST.2007.4444748 http://dx.doi.org/10.1109/COMST.2007.4444748

[18] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. 1999. An architecture for a secure service discovery service. *In Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking* (MobiCom '99). ACM, New York, NY, USA, 24-35. DOI=10.1145/313451.313462 http://doi.acm.org/10.1145/313451.313462

[19] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. 2000. The design and implementation of an intentional naming system. *SIGOPS Oper. Syst. Rev*. 34, 2 (April 2000), 22-. DOI=10.1145/346152.346192 http://doi.acm.org/10.1145/346152.346192

[20] Adriana Iamnitchi and Ian T. Foster. 2001. On Fully Decentralized Resource Discovery in Grid Environments. *In Proceedings of the Second International Workshop on Grid Computing* (GRID '01), Craig A. Lee (Ed.). Springer-Verlag, London, UK, UK, 51-62.

[21] Catlett, C. and others. 2008. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications, Advances in Parallel Computing; Volume 16, 2008; High Performance Computing and Grids in Action

[22] The National Science Fundation (NSF) Program Synopsis, http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=5456

[23] Extreme Science and Engineering Digital Environment (XSEDE) https://www.xsede.org/home

[24] Lee Liming, John-Paul Navarro, Eric Blau, Jason Brechin, Charlie Catlett, Maytal Dahan, Diana Diehl, Rion Dooley, Michael Dwyer, Kate Ericson, Ian Foster, Ed Hanna, David L. Hart, Chris Jordan, Rob Light, Stuart Martin, John McGee, Laura Pearlman, Jason Reilly, Tom Scavo, Michael Shapiro, Shava Smallen, Warren Smith, and Nancy Wilkins-Diehr. 2009. TeraGrid's integrated information service. *In Proceedings of the 5th Grid Computing Environments Workshop* (GCE '09). ACM, New York, NY, USA, , Article 8 , 10 pages. DOI=10.1145/1658260.1658271 http://doi.acm.org/10.1145/1658260.1658271

[25] Uniform Interface to Computing Resources (UNICORE), http://www.unicore.eu/index.php

[26] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen. 2007. Advanced resource connector middleware for lightweight computational Grids. *Future Gener. Comput*. Syst. 23, 2 (February 2007), 219-240. DOI=10.1016/j.cam.2006.05.008 http://dx.doi.org/10.1016/j.cam.2006.05.008

[27] F. Ould-Saada, The NorduGrid collaboration, SWITCH Journal 1 (2004) 23–24.

[28] The Estonian Grid home page, http://grid.eenet.ee

[29] The Swedish Grid testbed (SWEGRID) home page, http://www.swegrid.se.

[30] The NorduGrid Collaboration home page, http://www.nordugrid.org/

[31] Gabriele Garzoglio, Tanya Levshina, Parag Mhashilkar and Steve Timm. 2009. ReSS: A Resource Selection Service for the Open Science Grid. Grid Computing (2009), Part III, 89-98, DOI: 10.1007/978-0-387-78417-5_8

[32] The Open Science Grid home page, http://www.opensciencegrid.org/

[33] The FermiGrid home page, http://fermigrid.fnal.gov/

[34] Youcef Saad and Martin H. Schultz. 1988. Topological Properties of Hypercubes. *IEEE Trans. Comput*. 37, 7 (July 1988), 867-872. DOI=10.1109/12.2234 http://dx.doi.org/10.1109/12.2234

[35] Dong Xiang, Member, IEEE, Ai Chen, and Jie Wu, Senior Member, IEEE. 2003. Reliable Broadcasting in Wormhole-Routed Hypercube-Connected Networks Using Local Safety Information. *IEEE transactions on reliability*. Vol. 52, no. 2 (June 2003)

[36] M. Y. Chan and S. J. Lee. 1993. Fault-Tolerant Embedding of Complete Binary Trees in Hypercubes. *IEEE Trans. Parallel Distrib. Syst*. 4, 3 (March 1993), 277-288. DOI=10.1109/71.210811 http://dx.doi.org/10.1109/71.210811

[37] Jai Eun Jang. 1990. An optimal fault-tolerant broadcasting algorithm for a hypercube multiprocessor. *In Proceedings of the 1990 ACM annual conference on* Cooperation (CSC '90). ACM, New York, NY, USA, 96-102. DOI=10.1145/100348.100363 http://doi.acm.org/10.1145/100348.100363

[38] Chang, Y. 1993. Fault tolerant broadcasting in SIMD hypercubes. *In Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, 1-4 Dec 1993. 348-351. DOI: 10.1109/SPDP.1993.395512

[39] Gangadhar, D.K.;SCAFFOLD - self configuring overlays using metadata hypercubes, Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on Digital Object Identifier: 10.1109/ICUMT.2010.5676475 Publication Year: 2010, Pages: 875 - 880

[40] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. 2002. HyperCuP: hypercubes, ontologies, and efficient search on peer-to-peer networks. *In Proceedings of the 1st international conference on Agents and peer-to-peer computing* (AP2PC'02), Gianluca Moro and Manolis Koubarakis (Eds.). Springer-Verlag, Berlin, Heidelberg, 112-124.

[41] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario Schlosser, Ingo Brunkhorst, and Alexander Löser. 2003. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. *In Proceedings of the 12th international conference on World Wide Web* (WWW '03). ACM, New York, NY, USA, 536-543. DOI=10.1145/775152.775229 http://doi.acm.org/10.1145/775152.775229

[42] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.* 31, 4 (August 2001), 149-160. DOI=10.1145/964723.383071 http://doi.acm.org/10.1145/964723.383071

[43] Muhamed Sukkar. (2010). *Design and Implementation of a Service Discovery and Recommendation Architecture*. (Doctoral dissertation). Retrieved from http://hdl.handle.net/10012/5526.

[44] Jörg Liebeherr and Tyler K. Beam. 1999. HyperCast: A Protocol for Maintaining Multicast Group Members in a Logical Hypercube Topology. *In Proceedings of the First International COST264 Workshop on Networked Group Communication* (NGC '99), Luigi Rizzo and Serge Fdida (Eds.). Springer-Verlag, London, UK, UK, 72-89.

[45] Buyya, R. i Murshed, M., "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for GRID Computing", Monash University, Melbourne, Australia, 1-37.

[46] The GridSim home page, http://www.gridbus.org/gridsim/

[47]The Simjava home page, http://www.dcs.ed.ac.uk/home/hase/simjava/

[48]RIDGE - Reliable Service Infrastructure in Donation-based Grid Environments (http://ridge.cs.umn.edu/pltraces.html)

ANEXOS

ANEXO A: EVALUACIÓN DE ESCALABILIDAD

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	36,36	37,72	37,95
	9	25,18	26,25	26,33
	10	25,05	26,21	26,30
	11	22,11	23,15	23,25
	12	18,37	19,14	19,23
	13	17,44	18,29	18,40
	14	16,28	17,04	17,16
	15	13,33	13,95	14,05

Tabla 6: Porcentaje de la media de nodos vivos consultados (P_{sr}=25%) (60% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	32,81	34,16	34,29
	9	30,39	31,22	31,33
	10	26,48	27,78	27,91
	11	21,54	22,49	22,70
	12	21,42	22,51	22,71
	13	17,82	18,63	18,81
	14	15,63	16,40	16,58
	15	12,85	13,48	13,62

Tabla 7: Porcentaje de la media de nodos vivos consultados (P_{sr}=25%) (70% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	37,19	38,87	39,16
	9	31,37	32,61	32,78
	10	23,16	24,23	24,42
	11	21,14	22,10	22,33
	12	20,09	21,02	21,27
	13	17,60	18,42	18,61
	14	13,98	14,66	14,84
	15	13,43	14,06	14,25

Tabla 8: Porcentaje de la media de nodos vivos consultados (P_{sr}=25%) (80% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	28,60	29,14	29,23
	9	30,33	31,43	31,63
	10	22,84	23,70	23,89
	11	22,31	23,23	23,41
	12	19,11	19,96	20,18
	13	17,53	18,41	18,64
	14	14,08	14,76	14,95
	15	12,99	13,60	13,78

Tabla 9: Porcentaje de la media de nodos vivos consultados (P_{sr}=25%) (90% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	18,12	18,71	18,76
	9	8,52	8,62	8,62
	10	6,83	6,95	6,95
	11	5,30	5,37	5,38
	12	3,92	3,98	3,99
	13	3,29	3,35	3,35
	14	2,39	2,43	2,43
	15	1,69	1,73	1,73

Tabla 10: Porcentaje de la media de nodos vivos consultados (P_{sr}=50%) (60% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	12,22	12,40	12,40
	9	9,56	9,68	9,68
	10	8,27	8,44	8,45
	11	4,32	4,39	4,39
	12	4,37	4,45	4,45
	13	2,92	2,98	2,98
	14	2,31	2,36	2,36
	15	1,71	1,75	1,75

Tabla 11: Porcentaje de la media de nodos vivos consultados (P_{sr}=50%) (70% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	8,86	8,95	8,96
	9	7,48	7,54	7,54
	10	5,19	5,30	5,31
	11	4,98	5,07	5,08
	12	4,17	4,26	4,27
	13	2,85	2,90	2,90
	14	1,96	2,00	2,01
	15	1,59	1,62	1,63

Tabla 12: Porcentaje de la media de nodos vivos consultados (P_{sr}=50%) (80% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	11,79	11,96	11,97
	9	11,35	11,65	11,67
	10	5,65	5,76	5,76
	11	4,19	4,26	4,26
	12	3,34	3,41	3,41
	13	2,86	2,92	2,93
	14	1,91	1,95	1,96
	15	1,41	1,44	1,45

Tabla 13: Porcentaje de la media de nodos vivos consultados (P_{sr}=50%) (90% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	5,08	5,08	5,08
	9	2,70	2,70	2,70
	10	2,65	2,67	2,67
	11	1,03	1,03	1,03
	12	0,84	0,84	0,84
	13	0,54	0,55	0,55
	14	0,32	0,32	0,32
	15	0,21	0,21	0,21

Tabla 14: Porcentaje de la media de nodos vivos consultados (P_{sr}=75%) (60% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	5,62	5,63	5,63
	9	3,37	3,39	3,39
	10	1,57	1,57	1,57
	11	1,22	1,22	1,22
	12	0,73	0,73	0,73
	13	0,44	0,44	0,44
	14	0,27	0,28	0,28
	15	0,17	0,17	0,17

Tabla 15: Porcentaje de la media de nodos vivos consultados (P_{sr}=75%) (70% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	4,09	4,09	4,09
	9	3,24	3,25	3,25
	10	1,78	1,79	1,79
	11	1,02	1,03	1,03
	12	0,70	0,71	0,71
	13	0,38	0,38	0,38
	14	0,26	0,26	0,26
	15	0,16	0,16	0,16

Tabla 16: Porcentaje de la media de nodos vivos consultados (P_{sr}=75%) (80% ocupación).

	Dim.	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of queried nodes (%)	8	4,68	4,69	4,69
	9	2,26	2,26	2,26
	10	1,77	1,78	1,78
	11	1,20	1,21	1,21
	12	0,74	0,74	0,74
	13	0,39	0,40	0,40
	14	0,23	0,23	0,23
	15	0,16	0,16	0,16

Tabla 17: Porcentaje de la media de nodos vivos consultados (P_{sr}=75%) (90% ocupación).

ANEXO B: EVALUACIÓN EN HIPERCUBOS COMPLETOS

		N = 16.384 (n= 14)			
		Average of failed paths (%)			
		HaoRen (Dim 14)	Algorithm-vd (Dim 14)	Algorithm-va (Dim 14)	Algorithm-taux (Dim 14)
Probability of failure (Pf) (%)	0	0,00	0,00	0,00	0,00
	10	55,04	1,13	0,26	0,20
	20	81,35	5,09	1,74	1,57
	30	85,09	12,16	5,75	5,31
	40	93,09	23,09	13,72	13,24
	50	96,79	42,57	30,64	30,12

Tabla 18: Porcentaje de la media de *failed paths* para H^{14}.

		N = 131.072 (n= 17)			
		Average of failed paths (%)			
		HaoRen (Dim 17)	Algorithm-vd (Dim 17)	Algorithm-va (Dim 17)	Algorithm-taux (Dim 17)
Probability of failure (Pf) (%)	0	0,00	0,00	0,00	0,00
	10	43,38	1,06	0,25	0,19
	20	76,38	4,91	1,71	1,48
	30	89,20	12,40	5,90	5,49
	40	95,02	24,12	14,34	13,87
	50	98,51	41,36	29,80	29,48

Tabla 19: Porcentaje de la media de *failed paths* para H^{17}.

		N = 1.048.576 (n= 20)			
		Average of failed paths (%)			
		HaoRen (Dim 20)	Algorithm-vd (Dim 20)	Algorithm-va (Dim 20)	Algorithm-taux (Dim 20)
Probability of failure (Pf) (%)	0	0,00	0,00	0,00	0,00
	10	62,30	1,14	0,26	0,21
	20	87,75	5,02	1,75	1,58
	30	95,48	12,50	5,91	5,63
	40	98,20	24,89	14,89	14,61
	50	99,26	42,02	30,36	30,09

Tabla 20: Porcentaje de la media de *failed paths* para H^{20}.

ANEXO C: EVALUACIÓN EN HIPERCUBOS INCOMPLETOS

		Pf = 10 %, 60 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	33,49	3,36	1,57	0,70
	9	43,18	5,52	2,86	1,81
	10	35,34	1,38	0,56	0,38
	11	41,53	2,58	1,17	0,85
	12	43,87	2,05	0,80	0,50
	13	49,04	2,43	1,03	0,61
	14	49,61	1,95	0,74	0,44
	15	53,09	2,02	0,75	0,45
	16	54,82	2,09	0,81	0,46

Tabla 21: Porcentaje de la media de *failed paths* (P_f=10%, 60% ocupación).

		Pf = 10 %, 70 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	32,39	1,77	0,70	0,47
	9	35,24	2,80	1,05	0,41
	10	40,32	2,38	0,92	0,56
	11	44,20	2,67	1,08	0,66
	12	43,98	1,94	0,72	0,42
	13	49,61	2,41	0,90	0,44
	14	52,86	2,33	0,82	0,40
	15	54,10	2,03	0,71	0,37
	16	56,92	2,07	0,73	0,37

Tabla 22: Porcentaje de la media de *failed paths* (P_f=10%, 70% ocupación).

		Pf = 10 %, 80 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	33,93	1,60	0,66	0,18
	9	38,19	2,48	1,05	0,59
	10	38,68	1,76	0,60	0,28
	11	44,37	2,02	0,66	0,36
	12	47,57	2,39	0,87	0,39
	13	48,08	2,10	0,76	0,33
	14	52,42	2,24	0,78	0,34
	15	54,39	2,05	0,69	0,31
	16	57,02	2,00	0,66	0,28

Tabla 23: Porcentaje de la media de *failed paths* (P_f=10%, 80% ocupación).

		Pf = 10 %, 90 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	29,66	1,68	0,36	0,19
	9	32,63	1,49	0,39	0,16
	10	40,56	1,84	0,65	0,28
	11	40,76	1,82	0,56	0,23
	12	46,94	2,02	0,64	0,29
	13	48,05	1,96	0,60	0,24
	14	51,86	1,90	0,58	0,24
	15	52,38	1,71	0,50	0,20
	16	55,48	1,71	0,50	0,20

Tabla 24: Porcentaje de la media de *failed paths* (P_f=10%, 90% ocupación).

		Pf = 20 %, 60 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	51,08	8,51	4,77	3,22
	9	50,36	5,03	2,15	1,76
	10	56,16	6,65	3,12	2,33
	11	61,06	6,50	3,07	2,27
	12	65,39	6,14	2,81	2,15
	13	68,45	6,48	2,93	2,12
	14	73,72	7,16	3,33	2,43
	15	75,76	6,87	3,06	2,23
	16	78,53	6,70	2,94	2,14

Tabla 25: Porcentaje de la media de *failed paths* (P_f=20%, 60% ocupación).

		Pf = 20 %, 70 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	43,35	5,66	2,97	2,24
	9	54,11	8,12	4,38	2,55
	10	54,96	5,26	2,39	1,50
	11	64,37	7,03	3,21	2,02
	12	67,29	7,29	3,37	2,12
	13	72,59	8,01	3,68	2,48
	14	73,68	6,99	3,05	1,91
	15	76,66	6,78	2,92	1,89
	16	78,99	6,46	2,70	1,76

Tabla 26: Porcentaje de la media de *failed paths* (P_f=20%, 70% ocupación).

		Pf = 20 %, 80 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	49,77	7,81	3,93	2,25
	9	52,63	6,31	2,72	1,66
	10	56,66	5,46	2,23	1,28
	11	62,91	7,14	3,22	2,00
	12	67,55	6,85	3,00	1,83
	13	72,69	7,20	3,12	1,88
	14	74,10	6,76	2,92	1,74
	15	76,68	6,48	2,70	1,62
	16	78,90	6,44	2,67	1,59

Tabla 27: Porcentaje de la media de *failed paths* (P_f=20%, 80% ocupación).

		Pf = 20 %, 90 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	51,41	7,28	3,16	1,73
	9	52,14	5,46	2,23	1,31
	10	61,97	7,53	3,22	2,10
	11	66,11	7,40	3,25	2,15
	12	68,79	6,90	2,86	1,70
	13	71,23	6,53	2,61	1,50
	14	73,72	6,09	2,44	1,39
	15	76,44	6,04	2,39	1,37
	16	79,12	6,23	2,48	1,41

Tabla 28: Porcentaje de la media de *failed paths* (P_f=20%, 90% ocupación).

		Pf = 20 %, 60 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	59,64	10,65	5,27	4,96
	9	65,20	11,36	5,91	5,19
	10	70,62	15,20	9,15	7,28
	11	74,60	16,31	9,90	8,27
	12	78,50	14,39	8,00	6,36
	13	82,46	14,79	8,06	6,65
	14	85,13	14,43	7,72	6,43
	15	87,44	14,96	8,10	6,58
	16	89,55	14,89	7,95	6,40

Tabla 29: Porcentaje de la media de *failed paths* (P_f=30%, 60% ocupación).

		Pf = 20 %, 70 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	63,22	12,22	5,98	5,20
	9	66,25	15,82	10,05	8,12
	10	69,48	11,93	6,63	5,26
	11	76,96	16,57	9,68	7,44
	12	80,89	15,42	8,58	6,72
	13	83,49	14,46	7,71	6,02
	14	86,03	14,81	7,85	5,93
	15	88,00	14,75	7,82	5,86
	16	89,80	14,60	7,65	5,71

Tabla 30: Porcentaje de la media de *failed paths* (P_f=30%, 70% ocupación).

		Pf = 20 %, 80 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	62,31	10,77	5,13	4,00
	9	70,26	15,71	8,54	6,57
	10	74,20	14,52	7,59	5,62
	11	77,76	15,39	8,40	6,07
	12	79,25	13,42	7,02	4,96
	13	82,51	13,65	7,13	5,09
	14	85,86	14,25	7,44	5,31
	15	88,65	14,79	7,69	5,42
	16	90,29	14,73	7,66	5,39

Tabla 31: Porcentaje de la media de *failed paths* (P_f=30%, 80% ocupación).

		Pf = 20 %, 90 % occup.			
	Dim.	HaoRen et al.'s	Algorithm-vd	Algorithm-va	Algorithm-taux
Average of failed paths (%)	8	59,36	9,93	4,77	3,40
	9	70,44	15,05	8,00	6,04
	10	71,72	12,46	6,34	4,66
	11	77,86	14,57	7,64	5,45
	12	79,94	13,04	6,50	4,53
	13	83,31	13,52	6,77	4,62
	14	85,91	13,80	6,98	4,73
	15	88,02	13,69	6,88	4,64
	16	90,12	14,01	7,04	4,76

Tabla 32: Porcentaje de la media de *failed paths* (P_f=30%, 90% ocupación).

ANEXO D: TIEMPOS DE BÚSQUEDA POR SIMULACIÓN

	Algorithm	PERCENTILE (25%)	PERCENTILE (50%)	PERCENTILE (75%)	PERCENTILE (100%)
Time (msecs.)	HaoRen	337,50	463,00	589,50	1139,00
	Alg.Vd	353,75	462,00	578,00	1151,00
	Alg.Va	353,50	462,00	576,50	1151,00
	Alg.Taux	353,50	461,00	574,00	1151,00

Tabla 33: Tiempo de búsqueda (P_{sr}= 1%) (P_f=30%) (60% ocupación).

	Algorithm	PERCENTILE (25%)	PERCENTILE (50%)	PERCENTILE (75%)	PERCENTILE (100%)
Time (msecs.)	HaoRen	241,00	319,00	436,00	896,00
	Alg.Vd	211,75	288,00	378,00	589,00
	Alg.Va	210,50	285,50	376,25	589,00
	Alg.Taux	210,50	285,50	376,25	589,00

Tabla 34: Tiempo de búsqueda (P_{sr}= 1%) (P_f=30%) (75% ocupación).

	Algorithm	PERCENTILE (25%)	PERCENTILE (50%)	PERCENTILE (75%)	PERCENTILE (100%)
Time (msecs.)	HaoRen	210,00	295,00	394,00	1147,00
	Alg.Vd	198,75	267,00	346,50	693,00
	Alg.Va	197,50	266,00	346,50	693,00
	Alg.Taux	197,50	265,50	344,50	693,00

Tabla 35: Tiempo de búsqueda (P_{sr}= 1%) (P_f=30%) (90% ocupación).

	Algorithm	60% occupp.	75% occupp.	90% occupp.
Effectiveness (%)	HaoRen	50	80	89
	Alg.Vd	99	100	100
	Alg.Va	100	100	100
	Alg.Taux	100	100	100

Tabla 36: Porcentaje de veces que se encuentra un servicio buscado (P_{sr}= 1%) (P_f=30%).

ANEXO E: LISTA DE NODOS DE PLANETLAB UTILIZADOS

Lista de nodos de Noviembre 2011

Node identifier	Hostname
0	ple4.ipv6.lip6.fr
1	ple3.ipv6.lip6.fr
2	onelab-1.fhi-fokus.de
3	planetlab2.ionio.gr
4	host147-82-static.93-94-b.business.telecomitalia.it
5	planetlab1.urv.cat
6	planetlabpc1.upf.edu
7	planetlab1.tau.ac.il
8	planetlab1.thlab.net
9	uoepl1.essex.ac.uk
10	planetlab-2.imag.fr
11	ple01.fc.univie.ac.at
12	planetlab2.urv.cat
13	planetlab3.xeno.cl.cam.ac.uk
14	planetlab2.upc.es
15	planetlab01.tkn.tu-berlin.de
16	planetlab01.dis.unina.it
17	onelab6.iet.unipi.it
18	planetlabpc2.upf.edu
19	planetlab2.csg.uzh.ch
20	planetlab2.dit.upm.es
21	planetlab-1.cs.ucy.ac.cy
22	planetlab-3.imperial.ac.uk
23	planetlab1.informatik.uni-wuerzburg.de
24	plab1-itec.uni-klu.ac.at
25	uoepl2.essex.ac.uk
26	planetlab1.dit.upm.es
27	planetlab1.ci.pwr.wroc.pl
28	dplanet1.uoc.edu
29	planetlab1.csg.uzh.ch
30	planetlab-1.imperial.ac.uk
31	planetlab1.fct.ualg.pt
32	planetlab-1.iscte.pt
33	planet1.l3s.uni-hannover.de
34	openlab02.pl.sophia.inria.fr
35	planet2.itc.auth.gr
36	planetlab2.informatik.uni-wuerzburg.de
37	planetlab2.informatik.uni-goettingen.de
38	ops.ii.uam.es
39	planetlab2.tau.ac.il

Node identifier	Hostname
40	planetlab-2.di.fc.ul.pt
41	planetlab1.montefiore.ulg.ac.be
42	planetlab1.sics.se
43	planetlab-node-02.ucd.ie
44	ple02.fc.univie.ac.at
45	planetlab2.fri.uni-lj.si
46	planetlab2.tmit.bme.hu
47	planetlab1.upm.ro
48	planetlab-node-01.ucd.ie
49	planetlab-2.iscte.pt
50	planetlab04.cnds.unibe.ch
51	planetlab1.di.fct.unl.pt
52	planetlab-node3.it-sudparis.eu
53	peeramidion.irisa.fr
54	planetlab2.fem.tu-ilmenau.de
55	planetlab1.s3.kth.se
56	planetlab-4.dis.uniroma1.it
57	planetlab2.ifi.uio.no
58	thalescom-48-41.cnt.nerim.net
59	onelab7.iet.unipi.it
60	planetlab1.cs.aueb.gr
61	planetlab1.rd.tut.fi
62	lsirextpc02.epfl.ch
63	planetlab2.xeno.cl.cam.ac.uk
64	planetlab1.ionio.gr
65	planetlab3.di.unito.it
66	planetlab-um00.di.uminho.pt
67	planetlab2.wiwi.hu-berlin.de
68	planetlab1.research.nicta.com.au
69	planetlab2.eurecom.fr
70	planetlab-2.research.netlab.hut.fi
71	planetlab1.fri.uni-lj.si
72	iraplab2.iralab.uni-karlsruhe.de
73	planet2.unipr.it
74	onelab2.info.ucl.ac.be
75	ple2.tu.koszalin.pl
76	planetlab3.cs.st-andrews.ac.uk
77	planet3.prakinf.tu-ilmenau.de
78	planetlab3.informatik.uni-erlangen.de
79	onelab3.info.ucl.ac.be
80	plab2.ple.silweb.pl
81	planetlab2.di.fct.unl.pt
82	planet2.l3s.uni-hannover.de
83	planetlab4.hiit.fi

Node identifier	Hostname
84	planet2.inf.tu-dresden.de
85	planet2.prakinf.tu-ilmenau.de
86	planetlabeu-2.tssg.org
87	planet1.elte.hu
88	planetlab3.cslab.ece.ntua.gr
89	planet1.servers.ua.pt
90	planet1.inf.tu-dresden.de
91	planetlab4.cs.st-andrews.ac.uk
92	ait21.us.es
93	orval.infonet.fundp.ac.be
94	planetlab-2.tagus.ist.utl.pt
95	planetlab2.sics.se
96	planetlab2.ics.forth.gr
97	planetlab2.research.nicta.com.au
98	planetlab2.esprit-tn.com
99	planetlab1.exp-math.uni-essen.de
100	planet2.servers.ua.pt
101	planetlab2.willab.fi
102	ple6.ipv6.lip6.fr
103	plab3.ple.silweb.pl
104	plab4.ple.silweb.pl
105	planetlab2.upm.ro
106	planetlabeu-1.tssg.org
107	openlab01.pl.sophia.inria.fr
108	gschembra3.diit.unict.it
109	ple2.ise.pw.edu.pl
110	planetlab2.cs.vu.nl
111	planetlab-2.man.poznan.pl
112	planet1.prakinf.tu-ilmenau.de
113	planetlab2.cs.aueb.gr
114	planetlab1.ics.forth.gr
115	planetlab1.informatik.uni-erlangen.de
116	onelab-2.fhi-fokus.de
117	planetlab-3.cs.ucy.ac.cy
118	planetlab-1.research.netlab.hut.fi
119	ple5.ipv6.lip6.fr
120	planet1.itc.auth.gr
121	plab2-itec.uni-klu.ac.at
122	planet-lab-node1.netgroup.uniroma2.it
123	ple1.cesnet.cz
124	planet2.zib.de
125	planetlab1.eurecom.fr
126	planetlab2.ci.pwr.wroc.pl
127	stella.planetlab.ntua.gr

.

Node identifier	Hostname
128	planetlab1.uc3m.es
129	planetlab1.utt.fr
130	planetlab1.aston.ac.uk
131	planetlab3.upc.es
132	planetlab2.uc3m.es
133	ait05.us.es
134	planetlab-um10.di.uminho.pt
135	vicky.planetlab.ntua.gr
136	planetlab1.upc.es
137	planetlab2.fct.ualg.pt
138	ple2.ipv6.lip6.fr
139	plab1-c703.uibk.ac.at
140	plewifi.ipv6.lip6.fr
141	planetlab-1.man.poznan.pl
142	lsirextpc01.epfl.ch
143	planetlab1lannion.elibel.tm.fr
144	planetlab03.cnds.unibe.ch
145	planetlab2.aston.ac.uk
146	planetlab1.tlm.unavarra.es
147	planetlab1.cs.vu.nl
148	planetlab-1.di.fc.ul.pt
149	planetlab3.hiit.fi

ANEXO F: TIEMPOS DE BÚSQUEDA EN PLANETLAB

	Experiment	PERCENTIL (25%)	PERCENTIL (50%)	PERCENTIL (75%)	PERCENTIL (100%)
Response time (msec)	1	612,00	895,00	1403,50	28018,00
	2	722,00	1004,00	1410,00	15747,00
	3	699,25	958,50	1386,50	16161,00
	4	772,00	999,00	1548,00	22663,00
	5	698,75	1017,50	1433,00	42042,00
	6	788,50	1070,00	1923,50	6723,00
	7	782,00	989,00	1566,00	21963,00
	8	736,00	956,00	1313,00	21951,00
	9	764,50	1144,00	2038,50	9314,00
	10	738,00	971,00	1405,00	25329,00
	11	688,00	1075,00	1582,00	21898,00
	12	666,00	982,00	1284,00	21838,00
	13	635,00	997,50	1403,50	37624,00
	14	789,75	1059,50	1891,50	22017,00
	15	803,75	1025,50	1319,00	22535,00

Tabla 37: Tiempo de búsqueda en PlanetLab (P_{sr}= 1%).

	Experiment	PERCENTIL (25%)	PERCENTIL (50%)	PERCENTIL (75%)	PERCENTIL (100%)
Response time (msec)	1	514,00	641,00	839,00	14420,00
	2	461,00	657,00	916,00	21948,00
	3	492,50	672,00	864,50	4006,00
	4	475,00	644,00	915,00	21715,00
	5	500,00	600,00	900,00	20000,00
	6	434,50	639,50	890,00	3673,00
	7	487,00	656,00	851,00	5501,00
	8	477,25	635,50	921,75	5424,00
	9	496,25	651,50	878,00	22219,00
	10	501,75	631,00	853,25	11603,00
	11	526,00	729,00	1431,00	22484,00
	12	525,75	660,00	915,50	21810,00
	13	475,00	664,00	852,50	21448,00
	14	506,00	665,00	924,00	10586,00
	15	537,75	654,00	834,00	16591,00

Tabla 38: Tiempo de búsqueda en PlanetLab (P_{sr}= 5%).

	Experiment	PERCENTIL (25%)	PERCENTIL (50%)	PERCENTIL (75%)	PERCENTIL (100%)
Response time (msec)	1	505,00	652,50	927,50	21977,00
	2	419,25	549,50	685,75	24255,00
	3	464,25	629,50	771,25	11134,00
	4	416,00	544,00	765,25	9405,00
	5	449,25	571,00	751,50	3913,00
	6	456,25	662,00	819,25	75449,00
	7	424,00	564,50	745,50	4494,00
	8	367,50	524,50	659,50	1631,00
	9	376,25	507,50	662,50	21699,00
	10	399,50	562,00	825,00	4312,00
	11	442,25	598,00	834,50	21754,00
	12	392,25	516,50	775,75	5386,00
	13	410,00	575,50	758,75	10232,00
	14	360,00	505,50	685,75	8221,00
	15	424,50	558,00	693,00	58731,00

Tabla 39: Tiempo de búsqueda en PlanetLab (P_{sr}= 10%).

	Experiment	Psr=1%	Psr=5%	Psr=10%
effectiveness (%)	1	66,98	91,51	92,45
	2	61,32	91,51	92,45
	3	71,70	92,45	92,45
	4	72,64	91,51	92,45
	5	71,70	90,57	92,45
	6	62,26	92,45	92,45
	7	61,32	91,51	92,45
	8	65,09	92,45	92,45
	9	70,75	92,45	92,45
	10	66,98	92,45	92,45
	11	65,09	91,51	92,45
	12	66,98	90,57	92,45
	13	64,15	85,85	92,45
	14	62,26	91,51	92,45
	15	62,26	92,45	92,45

Tabla 40: Porcentaje de veces que se encuentra un servicio buscado en PlanetLab.