UNIVERSITÉ DE NEUCHÂTEL

Load-balanced Structures for Decentralized Overlays

Thèse

présentée le 29 octobre 2008 à la Faculté des Sciences de l'Université de Neuchâtel pour obtenir le grade de docteure ès science par

Silvia Cristina Sardela Bianchi

Institut d'Informatique

acceptée par un jury composé des personnes suivantes:

MM.:	Prof. Pascal	FELBER	Université de Neuchâtel
	Prof. Peter	KROPF	Université de Neuchâtel
	Prof. Fernando	PEDONE	Université de Lugano
Mme.:	Dr. Maria	POTOP-BUTUCARU	Université Paris 6

Université de Neuchâtel

FACULTE DES SCIENCES
Secrétariat-Décanat de la faculté
- Rue Emile-Argand 11
- CP 158
- CH-2009 Neuchâtel

IMPRIMATUR POUR LA THESE

Load-balanced Structures for Decentralized Overlays

Silvia Cristina Sardela Bianchi

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel,
sur le rapport des membres du jury

Mme M. Gradinariu Potop-Butucaru (Paris),
MM. P. Felber (directeur de thèse), P. Kropf,
F. Pedone (Lugano)

autorise l'impression de la présente thèse.

Neuchâtel, le 1er décembre 2008

Le doyen :
F. Kessler

UNIVERSITE DE NEUCHATEL
FACULTE DES SCIENCES
Secrétariat - décanat de la faculté
Rue Emile-Argand 11 - CP 158
CH-2009 Neuchâtel
felix Kessler

Dedicated to my father José Bianchi Neto
and in memory of my mother Maria Aparecida Sardela Bianchi

Acknowledgments

I would like to thank my advisor Professor Pascal Felber for his constant guidance, encouragement and support during my PhD studies. I am also thankful to Professors Peter Kropf and Maria Gradinariu Potop-Butucaru for their help and advice. Furthermore, I am grateful to Professor Fernando Pedone who kindly accepted to be part of the jury and useful remarks.

My gratitude also goes to Jaroslava Messerli for everything that she has done to help me with the administrative issues. I am also grateful to the Department of Computer Science, University of Neuchâtel for hosting me and giving me very good conditions to work. This thesis was supported by the Swiss National Science Foundation Grants 102819.

I cannot forget my ”friends” and ”collegues” Marc Shiely, Frederic Aubert, Steve Casera, Christian Spielvogel, Raphael Kummer, Lorenzo Lucio Leonini, Sabina Serbu, Etienne Riviére, Claire Fautsch, Ljiljana Dolamic, Heiko Sturzrehm, Damien-Bernard Chappatte and Cristian Gatu. Many thanks for the nice working environment, helpful discussions, kind encouragement and the wonderful time we spent together inside and outside the university.

I am forever indebted to my father José who not only gave his support in the difficult moments, love and patience, but also shared with me his knowledge.

I would like warmly to thank my brother Alexandre, my sister Cintia, my family and friends who always stood for me and gave me their kind encouragement and love. Also, a special thank to my aunt Fernanda for her infinite patience and love. Last but not least, Sassan Djavaheri who has made me a happy person and gave me extra force and encouragement to finish it.

Abstract

Keywords: Peer-to-peer, publish/subscribe, adaptive load-balancing, distributed R-trees, distributed Hilbert R-trees.

In large scale distributed environments, huge amounts of information are exchanged and accessed by a large and dynamic number of users. The information is not only stored in servers, but users also store and share the information. Because of these characteristics, the client/server communication model is not longer adapted for certain types of applications.

As an alternative to the client/server model, new paradigms such as peer-to-peer and publish/subscribe have been proposed. They provide mechanisms to locate the information stored and shared between the users and to disseminate it efficiently to interested users.

In this thesis, we focus on developing efficient lookup mechanisms that avoid bottlenecks in large scale peer-to-peer systems, as well as information dissemination techniques that make effective use of the available resources in the system.

In the first part of this thesis, we propose an adaptive load-balancing solution for structured peer-to-peer systems. The approach aims to balance the request and routing load of the peers under biased request workloads. We balance the routing load through dynamic reorganization of the routing tables and the request load by caching the most popular keys along the lookup paths. These mechanisms significantly improve the distribution of the load, and provide consequently better scalability.

In the second part of this thesis, we focus on achieving scalable and efficient information dissemination. We propose Distributed R-tree overlays and Distributed Hilbert R-trees, which use R-tree-based spatial filters to construct peer-to-peer overlays optimized for selective dissemination of information. We adapt well-known variants of R-trees to organize publishers and subscribers in balanced peer-to-peer networks that minimize the occurrences of false positives while avoiding false negatives. In addition, we implement self-stabilizing algorithms that guarantee consistency despite failures and changes in the peer population.

Resumé

Mots-clés : Pair-à-pair, publication/abonnement, équilibrage de charge, ”R-trees” distribués.

Dans les environnements distribués à large échelle, d'énormes quantités d'informations sont échangées et accédées par de très nombreux utilisateurs. Les informations ne sont pas seulement stockées sur les serveurs, mais également par les utilisateurs qui les partagent. Ces caractéristiques particulières rendent le modèle de communication client/serveur classique inadapté à certains types d'applications émergentes.

De nouveaux paradigmes ont été proposés comme alternatives, tels la communication pair-à-pair ou le modèle ”publication/abonnement” qui permettent efficacement de rechercher les informations partagées sur le réseau et les diffuser aux utilisateurs intéressés.

Dans cette thèse, nous nous concentrons sur le développement de mécanismes de recherche d'information permettant d'éviter les goulets d'étranglement dans les systèmes pair-à-pair à large échelle, ainsi sur la mise au point de techniques de diffusion d'information faisant un usage efficace des ressources disponibles dans le système.

Dans la première partie de cette thèse, nous proposons une solution pour l'équilibrage de charge dans des systèmes pair-à-pair structurés. L'approche vise à répartir le trafic en considérant une distribution de requêtes biaisée. Pour atteindre cet objectif, nous réorganisons dynamiquement les tables de routage et nous copions les informations les plus populaires sur les chemins qui permettent d'y accéder. Ces mécanismes améliorent la répartition de la charge et, par conséquent, permettent un meilleur passage à l'échelle.

Dans la deuxième partie de cette thèse, nous nous concentrons sur le problème de la diffusion d'information. Nous proposons des structures de type ”R-trees” et ”Hilbert R-trees” distribués pour construire un réseau applicatif pair-à-pair optimisé pour la diffusion sélective d'information. Nous adaptons les variantes de R-trees pour organiser les éditeurs et les abonnés dans un réseau structuré équilibré qui limite l'occurrence de faux positifs

tout en évitant les faux négatifs. En outre, nous mettons en oeuvre des algorithmes auto-stabilisant pour garantir la cohérence du système en dépit de pannes ou de variations importantes dans la population des pairs.

Contents

List of Figures

List of Tables

CHAPTER

1 Introduction

1.1 Context

Distributed systems may involve hundreds of thousands of geographically dispersed nodes. The most popular example of a distributed environment is the Internet. The Internet provides a high amount of information that is used by numerous heterogeneous devices such as workstations, cell phones, laptops or personal digital assistants. Because of increased processing power and memory capacity as well high bandwidth available, it is possible for users not only to access but also to store and exchange this huge amount of information. Therefore, all the information is distributed and disseminated world-wide, i.e., it is no longer only stored on single servers, but it is replicated to the edges of the network.

Due to these characteristics, the client/server communication model is not well adapted for some types of applications in a large scale distributed environment like the Internet. Client/ server systems are based on tightly coupled communication schemes, for example: a client receives all information from only one server. In this model, information is concentrated on a small number of servers that must provide a continuous service with reasonable response time. The main disadvantage of this model is the possible bottleneck at the servers which can also be single point of failure. To cope with these problems, new paradigms have been proposed, such as the peer-to-peer and publish/subscribe mechanisms.

Peer-to-peer systems offer an alternative to the traditional client/server model, where the resources are stored and managed at the edges of the Internet. Nodes proceed by direct communications in order to share the resources. Every node acts both as a client requesting for some resources and, as a server providing resources requested by other clients. These systems are inherently scalable, fully decentralized and self-organized: each node can join

or leave the system at any time.

In contrast to the client/server paradigm, publish/subscribe systems have gained popularity because of the loose interaction between the parties. In this paradigm, subscribers express certain interests in specific events through subscriptions, and publishers produce events that are asynchronously notified to the clients having interests in matching the event. The matching procedure is performed by a notification service, which is also responsible for the event delivery. In this way, publishers are related to subscribers through loosely coupled interactions.

1.2 Motivation

Large scale distributed systems have to deal with the exchange of huge amounts of information, large and dynamic numbers of participants possibly deployed over a large network and scarcity of resources. They have to provide adequate performance to serve all the requests coming from the end-users. A challenging problem in distributed environments is that the information is not distributed uniformly, i.e., the amount of information stored in the nodes differs from host to host. Moreover, information is accessed differently from the point of view of popularity. Some information is more popular than another are, thus being accessed more frequently, while other information is rarely accessed. This can lead to uneven request workloads that may adversely affect individual nodes in the system. In addition, the nodes that participate in forwarding requests to the nodes that hold popular objects may also be overloaded. Request and routing loads can significantly affect system scalability, which we refer to in terms of the number of messages processed per node. A single overloaded node can degrade the whole system's performance, but sharing the load among nodes can enable the system to scale to a higher traffic load.

Another challenge in distributed systems is represented by large scale information dissemination. An enormous volume of information is published daily on the network. From a user's perspective, this complicates the identification of useful information as each user has different interests. Thus, one solution is to disseminate the information to all users that register to receive update information. Nevertheless, this solution generates high amounts of messages in the network affecting the system scalability. In addition, disseminating huge volumes of information to all users regardless of their interest requires users to filter irrelevant information. This approach can lead to overloaded nodes that become hotspots and

decrease the overall performance of the system. Alternatively to that, the information may be filtered and disseminated selectively to the users, which reduces the traffic in the network. Thus, it is important to efficiently disseminate useful information from data sources to the users, depending on the user's interests. Considering that subscribers participate in the event dissemination, by organizing the subscribers based on their interests, the events can be quickly disseminated. However, it is important to consider fairness between the subscribers, i.e., the ones that are interested in more events will participate more in the event forwarding than the ones that are interested in less events.

1.3 Contribution and Organization of the Thesis

The document focuses on large scale information reorganization and dissemination to improve distributed system's scalability. Parts of this thesis were previously published in [15, 16, 17, 97, 98]. Our contributions tried to respond to the challenges mentioned in the previous section. This thesis consists of two major parts. The first part focuses on improving scalability of peer-to-peer systems by balancing the load of the nodes in the system. The second part aims to achieve scalable and efficient information dissemination, where the information is filtered in order to minimize the traffic in the network. By this we mean designing and implementing publish/subscribe systems based on peer-to-peer overlays. The contributions of this thesis can be summarized as follows:

Balancing the Communication Load in Structured Overlays. Chapter 2 provides a general background on peer-to-peer systems and presents some load balancing problems encountered in these systems. The chapter starts with a review of existing systems followed by a closer look on the impact of the overlay structure. Unstructured peer-to-peer systems have no control on the resource distribution and most of times use flooding as a search mechanism. In contrast, structured peer-to-peer systems define specific resource placement and use more efficient search protocols, which rely on the implementation of a distributed data structure (e.g., hash table). The remaining of the chapter discusses several load imbalance aspects that may arise in peer-to-peer systems, more precisely, in a specific kind of structured overlays. We show that due to the lack of flexibility in data placement in structured peer-to-peer, nodes may have more objects than others. Moreover, objects may have different popularity, i.e., some objects might be accessed more frequently than

others. Both situations can lead to uneven request workloads affecting individual nodes in the system.

Chapter 3 presents more precisely the load balancing problem in structured peer-to-peer systems under biased request workloads and our associated load balancing solution. We first study through simulation the impact of skewed request workloads in the system. We show that, with a uniform distribution of the objects in the system and a Zipf-like selection of the requests, the request load of the nodes in the system also follows a Zipf-like distribution. Moreover, the routing load of intermediary nodes that forward the requests, also exhibits similar unbalanced distribution. Consequently, the system shows a heavy lookup traffic load at the nodes responsible for popular objects, as well as at the intermediary nodes on the lookup paths to these nodes. Request and routing loads can significantly affect system scalability.

Based on our analysis, in the remaining of the chapter, we propose a load balancing approach, which takes into account the objects' popularity. Our main solution relies on the routing table reorganization in structured peer-to-peer systems to reduce the routing load of the nodes that receive many requests. In addition, as a complementary solution, we propose to cache the most popular objects at nodes on the lookup path to reduce the request load of the nodes that store these popular objects. We finalize the chapter by presenting simulation results that confirm the efficiency of our approach. These results demonstrate a more balanced traffic and, consequently, improved scalability and performance. Moreover, our solution has a minimum impact on the existing overlay since it does neither affect the topology nor the objects' distribution.

Content-based Publish/Subscribe Overlays. Chapter 4 presents publish/ subscribe systems, focusing on content-based routing approaches and their limitations. More precisely, the chapter includes an overview of the publish/subscribe paradigm and the challenges faced when implementing routing protocols for efficient event dissemination. The chapter starts with the presentation of the main two subscription models: topic-based and content-based. In topic-based systems subscribers register to individual topics and events published on a specific topic are forwarded to all clients registered for this topic. The topic-based approach lacks expressiveness, but can be implemented very efficiently. On the other hand, content-based systems provide a finer granularity: subscribers specify their interests based on a set of criteria applied to events' metadata, which represents that data content. Content-based publish/subscribe is more expressive and flexible, but requires more sophisticated

protocols. Subsequently, we present traditional content routing solutions including their advantages and disadvantages. We show that a broker-based overlay is not suitable for large scale and dynamic publish/subscribe systems. Moreover, we suggest that a content-based publish/subscribe system based on peer-to-peer overlays is more adequate for such environments. After that, we revisit the R-tree [59] characteristics, upon which we base our content-based routing approaches presented in the following two chapters.

In Chapter 5 we propose our Distributed R-tree (DR-tree) overlays that rely on R-trees [59] and R*-trees [14] to construct a peer-to-peer network optimized for selective dissemination of information. Distributed R-trees are a class of content-based publish/subscribe overlays where subscribers and publishers are organized in peer-to-peer balanced structures based only on their interests. We show that our overlays perform efficiently by organizing the subscribers in a distributed and decentralized balanced tree based on the similarity of the subscriptions. Using this structure, events can be quickly disseminated in the system while minimizing the message overhead. Our overlays guarantee subscription and publication times that are logarithmic in the size of the network. We also propose self-stabilizing algorithms in order to guarantee consistency despite failures and changes in the node populations. We show that the recovery time in face of joins/leaves is logarithmic in terms of the network size. We also prove that our overlays remain correct (bounded degree per tree node and balanced overlay) in spite of transient faults, joins, and leaves, and we analytically study their resistance to churn.

In Chapter 6 we propose another class of content routing overlays, called Distributed Hilbert R-trees (DHR-trees). We propose DHR-tree center, DHR-tree min/max and DHR-tree segment, which are based on Hilbert space filling curves [60]. As DR-trees, DHR-trees are a class of content-based publish/subscribe overlays where subscribers and publishers self-organize in a peer-to-peer balanced tree overlay based on the interests of the subscribers. DHR-trees differ from DR-trees in the subscription information and the mechanism used to organize the subscriptions in the overlay. However, DHR-trees rely on the same join, leave and self-stabilizing algorithms as DR-trees that are presented in Chapter 5. We show that, there is a tradeoff between routing accuracy and space cost for indexing subscriptions. The overlay organization of DHR-tree center and DHR-tree min/max are much less complex compared to DR-trees. As DR-trees, DHR-trees are self-organizing and scale to large numbers of subscribers. They also guarantee overlay organization (joins/leaves) and event dissemination logarithmic in the size of the network.

In Chapter 7 we present the performance evaluation of our DR-tree and DHR-tree overlays. We evaluate the routing accuracy of our overlays through the number of false positives generated in the system, i.e., the number of events that a subscriber receives that do not match its interests. The results show that most of our overlays present a very low false positives ratio confirming the efficiency of our approaches. In particular, DR-trees outperform DHR-trees but, DHR-tree center and DHR-tree min/max require less complex computations for the overlay organization and less space to index the data structures than DR-trees. We also evaluate and show the effectiveness of our self-stabilizing algorithms in DR-tree overlays. In addition, we propose two optimization strategies for the DR-tree overlays considering skewed event workloads.

We conclude this dissertation in Chapter 8. We revisit our main two problems studied: load imbalance and inefficient diffusion of information in large scale distributed environment. We conclude with some directions for future work.

Part I

Balancing the Communication Load in Structured Overlays

CHAPTER

2 Background

2.1 Introduction

The peer-to-peer paradigm became popular by file-sharing applications, such as introduced by Napster [81] and Gnutella [50]. However, this paradigm has been already identified in the early stage of the Internet [43]. When the Internet (ARPANET) was first conceived, it already behaved similar to peer-to-peer systems. Hosts were connected without any hierarchy or specific roles (master-slave or client-server) and they had total freedom to cooperate with each other (firewalls were introduced many years later). Even the first applications, such as Telnet and FTP, provide symmetric cooperation.

In the subsequent years, with the increase of popularity of the Internet, the communication models have drastically changed. At that stage client-server applications (such as web-browsing and email) have started dominating the Internet. In the client-server model, clients connect to a server, download (or in general request a service) some data and disconnect. The client-server paradigm plays a predominant role in the Internet.

Meanwhile, the computing power and storage increase at a high rate and ever more bandwidth is available. The information is no more stored only in servers, but also users, now geographically distributed, store and share the information. This new generation of very large scale distributed systems requires new mechanisms to locate the information stored and shared between the user home computers. The peer-to-peer paradigm, which was almost forgotten from the beginning of Internet, returns in the application level with some improvements. This paradigm now addresses important issues in the new generation of the Internet, such as fault-tolerance.

However, with the high amount of information completely distributed across the Internet, some load issues, as in traditional client-server model, become critical in this model. Hosts storing large amounts of information or popular content become hotspots when receiving large amounts of requests. Furthermore, the bandwidth consumption to locate this information also increases. In consequence, this promising paradigm requires load balancing mechanisms in order to reduce the stress in individual hosts and, consequently, a better utilization of physical resources (storage and bandwidth).

In the rest of this chapter we introduce in more detail the peer-to-peer paradigm and some challenges related to load balancing. In Section 2.2 we review the concept of peer-to-peer and the main differences between unstructured and structured systems. In the next section (Section 2.3) we take a closer look at structured peer-to-peer systems and we discuss the different aspects of load problems that may arise in such systems. Section 2.4 concludes the chapter.

2.2 Peer-to-Peer Systems

"P2P is a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet [101]."

Peer-to-peer systems offer an alternative to the traditional client-server paradigm for certain applications exploiting resources at the edges of the Internet. The edges or peers have a direct communication in order to share resources that can be content (file sharing, content distribution and information management), communication (messaging, gaming) or computing (CPU sharing). Each peer acts both as a server providing resources and as a client requesting services. Thus, in contrast to the traditional client-server architecture, most of peer-to-peer systems have no centralized control (Napster [81] is a hybrid approach where the indexing is centralized and file exchange is distributed).

The primary focus of peer-to-peer systems was on unsophisticated, but popular applications such as file-sharing. In order to coordinate the resources (e.g., files) completely distributed across the network, the peers are organized in an overlay network. Overlay network is a virtual network of nodes with logical links built on top of physical network. These systems must deal with intermittent participation, i.e., in file sharing normally users connect; download the files that they are interested in and disconnect. Moreover, since the peers run on home computers, there is a high heterogeneity in terms of storage and

bandwidth [46].

The main motivation behind peer-to-peer systems is their ability to scale to large node populations, to self-organize and to tolerate failures in the presence of highly dynamic environments. In this way, peer-to-peer systems become inherently scalable, i.e., they support large amounts of heterogeneous, dynamic and autonomous peers. Heterogeneity results from the absence at any restriction on the peer's capacities. The systems are dynamic and self-organizing since any peer can decide to join or leave the system at any time without need of a global reorganization.

Numerous peer-to-peer systems have been proposed in the past few years. Roughly speaking, they can be classified as either *structured* or *unstructured*. This classification is based on the presence of a data structure of a localization graph as part of the overlay structure or the absence of such a structure.

Unstructured peer-to-peer systems (e.g., Gnutella [50], Freenet [32]) have no precise control over resource (also called object) placement and generally use "flooding" search protocols. In contrast, structured peer-to-peer systems (e.g., Chord [103], CAN [89], Pastry [92]), use specialized placement algorithms to assign responsibility for each object to a specific node, and "directed search" protocols to efficiently locate objects.

2.2.1 Unstructured Overlays

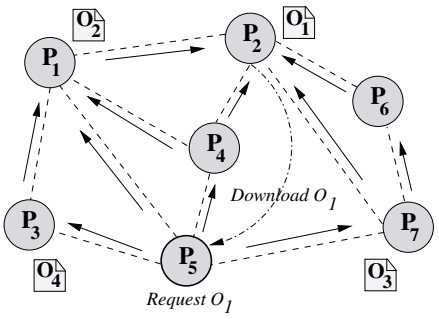


Figure 2.1: Example of a flooding search protocol. The node P_5 requests O_1: it sends the request to all its neighbors, which will forward it to their respective neighbors.

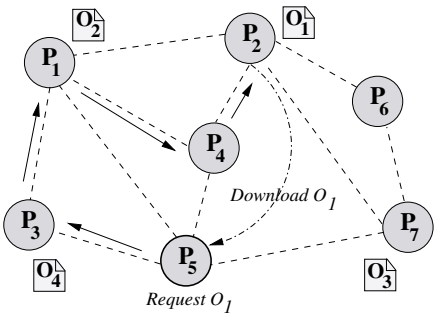


Figure 2.2: Example of a random walk search protocol. The node P_5 requests O_1: it forwards the request to one of its neighbors, which will forward it in the same way.

The first generation of unstructured peer-to-peer systems mainly focused on file sharing. As the name suggests, the peers are not organized in an imposed structure, but in a graph the peers have no knowledge about it.

Any new peer that joins the network contacts an existing peer and copies its neighbor's list. For resource discovering in the network, these systems support different search mechanisms. The straightforward mechanism is to flood the request in the network, as shown in Figure 2.1. The peer forwards the request to all its neighbors, which each of them will forward to its neighbors and so on. If the requested resource is not found, the forwarding procedure will be interrupted after a limited number of retransmissions. Normally, the number of steps can be determined by the Time-To-Live (TTL), as used in Gnutella [50], or Hops-To-Live (HTL), as used in Freenet [32]. The choice of an appropriate value of TTL or HTL affects the message overhead during the search. A high value generates unnecessary traffic and a low value may result on unsuccessful search even if the resource is available in the network. Therefore, the TTL should be high enough to guarantee that the search is successful. This generates a high amount of traffic [73]. In case of a highly connected graph in which nodes have a high degree, i.e., nodes have many communication links, the average number of hops necessary for the search decreases [107]. This allows minimizing the TTL, but it may generate an excessive number of duplicate messages thus increasing even more the overhead. In order to ensure that a search is successful, all the nodes should be addressed, generating $O(N)$ messages, where N is the total number of nodes in the system.

In order to cope with the message overhead, different mechanisms have been proposed to improve the search strategy. These include random walks (see Figure 2.2) [49, 73], percolation-based search [94] and bloom filters-based search [69]. Most of them try to reduce the search time from $O(N)$ to $O(log(N))$.

In random walks (see Figure 2.2), the nodes forward the request to a randomly chosen node. However, this routing approach still requires a mechanism to terminate the request. As in flooding, a TTL is used to limit the number of messages forwarded, and, in addition to that, it may use a check mechanism [73]. The check mechanism tries to reduce even more the traffic by performing periodically checks to the requesting node, by the forwarding nodes, before forwarding the request. While random walks usually generate fewer messages during the search, flooding provides a better response time.

Another approach is the exponential decay bloom filters [69]. This strategy tries to minimize the traffic by spreading probabilistic information about the location of the resources. Hence, peers close to the source node have accurate information about the resources and this information decreases with the distance to the source node.

The percolation [94] approach is based on random walks and relies on bond percolation, which uses probabilistic broadcast search, i.e., the search request is sent to the neighbors with certain probability.

For any of these approaches, resources available at several nodes are more likely to be found, but if the object is stored in only few peers, there is a high probability that the search will be unsuccessful. Since such systems give no control over resource placement, the peers have no knowledge about other peers' contents. Thus, if a search was not successful, it does not mean that the resource is not available in the network. On the other hand, the fact that these systems give no control over nodes' placements, i.e., the communication links are established arbitrary, they can be easily built and maintained.

2.2.2 Structured Overlays

In order to improve the search efficiency, structured peer-to-peer systems use specialized placement algorithms to assign responsibility for each resource to specific peers. The peers maintain information about resources stored by other peers and uses "directed search" protocols that guarantee resource localization with limited number of messages. To that end, each peer has a unique identifier and each resource or object a key mapped to the same identifier space. Typically, the peer identifier is generated from its IP address and the object key from its value, for example a file name. Distributed Hash Tables (DHT) are the most common type of structured peer-to-peer systems. They use consistent hashing functions to assign ownership of each object to a particular peer in the identifier space. The hash space, or key-space is partitioned into ownership zones among the participating peers. The hash function determines the identifier's distribution, i.e., the length of the ownership zone. So, a good hash function results in a uniform distribution of identifiers and keys.

In order to search for a specific key, DHTs offer a lookup service and an interface enabling to store and retrieve a certain data object corresponding to a specific key. Note that the lookup operation is not only used for searching, but also when nodes join and leave the system. Each peer in the overlay maintains a routing table consisting of the identifiers of some neighbor nodes and their respective IP addresses. Whenever a peer wants to search

for some resource, it uses the same hash function to determine the key of the object. This identifier is held by the peer responsible for the object. The routing directs the search towards that the peer holding the object's key, which arrives on the peer responsible for it. When using a common greedy routing algorithm, the peer forwards the request to a neighbor whose ID is closest to the ID of the object in the identifier space. Note that the peer responsible for a key may not necessarily hold a copy of the object, but it may instead store a pointer to the peer that stores the object.

The number of neighbors of a peer has a strong impact on the search cost. Considering a network consisting of N nodes, if each node maintains one neighbor in its routing table, the search cost is $O(N)$. On the other hand, if each peer maintains $N-1$ neighbors, the search cost is $O(1)$. Thus, it is evident that there is a tradeoff between the node's degree (number of neighbors) and the network diameter (average path length). Moreover, a large number of entries in the routing table involve high maintenance cost under high churn (frequent joins and leaves of peers) [56]; and a high network diameter incurs high latency [114]. Ideally, one would like to have the best of both worlds combining short path lengths with a small number of neighbors [56]. As shown in [114], the optimal would be a network with diameter $O(\frac{\log_2(N)}{\log_2(\log_2(N))})$ and routing table size $O(log_2(N))$.

Most DHTs achieve the lookup in $O(log(N))$ steps. However, the routing performance varies and depends on the rules used for associating the resources to the peers, the underlying topology, and the routing protocol.

Mainly there are different routing topologies for structured peer-to-peer overlays (e.g., ring, hypercube or tree). These topologies may be combined with different routing algorithms. However, the topology constrains the flexibility in choosing the neighbors to build the routing tables [56].

Ring. In the ring topology, the node's and object's identifiers are distributed on a ring. An example of this topology is Chord [103] as shown in Figure 2.3. Chord uses a consistent hashing function SHA-1 (Secure Hash Algorithm [61]) to generate an m-bit identifier that is mapped onto the ring. The maximum capacity of the ring is constraint by 2^m identifiers for peers and the objects. Note that m must be large enough to avoid having two nodes (or objects) with the same identifier.

In Chord, the assignment of keys to the nodes is performed as follows: a key k is assigned to the first node whose identifier is equal or greater (the following node clockwise) than k in the identifier space. In contrast, Pastry [92] considers the smallest distance

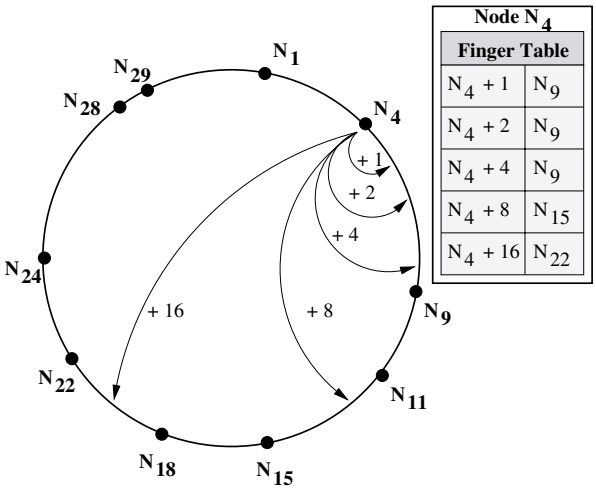


Figure 2.3: Example of a ring topology (Chord).

between the key and the peer in both directions, clockwise and anti-clockwise.

When a node joins or leaves the system, it is necessary only to assign a new ownership to the keys under the responsibility of the leaving or joining node. Thus, with high probability, when the N^{th} node joins the system, only $O(1/N)$ keys are moved to different nodes. In Chord, when a node joins the system, a portion of the keys that are assigned to the node's successor falls under the responsibility of the joining node. In the same way, when the node leaves the system, its keys will be attributed to its successor. Finally, when a node joins or leaves the system, $O(K/N)$ (where K is the number of keys and N the number of nodes in the system) keys will have their responsible node changed.

In order to improve the lookup performance, each node maintains a routing table containing $O(log(N))$ neighbors, used by the node to forward the requests. Normally, the peer selects the neighbors closest to the destination and each hop reduces the distance to the destination by a constant factor.

The i^{th} entry in the routing table points to the first node n that succeeds the node by 2^{i-1} identifiers. As a consequence, the identifier of the entry in the routing table is the successor of $n + 2^{i-1}$, where $1 \leq i \leq m$. Even so, this topology provides some flexibility in the neighbor selection; the i^{th} neighbor may be selected from the range $[2^{i-1}, 2^i)$ instead of picking it at distance 2^{i-1} [56]. Similarly, there is some flexibility in the route selection as well, since there are $log(N)$ options for the first hop, $log(N) - 1$ for the second and so on. Choosing the neighbor from the routing table closest to the destination, implies a short

path length (normally $O(log(N))$). On the other hand, selecting a close neighbor rather than taking the one nearest to the destination generates longer path lengths.

Pastry uses a ring topology with prefix-based routing (to be presented in the following). Therefore, the flexibility for the neighbor and the route selection is the same as for tree topologies based on prefix routing.

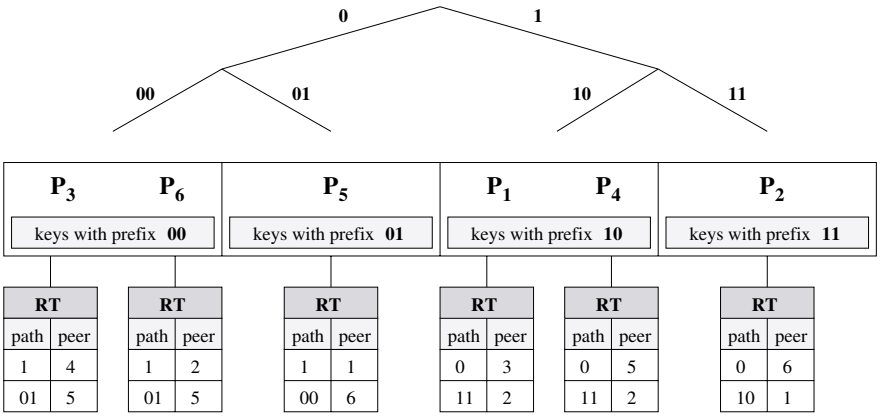


Figure 2.4: Example of a tree topology (P-Grid).

Tree. In this topology, node identifiers are stored in the leaves of a tree of height $log(N)$, where N is the number of nodes. Thus, the leaves represent the identifier space and the height of the subtree the distance between the ownership zones.

The PRR scheme (proposed by Plaxton, Rajaraman and Richa) [85] is one of the first approaches, which relies on a tree. This scheme takes into consideration the network topology such that the requests will not travel further in network distance than the destination node. Pastry (which also uses a ring topology) and Tapestry [120] are based on PRR, thus, they inherit the same property. Nevertheless, PRR is a static version that does not consider churn, i.e., no nodes join and leave the system. For this reason, it is not self-organizing as Pastry and Tapestry.

As in the ring topology, each node has $log(N)$ neighbors, which can be chosen from 2^{i-1} candidate nodes at distance i. The candidate nodes locate in the subtree that shares the first $log(N) - i$ bits with the given node and differs the bit at position $log(N) - i + 1$. This neighbor selection flexibility increases exponentially with the distance i [56]. From the set of candidate nodes, Pastry selects a node based on proximity metrics that reflects static

properties of the underlying physical network, such as Round-Trip-Time (RTT), bandwidth or number of hops. Hence, Pastry takes into account the physical network topology in order to reduce the lookup latency with the cost of additional messages in the join procedure.

The routing procedure operates by fixing a bit or a set of bits (prefix or suffix) that differ between the forwarder node and the destination. Pastry uses prefix-based routing whereas Tapestry uses suffix-based routing. In both cases, the routing table has only one entry possibility to a node closest to a given destination. Thus, Pastry and Tapestry provide neighbor selection flexibility but no route selection flexibility.

Another tree-based approach is P-Grid [2] (see Figure 2.4), which provides both neighbor and route selection flexibility. The routing procedure is similar to prefix-based routing. Upon receiving a request, the node forwards the request to the node that shares the longest common prefix with the key. The routing table is composed of $log(N)$ rows but with multiple entries in each. Since each entry in the routing table has multiple nodes, the current node selects one randomly, and it checks if it is online before forwarding the request. The original version of P-Grid uses a binary tree-like structure for routing requests but, the approach is not limited to a binary tree and it can be extended to a k-ary tree structure [86].

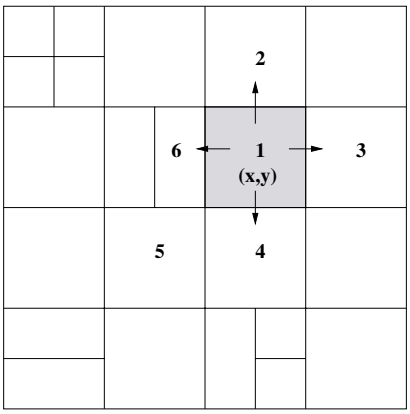


Figure 2.5: Example of a multi-torus topology (CAN).

Multi-Torus. Content Addressable Network (CAN) [89] (as shown in Figure 2.5) uses d-torus topology. The identifier space is a d-dimensional Cartesian space. The coordinate identifier space is dynamically partitioned among the nodes that join the system in the sense that each peer is responsible for a virtual zone, represented by a binary string, within the

identifier space. In this system, the pair (key,value) is mapped onto a coordinate in the identifier space using a hash function. Thus, the node that is responsible for the zone that lies in the corresponding coordinate, stores the (key,value) pair. To retrieve the key, the requesting node applies the same hash function to map the key to the respective coordinate.

When a node joins the system, a zone must be allocated to it. To that end, a zone already allocated to a certain node must be split in two and each half is allocated to the corresponding node. After splitting, the neighbors must be notified of the new node arrival. When a node leaves the system, the zone associated to the node will be merged with the zone of one of its neighbors.

The main motivation of CAN is that tree-based approaches do not perform well under churn; when nodes join or leave the system, a logarithmic number of nodes are affected. In contrast, each node in CAN maintains a constant number of neighbors ($2d$) in its routing table. Each entry in the routing table contains the IP address and the coordinate zone of each of its direct neighbors in the coordinate space. In a d-dimensional space, two nodes are neighbors if their zones overlap in $d - 1$ dimensions. Accordingly, as the state of the nodes is independent from the number of the nodes in the system, this topology does not provide neighbor selection flexibility [56].

To forward a request, a node uses greedy routing: it selects the neighbor where coordinate zone is the closest to the destination, achieving an average path length of $O(d(N)^{1/d})$. Even if there is no flexibility in the neighbor selection to build the routing tables, this topology gives a route selection flexibility as the bits may be corrected in any order (unlike with the tree topology). Thus, a node that has distance $log(N)$ from the destination node, has $log(N)$ first-hop possibilities and it decreases for each successive hop on the route [56].

The main advantage of this topology is that the state maintained by CAN does not depend on the number of nodes in the system, but, on the other hand, the lookup cost is higher $O(N^{1/d})$.

2.3 Load Imbalance

As in traditional distributed systems, peer-to-peer systems may suffer from load-imbalance problems that can degrade the performance. These problems can be caused by different reasons such as skewed object distribution or inefficient routing. Load imbalance is critical and must be treated in order to fairly use the available physical resources.

Unstructured peer-to-peer systems can develop hotspots if certain objects have a high popularity: nodes holding popular objects will receive a large amount of requests and thus become overloaded. However, since in unstructured peer-to-peer systems the nodes have no knowledge about the location of the resources, caching and replication strategies [33, 73] can be used to reduce the incoming traffic in the node holding the popular object.

Another load issue in unstructured peer-to-peer systems is the load resulting from the uneven distribution of the traffic in the network. Traditional unstructured peer-to-peer systems use flooding that overflows the network with messages. Thus, the load of the nodes grows linearly with the number of requests and, consequently, with the size of the network. Hence, upon high request rates, the nodes quickly become overloaded.

Many approaches have been proposed in order to provide better search performance and to reduce the number of nodes to be probed [73, 121]. It is also possible to take advantage of the heterogeneity of the nodes [93] in order to adapt the number of requests processed by the nodes based on their capacity.

As unstructured peer-to-peer systems provide no control on data placement and use random algorithms for routing, simple techniques are enough to cope with imbalance in the system. On the other hand, structured peer-to-peer systems provide deterministic routing algorithms. They generate load imbalance along the following dimensions:

Load imbalance due to object and node placement. The hash function determines the uniformity of the identifiers and keys distribution. Accordingly, depending on the hash function used to generate the identifiers and keys, it may happen that nodes are responsible for longer partition of the identifier space than others or some regions are more densely populated with keys than others. In both cases, this results in a non-uniform identifier distribution. This imbalance can reach an $O(log(N))$ factor [51, 66] depending on the consistent hashing, i.e., the fraction of space owned by a node is exponentially distributed.

Even if the consistent hashing function generates an even distribution of objects and nodes, the churn (nodes join and leave) may destroy the balance of the system. Another aspect that may generate imbalance in the distribution of the keys is the fact that certain applications (e.g. range queries) must preserve the lexicographic ordering. This can lead to nodes being more loaded than others as these applications require to associate semantics to their IDs, constraining the keys distribution.

Request processing hotspots. If the system generates a load imbalance on the objects placement, it may happen that a node holds a large number of objects. In this way, this node will receive a high number of requests that is proportional to the number of objects that it owns. This situation is more likely to appear under uniform request distribution.

However, if we consider skewed request distributions, some objects are more popular than others, thus the nodes that store popular objects will become hotspots. Moreover, comparing unstructured and structured peer-to-peer systems, structured ones incur significantly higher overheads than unstructured ones in case of popular objects that do not have many replicas distributed in the system. This is due to the fact that unstructured systems do not guarantee that the search is successful.

Routing hotspots. Some structured overlays use proximity metrics in order to improve search performance in terms of latency and bandwidth usage. One strategy is to use proximity neighbor selection (PNS), where an entry to the routing table is picked from the set of candidate nodes based on latency or bandwidth; it depends on the desired qualities of the resulting overlay (low delay, high bandwidth, low network utilization). Another option is to use proximity route selection (PRS), where the choice of the next-hop node depends on the distance to the destination [56].

Whatever strategy is used to improve the lookup performance, both PNS and PRS lead to path convergence, i.e., the routing path from nearby nodes to the same destination tends to overlap in the hops nearby the destination. This is due to the greedy routing strategy. For that reason, nodes near to a destination holding a popular object become overloaded with the forwarding traffic as well. In [56], it has been shown that using PNS, the routing paths converge stronger than when using PRS, since the nodes tend to select more often the nodes with smaller latency between each other. In contrast, if we consider a large population of nodes, using PRS, the paths will converge less often since the nodes in the routing table are chosen randomly from a large number of candidates.

With prefix-based routing, the distance that the messages travel increases exponentially while the number of candidate nodes for the next hop decreases with the length of the prefix match. Thus, the probability of path convergence increases at each hop. As mentioned in the previous section, Pastry and Tapestry use proximity metrics to choose neighbors from the set of candidate nodes. This increases the likeliness that nearby nodes choose the same node at a longer distance. So, depending on the locality properties, some nodes will be chosen more often than others. Consequently, they might be overloaded. This problem was

already identified as an open question in [90]. Moreover, Chun et al. [31] measured the node's in-degree in Tapestry with different neighbor selections. They showed that using PNS, this generates an uneven incoming node's degree in contrast to a random neighbor selection that provides a uniform distribution.

2.4 Conclusion

Peer-to-peer (P2P) systems are distributed systems where every node – or peer – acts both as a server providing resources and as a client requesting services. These systems are inherently scalable and self-organized: they are fully decentralized and any peer can decide to join or leave the system at any time.

Numerous peer-to-peer systems have been proposed in the past few years and they can be classified as either *structured* or *unstructured*. Unstructured P2P systems have no precise control over object placement and generally use "flooding" search protocols. In contrast, structured P2P systems use specialized placement algorithms to assign responsibility for each object to specific nodes, and "directed search" protocols to efficiently locate objects.

Peer-to-peer systems must scale to large and diverse node populations and provide adequate performance to serve all the requests coming from all end-users. A challenging problem in structured peer-to-peer systems is that due to the lack of flexibility in data placement and replication – all the objects have specific positions in the overlay – can lead to uneven request workloads that may adversely affect individual nodes in the system. Performance may drastically decrease as overloaded nodes become hotspots in the system.

Several strategies have been proposed to improve load balancing by adjusting the distribution of the objects and the reorganization of the nodes in the system. However, such techniques do not satisfactorily deal with the dynamics of the system, or heavy bias and fluctuations in the popularity distribution. In particular, requests in many P2P systems have been shown to follow a Zipf-like distribution [57], with relatively few highly popular objects being requested most of the times. Consequently, the system shows a heavy lookup traffic load at the nodes responsible for popular objects, as well as at the intermediary nodes on the lookup paths to those nodes. In the next chapter, we present our request and routing load balancing approach under the assumption of unequal object popularity.

CHAPTER

3 Adaptive Load Balancing

3.1 Introduction

This chapter proposes adaptive mechanisms to balance the load in structured peer-to-peer systems under biased request workloads. We first performed simulations, whose results demonstrate that, with a random uniform placement of the objects and a power-law selection of the requested objects, the *request load* on the nodes also follows a Zipf law. More interestingly, the *routing load* resulting from the forwarded messages along multi-hop lookup paths exhibits similar power-law characteristics, but with an intensity that decreases with the hop distance from the destination node. One important point that must be noted here is that application data transfer load (e.g. downloading files) is not considered in this work as this takes place out of band, i.e., externally to the peer-to-peer system.

Request and routing loads can significantly affect system scalability, to which we refer in terms of the number of messages the DHT can process per unit of time. A single overloaded node can degrade the whole system's performance, but sharing the load among nodes can enable the DHT to scale to a higher traffic load.

Based on our analysis, we propose a novel approach for balancing the system load [97, 98] by taking into account object popularity for routing. We dynamically reorganize the routing tables to reduce the routing load of the nodes that have a high request load, so as to compensate for the bias in object popularity. In addition, we propose to complement this approach by caching [17] the most popular objects along the lookup routes in order to reduce the request load in the nodes that hold those objects.

Our solution has the following characteristics:

- *minimum impact on the overlay*: neither changes to the topology of the system, nor

to the association rules (placement) of the objects to the nodes are necessary;

- *low overhead*: no extra messages are added to the system, except for caching. If a node has free storage space, it can dedicate a part of it for caching, which will lead to better load balancing in the system. Other nodes can simply ignore the caching requests;
- *high scalability*: the decision to reorganize routing tables or cache objects are local;
- *high adaptivity*: the routing table reorganization and caching adapt to the popularity dynamics of the objects in the system.

The chapter is organized as follows. In Section 3.2 we introduce the characteristics of the structured peer-to-peer system taken into consideration in this work. Then, we present, in Section 3.3, simulations showing that a Zipf distribution of requests results in an uneven request and routing load in the system. In Sections 3.4 and 3.5 we present, respectively, our approach for popularity-based load balancing and its evaluation. We present some related work in Section 3.6. Finally, Section 3.7 concludes the chapter.

This work was done in collaboration with Sabina Serbu, Peter Kropf and Pascal Felber and published in [17, 97, 98].

3.2 System Design

The basic principle in distributed hash table (DHT)-based abstractions is to associate nodes with objects and to construct distributed routing structures to efficiently locate the objects. The various DHT approaches proposed over the years differ primarily in terms of hash spaces they consider (ring, Euclidean space, hypercube), the rules for associating the objects to the nodes, and the routing algorithm (see also Chapter 2).

In our work, we assume a classical DHT overlay, similar to Pastry [92], composed by N physical nodes and K objects. Each node and object has an *m*-bit identifier with a sequence of digits in base 2^b that determines its position on the ring. The maximum capacity of the ring is 2^m, which corresponds to the maximum number of nodes and objects. A node's identifier is the result of hashing its IP address or its public key. Likewise, an object's identifier is determined by hashing its name; each object is managed by the closest node on the ring. For *consistent hashing*, we used the SHA-1 cryptographic hash function; such

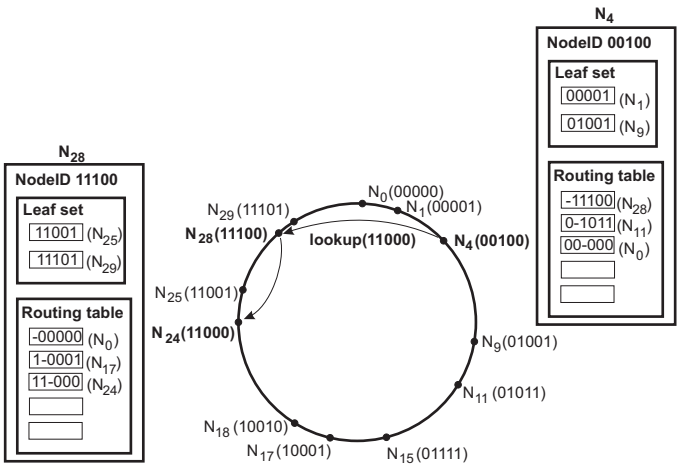


Figure 3.1: System overlay structure and routing principle. Node N_4 sends a request to object O_{24} routed through node N_{28}.

that, with high probability, the distribution of the assigned identifiers on the ring is uniform, i.e., all nodes receive roughly the same number of keys.

For the routing strategy, each node maintains a routing table and a leaf set. The routing tables are composed of $log(_{2^b}(N))$ rows with $2^b - 1$ entries each, where N is the number of physical nodes in the overlay. The i^{th} entry in the routing table of a node n maps to a node that shares a common prefix with length i with the node n and differs in the position $i + 1$. Note that there are typically more than one node satisfying the rule for an entry. In Pastry, the selection of the node for each entry is based on proximity metric. In our system we propose to reorganize the routing tables by selecting the nodes with the lowest load. Our solution can be applied to any DHT with neighbor selection flexibility [56].

Each node also knows its numerically closest neighbors on the ring (predecessors and successors): its leaf set contains L nodes, half of which have numerically smaller and half numerically larger node identifiers.

The lookup procedure is based on prefix routing (also similar to Pastry [92]), with at most $O(log_{2^b}N)$ messages necessary to route a request. When a node receives a request for an object, it forwards the request to the node in its routing table or leaf set whose ID shares the longest common prefix with the object. In Figure 3.1, for example, node N_4 wants to request for the object O_{24}. Hence, it forwards the request to the node N_{28}, in which it is a neighbor in its routing table that shares the longest common prefix with the object O_{24}. In

the same way, the node N_{28} forwards the request to the node N_{24}, which holds the object O_{24}. In this example, $b = 1$ and $L = 2$ with range of 2^5 identifiers for the nodes and the objects in the system.

In this work, we describe our algorithms under the assumption that there is no churn, i.e., no node joins nor leaves the system. We also assume that all the nodes feature the same characteristics (CPU, memory, storage size), all links have the same bandwidth, all objects have the same size. Finally, we do not explicitly take into account the underlying topology.

We do not expect these limitations on the system architecture to reduce our load balancing algorithm's effectiveness.

3.3 Implications of Zipf-like Requests

Similarly to Web requests [19], the popularity of the objects in distributed hash tables typically follows a Zipf-like distribution [57]. This means that the relative probability of a request for the ith most popular object is proportional to $1/i^{\alpha}$, where α is a parameter of the distribution. This bias tends to create hotspots at the nodes that hold the most popular objects.

In case of file sharing applications, many studies have observed that the request distribution has two distinct parts. Very popular files are equally popular, resulting in a linear distribution, and less popular files follow a Zipf-like distribution. This usually happens because of the shared objects immutability in file sharing, which leads clients to request and download each object only once [55, 68, 102].

In both cases, the amount of traffic received and forwarded by some nodes is much higher than for other nodes. In this context, we analyzed in our studies the worst case (Zipf-like distribution) and focused on improving the degraded performance caused by hotspots.

Each node n_i has a capacity for serving requests c_i, which corresponds to the maximum amount of load that it can support. We consider the load as the number of received and forwarded requests per unit of time. Some nodes hold more popular objects than others (i.e., have a higher number of received requests), thus being overloaded, with a load $\ell_i \gg c_i$. Other nodes hold less or no popular objects thus presenting a small load compared to their capacity $c_i \gg \ell_i$. Moreover, with a random uniform placement of the objects and a Zipf-like selection of the requested objects, the *request load* on the nodes also follows a Zipf

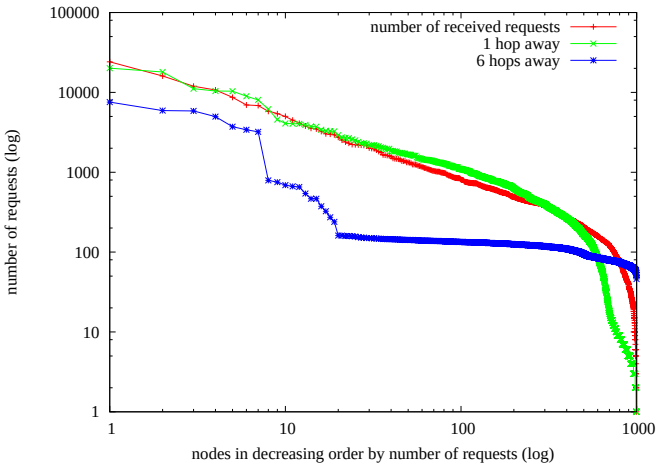


Figure 3.2: Request and routing load of the nodes in the system under Zipf-like requests.

law. Consequently, the *routing load* resulting from message forwarding along intermediary nodes to the popular objects also exhibits power-law characteristics, but with intensity that decreases with the hop distance from the destination nodes.

To better understand this problem, we have performed simulations to gather request load information associated to the nodes in the system. At each node, we tracked the number of requests received by every node, as well as the number of requests forwarded for a node target located i hops away from the destination node (that is, the message will reach its final destination in i hops from the current node).

The value of an i-hop away entry represents the number of requests that a node forwards to destination nodes at a distance of i hops. Figure 3.2 compares the number of requests received by each node, as well as the number of requests it must forward for a destination node that is 1 hop and 6 hops away, using logarithmic scale. The forwarding load also follows a Zipf-like distribution, just like the request load, but with decreasing intensity as the distance from the target node increases.

A node that receives many requests might forward only a few ones, and vice-versa. Figure 3.3 illustrates these observations with numbers obtained from simulating an average-sized overlay network with $1,000$ nodes, $20,000$ objects randomly and uniformly distributed, and $100,000$ requests following a Zipf-like distribution. Node n_i receives only few requests, but it forwards many requests. Conversely, node n_j holds a popular object. It thus receives many requests, but forwards only few requests since it is not on a path

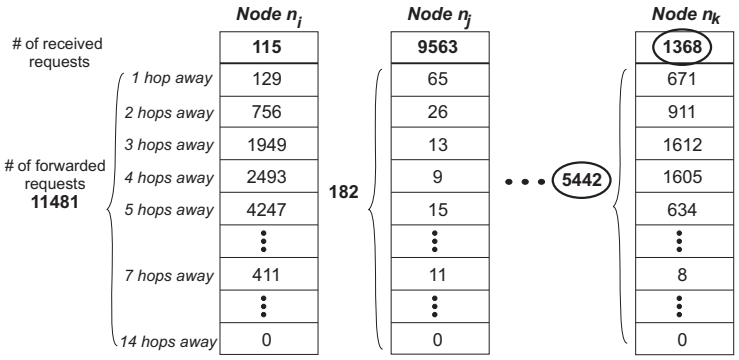


Figure 3.3: Statistics of received and forwarded requests for certain nodes in the system under Zipf-like requests.

to a popular object. Node n_k presents both a high request load and a high routing load. Obviously, nodes n_j and n_k become hotspots.

The DHT's default routing strategy (neighbor and route selection) concentrates on the lookup latency, ignoring the imbalance in the routing traffic that it may generate. Thus, nodes that answer many requests because they hold popular objects may also be loaded by the forwarding traffic if they are on the path to popular objects. On the other side, some nodes do not own popular objects and, equally, are not on the path to popular objects. In this way, these nodes handle low traffic: few requests to process and to forward towards their destinations. Clearly, the problem is the lack of uniformity in the system node's load. A better routing strategy could solve the problem: a uniform load in the system would increase fairness and, more practically, fault tolerance.

In the next section we present our load balancing solution that aims to equilibrate the routing and request load of the nodes in the system.

3.4 Adaptive Load Balancing

In an attempt to address the load balancing issues in DHTs, we established a simplified system model (presented in the previous section) that we used to study different strategies. Based on this model, we present our dynamic routing table reorganization mechanism and caching to achieve an adaptive load balancing.

3.4.1 Routing Table Reorganization (RTR)

The key principle of our approach is to dynamically reorganize the "long range neighbors" in the routing table in order to reduce the routing load of the nodes that have a high request load, so as to compensate for the bias in object popularity.

As previously mentioned, each entry of a routing table can be occupied by any one of a set of nodes that share a common prefix. Pastry selects the node, from the set of candidate nodes, based on proximity metric (such as topology, number of hops, or bandwidth). In our approach, we reorganize the routing tables by choosing the nodes with the lowest (request and forwarding) load in order to offload the most heavily-loaded nodes. The overloaded nodes (as a consequence of a popular object, or too many forwarded requests or both) are removed from the other nodes' routing tables in order to reduce their load. Instead, the entry will contain another node, from the same region (same prefix), which is less loaded. This way, the nodes that have a high request load will have a small forwarding load, and the nodes with low request load will share the forwarding load.

Figure 3.4 shows an example in which the routing table is updated according to topological closest nodes; and Figure 3.5 illustrates the update based on our load balancing mechanism. In the figures, "++" indicates a high load and "−−" a low one. Node N_4 holds a popular object resulting in a high request load. Since it is a heavily-loaded node, it is removed from the other nodes routing tables. Node N_{24} updates its first entry with node N_9, which is less loaded than node N_4. Consequently, the load of node N_4 decreases, and the load of node N_9 increases, thus equilibrating the overall load in the system.

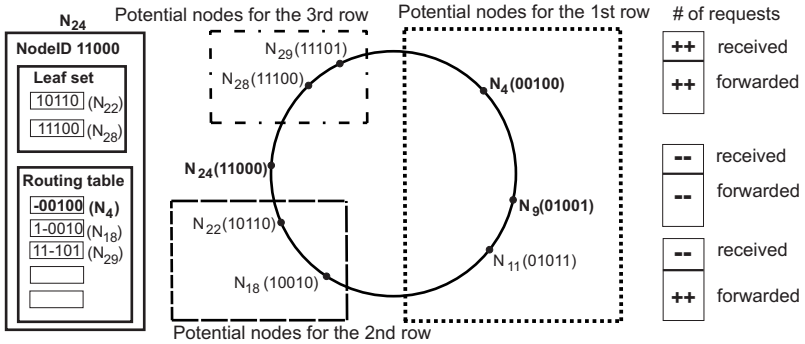


Figure 3.4: Example of a routing table before reorganizing for load balancing.

Each node updates its routing table dynamically, while running the requests, without

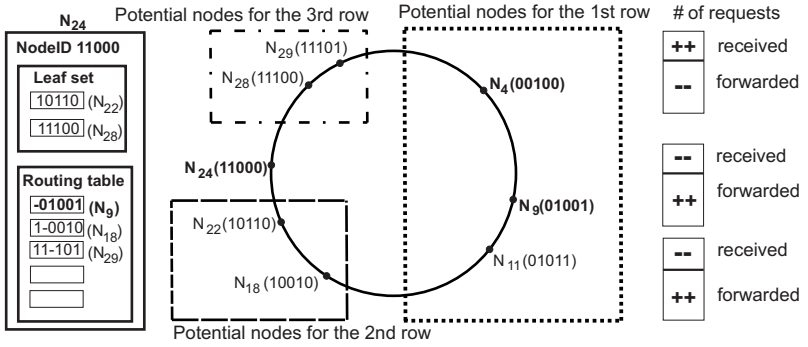


Figure 3.5: Example of a routing table after reorganizing for load balancing.

increasing the number of messages. The algorithm is shown in pseudo-code in Figure 3.6, for a node n_i that forwards a request.

Every node keeps track of the approximate load ℓ_k of each other node n_k in its routing table. Before forwarding or sending a request message, each node adds itself and its load in the message (line 14). The i^{th} node in the request path is thus aware of the load information of i other nodes in the request message. A node n_i that receives the request, besides handling it, uses the load information in the message to possibly update its routing table. Each node n_j in the message can match exactly one entry in the routing table of node n_i. If node n_j has smaller load than the node n_k found in the routing table, n_i updates its routing table by replacing n_k with n_j and setting its load to l_j (lines 6-8).

The load information corresponding to the entries in the routing table of node n_i is usually inaccurate, since the nodes cannot know at each moment the real values for the load of each entry. In order to compensate for this limitation, we use several techniques:

- even if the loads for the two nodes n_j and n_k are equal, the entry is updated, since load l_j is n_j's real load but l_k is only an estimation of n_k's load (line 6-8);
- if n_i receives the load information of a node that is already in its routing table (node n_j is the same as node n_k), its load is updated (line 10);
- when node n_i forwards or sends a request to a node n_k, n_i updates the load information for n_k in the routing table using an estimation e of the load of n_k (line 13).

In our experiments, we use an estimation e of 1, since we know exactly that the load of n_k will increment by at least 1 from the request that n_i forwards.

```
1: upon receive request at node n_i
2:   for each (n_j, ℓ_j) in the message do
3:     entry ← matching entry for n_j in the routing table
4:     n_k ← current node at entry
5:     if n_j ≠ n_k then
6:       if ℓ_j <= ℓ_k then
7:         replace n_k by n_j at entry
8:         store ℓ_j at entry
9:     else
10:      store ℓ_j at entry
11:  if n_i not owner of requested object then
12:    n_k ← next node to forward request
13:    ℓ_k ← ℓ_k + e
14:    add (n_i, ℓ_i) to the request message to be forwarded
```

Figure 3.6: RTR algorithm executed at node n_i

As previously mentioned, the node adds its load information at the end of the data packet, in form of $O(log_{2^b}(N))$ pairs of integers, such that each pair corresponds to a node and its load. This small amount of data will not typically add extra packets at the network layer and should thus have no effect on bandwidth usage. With respect to the CPU load, the routing table update mechanism adds only negligible overhead - in fact, it indirectly decreases CPU load for overloaded nodes by reducing the number of messages to process. Moreover, this algorithm does not add significant complexity to the system since it is based on local load estimations, as well as the information received with each request.

To handle churn, we employ a mechanism similar to that used in Pastry. When a new node n_i joins the system, it populates its routing table by collecting entries from the nodes that the join request goes through. Similarly, the new node's initial load l_i becomes the average of the loads of its new neighbors. Both the routing table and l_i are updated over time according to the routing table reorganization algorithm. Node departures do not require special handling.

As extensions of the algorithm, we might consider a combined metric: proximity (as proposed in Pastry [92]) and load information. This could be a tradeoff between proximity-aware routing and load-aware routing. In addition, we might consider adding load information also to the response message of the lookup for an object. Of course, the more information about the system is available, the better lookup traffic can be balanced.

3.4.2 Caching

The routing table reorganization achieves load balancing in terms of forwarding traffic of the nodes in the overlay, but the traffic resulting from the received requests still leads to a bottleneck at the destination node. In addition to the routing table reorganization, we propose caching as a complementary feature in order to minimize the number of received requests at the nodes holding popular objects. As a consequence, the request traffic for each cached object will be shared among the node owning the object and the nodes holding the replicas.

Basically, there are two ways to initiate caching: by the client that requests the object and by the server that holds the object [73]. Client-initiated caching is not adequate for applications such as file sharing because a client usually requests an object only once. Therefore, in our approach, the server replicates the object to be cached on some other node(s) in an attempt to reduce its request load. Moreover, since most DHT's present path convergence, it becomes interesting to cache the object in the nodes that lay on the lookup path, since they are more likely to serve the request. Hence, when a request arrives at a node that holds a replica of the requested object in its cache, that node can directly respond to the request.

We refer to two kinds of objects that a node holds:

- *owned object*: an object that belongs to the node according to the initial mapping of objects to nodes;
- *cached object*: a replica of an object owned by another node.

The algorithm is shown in pseudo-code in Figure 3.7. We make use of two types of counters for the received requests at each node: a per-object counter (for the number of received requests for each object held by the node) and a per-node counter (for the total number of received requests). The counters are incremented at each received request (lines 2-3). A threshold T is defined for the per-node counter. Each time the threshold is reached (line 4), a weight is computed for each object held by the node based on the per-object counters (lines 5-9); then, the counters are reset (lines 14-16). The most popular object on the node is the object with the highest weight (line 11).

To compute the popularity of an object, we use a weighted moving average, where the weight is computed as a combination of its previous value and the value computed over the last period (line 9). We use a β value of 0.9, such that both terms count in the computation

1: **upon receive caching request** at node n_i from node n_j
2: increment $req\_recv\_counter$
3: increment $req\_recv[requested\_object][n_j]$
4: **if** $req\_recv\_counter = T$ **then**
5: **forall** objects o on node n_i **do**
6: **if** o is the last cached object **then**
7: $w[o] \leftarrow req\_recv[o]/T$
8: **else**
9: $w[o] \leftarrow w[o] * \beta + (req\_recv[o]/T) * (1 - \beta)$
10: **if** n_i is loaded **then**
11: $m\_p\_o \leftarrow o$, where $w[o]$ is max
12: $n_c \leftarrow n$, where $req\_recv[m\_p\_o][n]$ is max
13: send a request to n_c to cache m_p_o
14: **forall** objects o on node n_i **do**
15: $req\_recv[o] \leftarrow 0$
16: $req\_recv\_counter \leftarrow 0$

Figure 3.7: Caching algorithm when node n_i receives a request from n_j

of the object's weight, but the old value (which is a stable information) counts more than the new one. After a caching request has been issued for an object, only the new value is considered (line 7).

For the caching mechanism, the following considerations must be taken into account: storage size of the cache and its policies, when to cache an object and where to store it. In the following we detail all these aspects.

The cache and its policies. Each node has a cache (storage capacity) for C replicas. Whenever a caching request is received and the storage capacity is exhausted, the replica entry with the lowest weight is discarded and the new replica is stored.

When to cache an object. A caching request is issued each time the threshold T is reached in case the node is loaded (lines 10-13). Obviously, if the node is not loaded, no caching request is issued, at least until the next threshold.

To know whether a node is loaded or not, we perform two checks:

- *if the node is globally loaded.* We use the load information of the nodes in the routing table; this is not an up-to-date information, yet a rather good estimation of the load

of some nodes in the system. A node is globally loaded if its load is bigger than the average load of these nodes;

- *if the node has a lot of received requests*. A node would have a balanced load if the number of received requests is equal to the number of forwarded requests divided by the average path length. Therefore, we compute the average path length of the requests that the node received between two consecutive thresholds. To justify a caching request, a node must satisfy the following condition:

$$recv\_requests > fwd\_requests/path\_avg,$$

where the counters for the number of received and forwarded requests are reset after each occurrence of reaching the threshold T.

If both conditions are satisfied, a node will issue a caching request.

Where to store the replica. Since every message contains information about the request path and because DHTs tend to path convergence, the most suitable method is to cache along that path. This can be done (1) on all the nodes in the request path, (2) close to the destination node, (3) at the node that requested the object or (4) randomly. We choose to do the caching at the last node in the request path once the path convergence is more probable in the last hops. This has the advantage that the object is cached in the neighborhood of its owner where the possibility for a request to hit a replica is much bigger than elsewhere in the system.

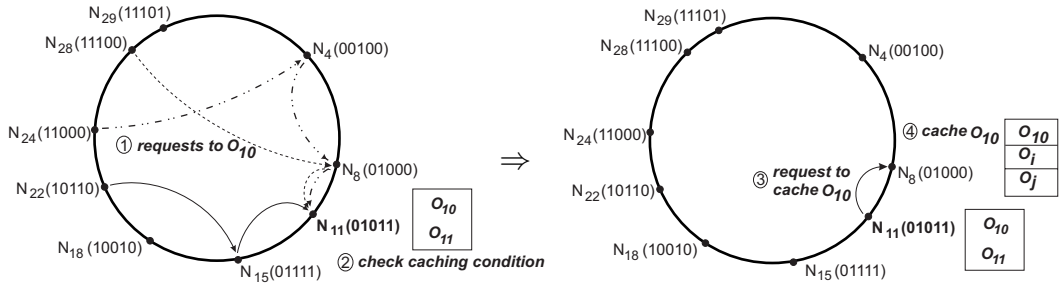


Figure 3.8: Example of the caching method.

The requests for a given object may come from any node in the system, thus the last hop is not necessarily the same. The node chosen for caching the object is the one that most frequently served as last hop for this object in the lookup paths (line 12).

Figure 3.8 presents an example of the caching mechanism. Many nodes send requests for the object O_{10} (step 1). After N_{11} receives the requests, it checks the caching condition (step 2) and, if true, computes its most popular object, O_{10}, and the node where to store a replica, N_8. Then, it issues a caching request for object O_{10} to node N_8 (step 3). Finally, N_8 caches the object O_{10} (step 4).

3.5 Experimental Evaluation

In this section we evaluate our approach by the means of simulations. First, we present results of the experiments when we apply only routing table reorganizations. Next, we analyze the effect of caching. Finally, some statistics with different Zipf distributions are presented.

Our simulations are based on an overlay network with 1,000 nodes, 20,000 objects randomly and uniformly distributed, and 500,000 requests following a Zipf distribution with different α parameters (0.5, 1.0, 2.0). For the routing table reorganization and caching simulations, we have fixed $\alpha = 1.0$. For the routing tables, we have used leaf set size with 4 (default value) and 8 entries. The identifiers are a sequence of $m = 16$ bits in base $2^b = 2$.

3.5.1 Routing Table Reorganization (RTR)

To analyze the load balancing algorithms, we use the same experimental setup while applying different routing strategies:

- **run0**: as a base for comparison, the experiment is run with no load balancing;
- **run1**: dynamic run, where the routing tables are dynamically updated while handling the requests;
- **run2**: same as run1, with the difference that the experiment starts with the routing tables obtained after run1;
- **run3**: static run, with no routing tables updates at all, where the experiment starts with the routing tables obtained after run2.

The results are evaluated after each run of the experiment. The first run (run0) is used for comparison purposes, where the entries in the routing table are selected based on the proximity in the overlay. The first dynamic run (run1) shows the efficiency of the routing table updates. The second dynamic run (run2) simulates the continuation of the first dynamic run, while running the same requests. The purpose of the last run (run3) is to show that in a system with no load balancing strategy, the results are better when starting with optimized routing tables.

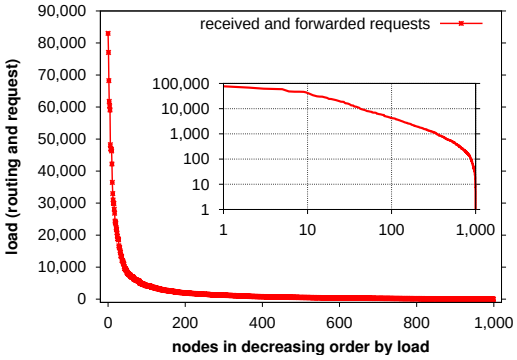


Figure 3.9: Load distribution without load balancing (run0).

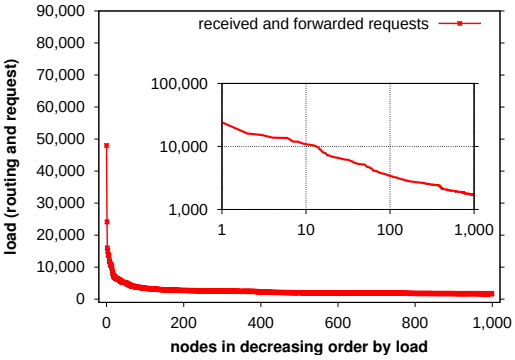


Figure 3.10: Load distribution with load balancing (run2).

The requests follow a Zipf distribution and, as a consequence, the same applies for the load distribution in the system (as can be seen in Figures 3.9 and 3.10), where the nodes are ordered in decreasing order of load.

Figure 3.9 shows the load distribution with no load balancing (run0). The load is not evenly distributed among the nodes: some of the nodes have very high load (the left side of the graph), and other nodes have just a small load or no load at all (the right side of the graph).

Figure 3.10 shows the load distribution in the same overlay with a Zipf request distribution, but with our routing table reorganization, after the second dynamic run of the experiment (run2). As shown in the graph, the highest load decreases by half. Moreover, the load in Figure 3.9 tends to 0, while in Figure 3.10 it remains almost constant (approximately from node 300), showing that most of the nodes have the same load. The improvement factor is even more visible with logarithmic scales (inset graphs in Figures 3.9 and 3.10).

In order to better perceive the load distribution for the most loaded nodes, Figures 3.11 and 3.12 show the same data as Figures 3.9 and 3.10 for the first 300 nodes. They also

show the number of received requests per node.

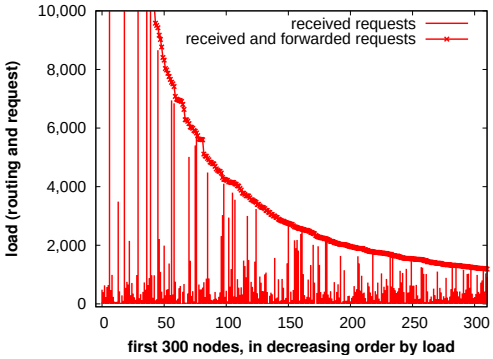


Figure 3.11: The 300 most loaded nodes, without load balancing (run0).

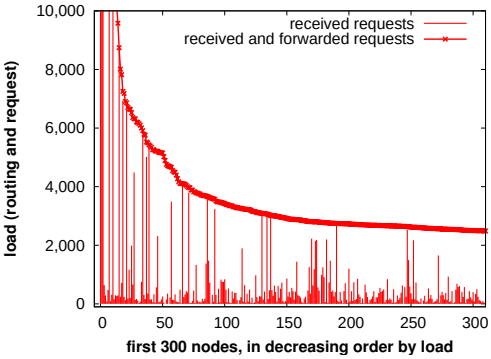


Figure 3.12: The 300 most loaded nodes, with load balancing (run2).

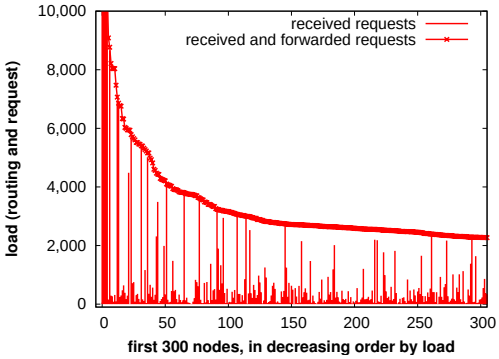


Figure 3.13: Load distribution for the 300 most loaded nodes, using a leaf set of 8 nodes (run2).

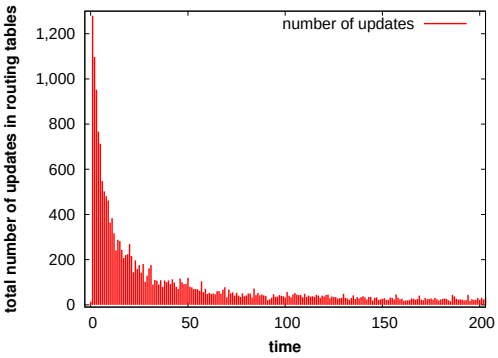


Figure 3.14: Evolution of the number of updates over time (100 requests per unit of time) in the first 200 time units (run2).

The nodes at the left hand side of the graphs are the most loaded ones. Comparing the two graphs, Figure 3.11 exhibits more nodes with a high load mostly induced by the forwarded requests. In Figure 3.12, fewer nodes have a high load, which mainly results from the received requests. The most loaded nodes are now the nodes with the highest number of received requests; the next most loaded nodes are their direct neighbors. The less loaded nodes at the right hand side of the graph (see Figure 3.11) are now more loaded, which results in a more balanced overall load tending towards a constant (see Figure 3.12).

Until now, we considered a leaf set of 4 nodes. With a larger leaf set of size 8, the results are even better, as the routing load is shared by more nodes in the vicinity of a

Table 3.1: Statistics of routing table reorganization (RTR).

Exp	Leaf	Avg	Var
run0: no update	4	2,353	7,161
run 1: update	4	2,535	2,526
run 2: update	4	2,585	2,167
run 3: no update	4	2,648	2,466
run 0: no update	8	2,253	7,103
run 1: update	8	2,319	2,394
run 2: update	8	2,350	1,966
run 3: no update	8	2,383	2,152

popular node. These results after two dynamic runs are shown in Figure 3.13.

Figure 3.14 shows the rate of updates to the routing tables in the second dynamic run (run2): the rate of updates is high in the beginning, but quickly stabilizes at a small value.

Table 3.1 contains some statistics (load average and variance) from two experiments. The first experiment (run0) has no load balancing solution. For the second experiment we show the statistics after each of the three runs. Both types of experiments are done for a leaf set containing 4 and 8 nodes. The statistical analysis shows that the variance of the system load decreases from $7,161$ for the results shown in Figure 3.9 (run0), to $2,167$ for the results shown in Figure 3.10 (run2). This confirms that the load extremes are getting closer. The load averages slightly increases from $2,353$ to $2,585$, because changing the routing tables in the destination node's closest area might increase in some cases the path length; it remains, however, in $O(log_{2^b} N)$, where N is the number of the different nodes in the system.

Our algorithm for dynamically updating the routing tables of the nodes in the system shifts the load from the most loaded nodes to less loaded nodes, by having the less loaded nodes forward most of the traffic instead. This way, the highly loaded nodes will get rid of the traffic that they have to forward, thus become less loaded. The solution does not deal with the distribution of the keys. This problem has already been well studied and can be addressed by using virtual servers [51]. Our routing table reorganization cannot decrease the load below the number of requests addressed to a node. Thus, we still have a Zipf-like distribution, but with much lower intensity.

In the next subsection we present the results for the routing table reorganization strategy complemented by caching, in order to reduce the load due to received requests (observed at the left hand side of the graphs).

3.5.2 Routing Table Reorganization and Caching (RTR&C)

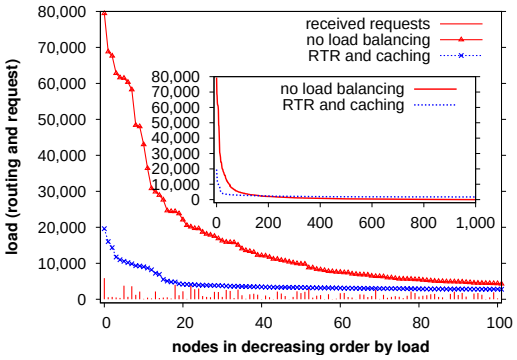


Figure 3.15: Load balancing using dynamic routing table updates and caching (*RTR&C* run1).

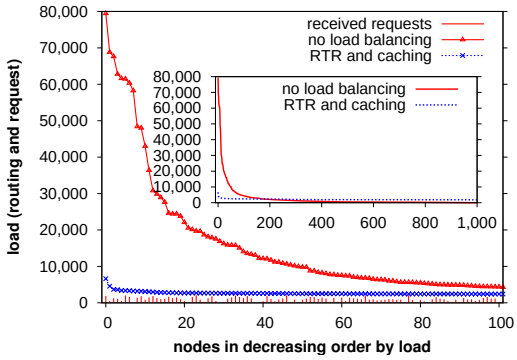


Figure 3.16: Load balancing using dynamic routing table updates and caching (*RTR&C* run2).

In our experiments, we used a cache with storage size $C = 3$ (i.e., each node has a cache for 3 replicas) and a threshold of $T = 500$ requests. Note that a caching request is issued each time the threshold is reached only in case the node is loaded. The results using both solutions (routing table reorganization and caching) are shown in Figure 3.15 and 3.16, after, respectively, run1 and run2. The experiments were done under the same conditions as before, which allow us to distinguish the benefits of using caching as a complementary solution. Comparing Figure 3.10 with Figure 3.16, we note the improvement in load balancing for the most loaded nodes (left-hand side of the graphs), where the load dramatically decreases; the load for the nodes at the right-hand side of the graph slightly increases, which demonstrates that nodes share the load more evenly.

A potential source of overhead resides in the additional messages sent for caching. For the results presented in Figure 3.16, there are 243 extra messages. This is negligible if we take into consideration the number of requests issued $(500,000)$.

The two principal variables in the system are the size of the cache C and the value of the threshold T. Table 3.2 presents some statistics of the results obtained while running the

Exp	Cache	Tshd	Msg	Avg	Var
run0	-	-	-	2,403	7,243
run1					
RTR	-	-	-	2,505	2,336
RTR&C	1	500	253	2,248	1,285
RTR&C	3	500	274	2,214	1,231
RTR&C	1	1,000	123	2,322	1,373
RTR&C	3	1,000	123	2,308	1,312
C	3	500	639	1,870	6,093
run2					
RTR	-	-	-	2,563	2,056
RTR&C	1	500	261	2,099	369
RTR&C	3	500	243	2,059	304
RTR&C	1	1,000	151	2,196	465
RTR&C	3	1,000	134	2,167	380
C	3	500	904	1,657	5,871

Table 3.2: Statistics of routing table reorganization and caching (RTR&C).

experiment with different values for the cache size and the threshold in the same system. The statistics are for the three experiments: run0, run1 and run2. Besides the load average and variance, we also show the number of messages necessary for the caching requests.

We observe that the variance of the system load decreases from $7,243$ to $2,056$ when using the *RTR* strategy (Figure 3.10). As presented before, the path length slightly increases but still in the order of $O\ (log_{2^b} N)$. The variance decreases even more to 304 when using the *RTR&C* solution (Figure 3.16). The load average also decreases, the path becoming shorter in this case.

A smaller value of the threshold means a higher frequency of caching requests, and consequently more messages. However, there is no notable improvement.

The cache does not need a large storage capacity to be effective. We obtained the same results when using $C = 3$ and $C = 100$. There is a small improvement when using $C = 3$ over $C = 1$ because the most popular objects can remain permanently in the cache.

For comparison purposes, we also ran an experiment using just caching (*C*), with no routing table update strategy. The results show that there is no significant improvement (the variance is still high, $5,871$ after run2). This means that caching alone is no satisfactory

solution.

As we have mentioned, in these experiments we used a Zipf distribution with parameter $\alpha = 1$ for the request workload. The results using other values for α are shown in the next subsection.

3.5.3 Zipf-like Requests with Different Parameter

	$\alpha = 0.5$			$\alpha = 2.0$		
Exp	**Msg**	**Avg**	**Var**	**Msg**	**Avg**	**Var**
run0	-	2,392	6,639	-	2,321	16,568
run1						
RTR	-	2,339	708	-	2,625	13,185
RTR&C	110	2,336	684	546	990	2,215
run2						
RTR	-	2,336	96	-	2,683	11,661
RTR&C	252	2,318	74	328	719	574

Table 3.3: Statistics of RTR&C using $\alpha = 0.5$ and $\alpha = 2.0$ Zipf parameter.

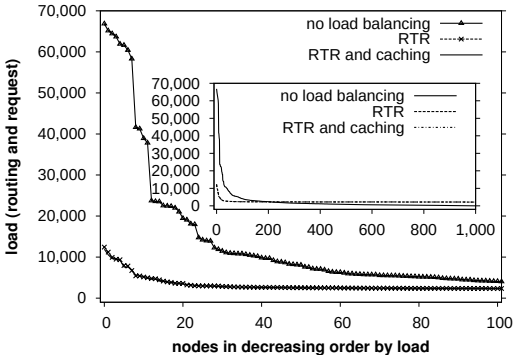


Figure 3.17: *RTR* and *RTR&C* using Zipf's $\alpha = 0.5$ (run1).

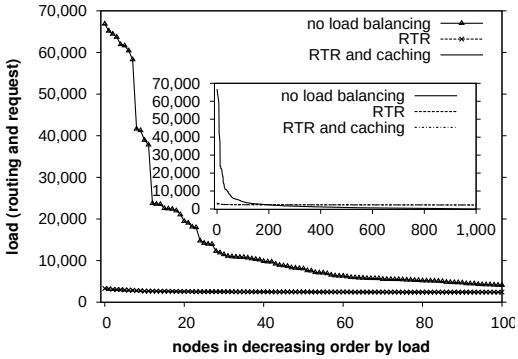


Figure 3.18: *RTR* and *RTR&C* using Zipf's $\alpha = 0.5$ (run2).

Based on [68] and [19], we performed some experiments varying the value α in order to evaluate the efficiency of our approaches. The caching storage size is set to $C = 3$ and the threshold to $T = 500$ requests.

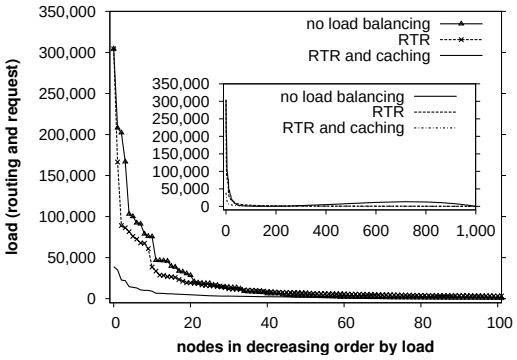


Figure 3.19: *RTR* and *RTR&C* using Zipf's $\alpha = 2.0$ (run1).

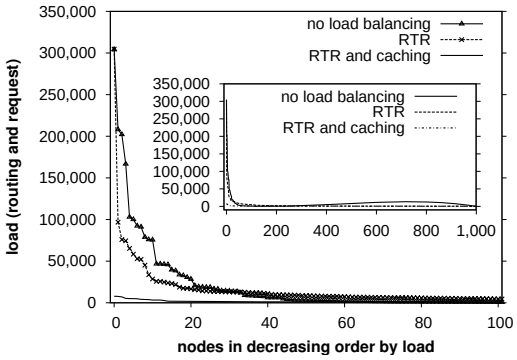


Figure 3.20: *RTR* and *RTR&C* using Zipf's $\alpha = 2.0$ (run2).

Table 3.3 presents the statistics for $\alpha = 0.5$ and $\alpha = 2.0$. Comparing the results with the Table 3.2, we observe a high difference between the variances when varying the α, this is due to the fact that α determines the degree of the popularity, i.e., a small value of α tends to a more uniform request distribution and a higher value to a more skewed distribution. Figures 3.17 and 3.18 show the results for $\alpha = 0.5$. We observe that after two dynamic runs (i.e., run2) our routing load balancing solution is satisfactory to balance the load, being unnecessary to use the complementary solution (caching). This is because the load is concentrated mainly in the forwarding traffic. As shown in the graphs, the results using the *RTR* solution and the results using the *RTR&C* solution tend to overlap. On the other hand, when we use $\alpha = 2.0$ (see Figures 3.19 and 3.20), the number of received requests for the most popular objects is very high compared to the other objects; the problem is thus only partially solved by the routing load balancing strategy and caching becomes necessary.

3.6 Related Work

Several strategies have been proposed to improve the load balancing in peer-to-peer distributed hash tables. We classify these load balancing schemes in three categories: fair partition, nodes and objects reassignment, caching and replication.

3.6.1 Fair Partition

Kenthapadi et al. [67] propose a load balancing solution where each joining node can choose its position in the hash space by learning the positions of a few existing nodes. The joining node will be placed at the mid-point of the largest partition from n randomly chosen positions in order to minimize the variation of the partitions owned by the nodes in the system. However, there is a tradeoff between the ratio of the largest and the smallest partition and the amount of knowledge of existing node positions available. If the joining node knows about all available positions, then it can join in the way that $ratio \leq 2$. But, if the node has no knowledge then it will be assigned in a random position and $ratio = O(n\ log\ n)$.

The local and global neighboring information is also used in [47] to balance the object space. In order to ensure that the ratio is small, objects may have to be moved from one node to another as the ratio grows or shrinks.

In the same context, [18] also tries to minimize the ratio, making the lengths of all partitions even. However, even if the distribution of nodes and objects are uniform and all the nodes have almost the same length of partition, the churn may degrade this distribution. So, both approaches proposed in [111] and [75] take also into consideration the ratio after nodes join and leave the system.

In the same direction, Giakkoupis et al. [48] propose a scheme for evenly partitioning the object space under join and departure of nodes. The strategy is based on the ratio of the maximum weighted size of any node partition to the minimum weighted size partition.

Finally, Aberer et al. [1] propose two load balancing strategies: storage load balancing and object replication. For the storage load balancing, the object space is recursively partitioned during the object assignment. In addition, this mechanism also addresses dynamic changes providing updates in the partitioning. The replication algorithm relies on each node obtaining an approximate local view of the system and making a decision to replicate an overloaded object space.

All these approaches focus on balancing the nodes and objects through even partitioning the hash space. Consider a skew distribution of objects popularity, even if the nodes store even number of objects, nodes holding the popular objects will become overloaded with the received requests. Therefore, these solutions have applicability in systems where the objects have uniform popularity, i.e., all the objects are requested with the same probability.

3.6.2 Node and Object Reassignment

Single ID per Node. In order to balance the address space, some DHTs use the concept of virtual servers, first proposed in [39], where each node simulates several other nodes. However, this approach requires more storage (proportional to the number of virtual servers) and network bandwidth for maintenance (every virtual server must frequently ping its neighbors).

In this way, Karger et al. [65] propose to balance the address space by activating only one of the $O(\log N)$ virtual servers at a given time. Thus, the node will occasionally check its inactive virtual nodes, and may migrate to one of them if the load in the system has changed. Since the reassignment of IDs is done by join and leave operations, this solution increases the churn of the system.

In addition, they propose to cope with uneven distribution of objects among the nodes. Normally, this problem arises for certain database applications that cannot apply a randomized distribution of the objects. Thus, even if the address space is uniformly partitioned among the nodes, an uneven distribution of the objects may lead to the whole load being placed to one machine. To solve this problem, they propose to move the nodes to arbitrary addresses minimizing the cost of load balance maintenance. Similar to that, a reorganization of the objects distribution is proposed in [127], moving the objects from one node to another based on the node's load and capacity.

Byers et al. [20] propose to use the *power of two choices* as an alternative to virtual servers. Each object is hashed onto $d \geq 2$ different IDs and placed in the least loaded node. The algorithm provides a partial traffic balancing by having each node request at random one of the d possible nodes, which all maintain redirection pointers. Since these nodes do not hold the objects, additional messages are issued, increasing the path length.

The *k-choices* load balancing algorithm proposed in [70] presents a similar approach. At joining phase, the node selects an ID from a set of k verifiable IDs in a way to minimize the discrepancies between capacity and load for itself and the nodes that will be affected by its join. Contrary to the *power of two choices*, this solution uses only one hash function with different parameters. Also, each node may change its position in the identifier space according to changes in the system (new join and departures of nodes).

All these approaches focus on objects and nodes distribution imbalance ignoring the popularity of the objects. Therefore, even if they can avoid more than one popular object to be stored in the same node, the request distribution at the destination remains skewed as

a consequence of the popularity of some objects in the overlay.

Multiple IDs per Node. Three different schemes for load balancing are proposed in [66]. All the three strategies achieve load balancing through transferring virtual servers from heavily loaded to lightly loaded nodes. The simplest one is the one-to-one scheme, where a light node picks at random an ID and verifies if the node responsible for this ID is heavy, if so, it initiates the transfer of the virtual server. The second scheme, one-to-many, permits to a heavy node considers more than one light node at a time. This scheme is implemented by maintaining directories that stores the load information about the light nodes. Therefore, it requires more storage (the load information is stored in the directories of each node) and bandwidth (extra messages to inform the load of the nodes). Finally, the many-to-many scheme matches many heavy nodes to many light nodes. This scheme uses the concept of a global pool of virtual server, an intermediate step in moving a virtual server from a heavy node to a light node. Depending on the load distribution, the pool may become a bottleneck during the transfer.

Godfrey et al. [51] complement this idea taking into consideration also the dynamism and the heterogeneity of the system. The proposed algorithm combines elements of the many-to-many scheme and one-to-many scheme proposed in [66]. The approach keeps the idea of directories to store load information and capacity of the nodes, and periodically schedule reassignments of virtual servers to achieve a better balance. This solution suffers from additional network traffic and temporary storage for virtual servers to be reloaded.

All these solutions do not take into consideration the heterogeneity of node's capacity. Instead of providing a uniform partition of the address space, the approach proposed in [52] allocates the partition to a certain node proportional to its capacity. Thus, the partitioning takes into consideration the load balance, where the load of each node is proportional to its capacity; the load movement; and normalized degree to reduce the overhead of the traffic required to keep update the routing table entries.

Zhu et al. [122, 123] focus is not only to ensure fair load distribution over nodes proportional to their capacity, but also to minimize the load balancing cost by transferring virtual servers between heavily loaded and lightly loaded nodes in a proximity-aware fashion. Taking into consideration the network proximity, it is possible to reduce the bandwidth consumption dedicated to the load movement and avoid transferring loads across high-latency wide-area links. This solution improves the communication cost at the expense of a more complex structure (*Distributed k-ary tree*) that must be maintained.

Again, these solutions ignore object popularity, although, they also can be used to balance virtual servers to ensure that no node holds more than one popular object.

3.6.3 Caching and Replication

Solutions addressing the uneven popularity of the objects are based on replication and caching. Lv et al. [73] propose three replication strategies. With the "owner replication" the requesting node keeps a copy. The number of replicas increases, thus proportionally to the number of requests to the object. In [80] and [53], a threshold is used to minimize the number of replicas. These strategies work well only when a node requests the same object many times. The second strategy, "random replication", creates copies on randomly chosen nodes along the lookup path. Simulation results [73] have demonstrated that the third strategy, "path replication", which replicates objects on all nodes along the lookup path, performs best. As an example, [39] proposes DHash replication of k successors and caching on the lookup path. The caching is initiated by the requesting node, which sends a copy of the object to each of the nodes in the lookup path. Future versions send a copy just to the nodes near to the destination.

Yamamoto et al. [116] propose the path random replication and path adaptive replication. In the first method the object is replicated at each node in the path with a certain probability that is the same for all the nodes in the system. This method can lead to imbalance since high degree nodes are frequently located in the path. The adaptive method determines the probability of replication according to a predetermined replication ratio and storage capacity.

Ramasubramanian et al. [88] strategy replicates objects based on the Zipf parameter α in order to minimize the resource consumption: storage and bandwidth. This solution requires several message exchanges to decide the replication level for each object.

Swart [104] proposes to use a second hash function to obtain a subset of $r + 1$ virtual servers and place the object on the r nodes with the lowest load. Since DHTs exhibit path convergence [56], this solution is less adapted to the popularity problem than path replication. This drawback has been already identified in [30].

Clearly, caching and replication can reduce the request traffic of the nodes that hold popular objects. However, DHTs inflexibility in terms of data placement limits the applicability of these techniques. For this reason, these mechanisms may be used as a complementary solution once they do not completely address the problem of routing overhead under

skewed request distribution in DHTs.

3.7 Conclusion

In this chapter, we presented our approach to balancing the traffic load in peer-to-peer systems. For the routing and request load balancing, we proposed, respectively, routing table reorganizations and adaptive caching based on the popularity of the objects. Our routing table reorganization mechanism primarily decreases the routing load for the most loaded nodes in the system (by charging the least loaded nodes). However, this technique can not decrease the load below the number of requests addressed to a node. Thus, we still have a Zipf-like distribution but with much lower intensity. For that reason, we proposed caching as a complementary solution in order to decrease the request load for the nodes holding the most popular objects, thus ideally balancing the global routing load in the system.

Our solution requires neither changes to the topology, nor to the association rules (placement) of the objects to the nodes. Only caching requires some extra messages to be exchanged. Results from experimental evaluation demonstrate a more balanced traffic and, consequently, improved scalability and performance.

Obviously, distributing the traffic among the nodes is not the only solution to reduce the traffic load of the nodes in the system. Moreover, for certain applications, such as range queries or publish/subscribe, many nodes are addressed during the request generating high amounts of traffic. Thus, balancing the routing traffic among the nodes in the system will not efficiently reduce the stress of individual nodes. In this way, another solution is to filter the traffic avoiding unnecessary messages to be routed in the network. In the following chapters, we present our filtering approach for efficient selective dissemination of information.

Part II

Content-based Publish/Subscribe Overlays

CHAPTER

4 Background

4.1 Introduction

The Internet has considerably transformed the constraints of distributed systems that have become more and more heterogeneous, asynchronous and dynamic. The amount of information available in the network has increased and is completely distributed in the network. Users want to be frequently updated with some information that they are interested in. Therefore, they must access the information in the Web often; otherwise they may miss important information. In order to address these user requirements, the enormous volume of information can be filtered in order to deliver only relevant information to the users. Thus, one of the biggest challenges of large-scale distributed systems is to efficiently disseminate relevant information to the users.

Publish/subscribe is a communication scheme that has received increasing attention because of its loosely coupled interaction. In this paradigm, subscribers register their interests in specific events and are asynchronously notified of any event, generated by publishers, matching their interests. The matching procedure is performed by the notification service, which is also responsible for the event delivery.

Early publish/subscribe systems (e.g., TIB/Rendezvous [82] and Elvin [95]) focused on exploiting native LAN broadcast in order to reduce the network traffic. Thereafter, with the increase of the number of subscribers, proposals were made targeting the deployment over WANs (e.g., SIENA [21] and Gryphon [13]). In these systems, the message exchanges were done using TCP/IP, which provides basic point-to-point connectivity [9]. However, these approaches lack of scalability, i.e., they reside in a fixed infrastructure, which becomes a limitation for dynamic subscription populations. On the other hand, peer-to-peer paradigm

is appropriate to build large-scale distributed applications. Peer-to-peer systems are completely decentralized, self-organizing, and fault-tolerant allowing the system to reorganize the network when a node joins or leaves. In order to achieve scalability in publish/subscribe systems, these systems ca be built on top of a fully decentralized peer-to-peer overlay (e.g., SCRIBE [23] and Bayeaux [126]).

In the rest of this chapter we will present and discuss publish/subscribe systems focusing on content-based routing approaches. In Section 4.2 includes an overview of the publish/subscribe paradigm and the challenge to implement a routing protocol for efficient event dissemination. Subsequently, the system model that we take into consideration in this work is introduced in Section 4.3. After that, we revisit the R-tree characteristics (Section 4.4), upon which we base our content-routing model presented in Chapters 5 and 6. Finally, we present the summary of the chapter in Section 4.5.

4.2 Publish/Subscribe Systems

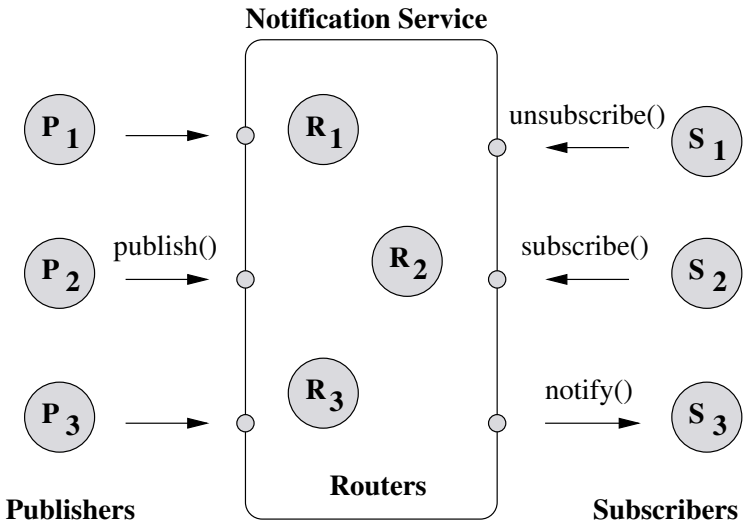


Figure 4.1: High level overview of publish/subscribe systems.

Publish/subscribe is an appealing communication primitive for large scale dynamic networks due to the loosely coupled interaction between publishers and subscribers. It is likely that an efficient, asynchronous and transparent communication system will lead to other

new large scale applications. Many applications such as stock exchange, network management systems, RSS feed monitoring, and inventory control can directly benefit from this paradigm.

In publish/subscribe systems, publishers produce *events* and subscribers express their interests through *subscriptions*; any *event* matching the subscription is delivered to the corresponding subscriber. The matching procedure is performed by the notification service, which is also responsible for the event delivery. The notification service provides an interface between publishers and subscribers, where subscribers may subscribe or unsubscribe for certain events and publishers may publish events, which are routed and delivered to the interested subscribers (as shown in Figure 4.1). Publish/subscribe systems provide decoupling between publishers and subscribers with respect to three different aspects [45]:

Figure 4.2: Space decoupling.

- *Space decoupling*: publishers and subscribers are unaware of each other. Publishers only publish the events through the notification service interface without having any information about the subscriber identities. Similarly, subscribers receive the events from the notification service without having any information about the publishers (see Figure 4.2).

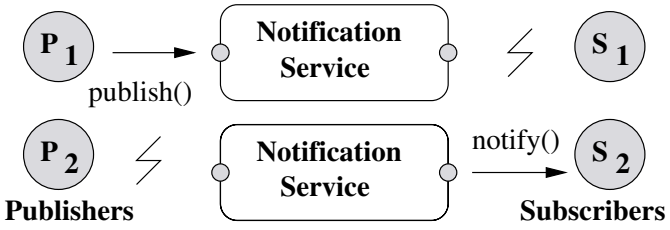


Figure 4.3: Time decoupling.

- *Time decoupling*: publishers and subscribers do not need to be active both at interaction time. Publishers might publish events by the notification service while interested subscribers are disconnected. When the subscriber connects, it will be notified for the events that were published during the period that it was disconnected (see Figure 4.3).

Figure 4.4: Synchronization decoupling.

- *Synchronization decoupling*: the event notification is asynchronous. Subscribers are not blocked waiting for the event. Subscribers may be notified at any time while performing other concurrent activities. In the same way, publishers are not blocked during and after publishing an event (see Figure 4.4).

The decoupling characteristic of publish/subscribe systems increases the scalability as it minimizes the dependencies between the parties, avoiding to deal with aspects such as synchronization and direct communication between publishers to subscribers.

4.2.1 Subscription Models

Subscribers express their interests by the mean of subscriptions. A subscription S_i is a filter over a set of events and is expressed as a set of constraints. Thus, an event E_i matches a subscription S_i if all the constraints are satisfied by the event. Different models exist for specifying subscriptions, which differ in the expressiveness to define the subscriptions. There is a tradeoff between the expressiveness given by the subscription language and the scalability of the system. In this subsection, we point out the two main models that differ in the granularity of the subscription specification: *topic-based* and *content-based* [45].

Topic-based model. In the *topic-based* model, the event space is partitioned into disjoints topic zones. These systems are similar to group communication where clients subscribe to

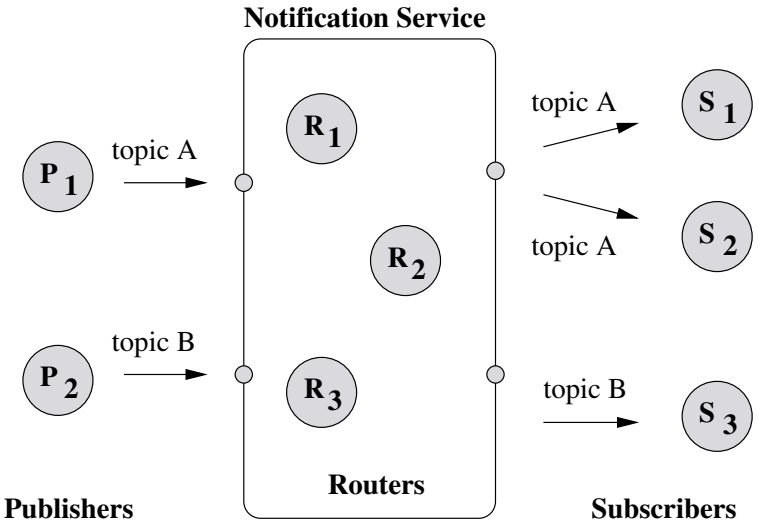


Figure 4.5: Example of topic-based publish/subscribe.

individual topics (or subjects) and the events published on a specific topic are forwarded to all subscribers participating in this topic. Figure 4.5 illustrates an example of this interaction. Subscribers S_1 and S_2 subscribe for topic A while subscriber S_3 for topic B. When the publisher P_1 publishes an event on topic A, the notification service has to deliver the event to the subscribers S_1 and S_2. In the same way, the publisher P_2 publishes on topic B in the notification service, which notifies the subscriber S_3. Many implementations of publish/subscribe systems rely on this model, such as TIB/RV [82], Scribe [23], Bayeux [126] and CORBA Notification Service [54].

Topic abstractions are easily deployed since they depend only on unique identifiers. Each topic corresponds to a logical channel connecting publishers to subscribers. Therefore, it is not necessary for the routers to perform a complex matching for each event, thus facilitating the notification process.

These systems can be easily and efficiently implemented using IP multicast. Topics are considered as multicast groups and the notification process is reduced to a lookup in the routing table for the multicast address associated to a specific topic. The main drawback of this approach is that it offers a poor expressiveness since users are limited to unique identifiers to specify the topic's name. Depending on the granularity of the subscription specification, an event may fit different topics. A subscriber that subscribes for two topics may thus receive duplicated messages if the event satisfies multiple topics (e.g. topic "Wild

Life" includes topics "Ocean Animals" and "Jungle Birds").

In order to improve the expressiveness, it is possible to define topic hierarchies, i.e., the topics may be organized according to a containment relationship. Accordingly, a subscription to a topic at level l in the hierarchy implicitly includes the set of subscriptions to all subtopics. This approach has been used by TIB/RV [82].

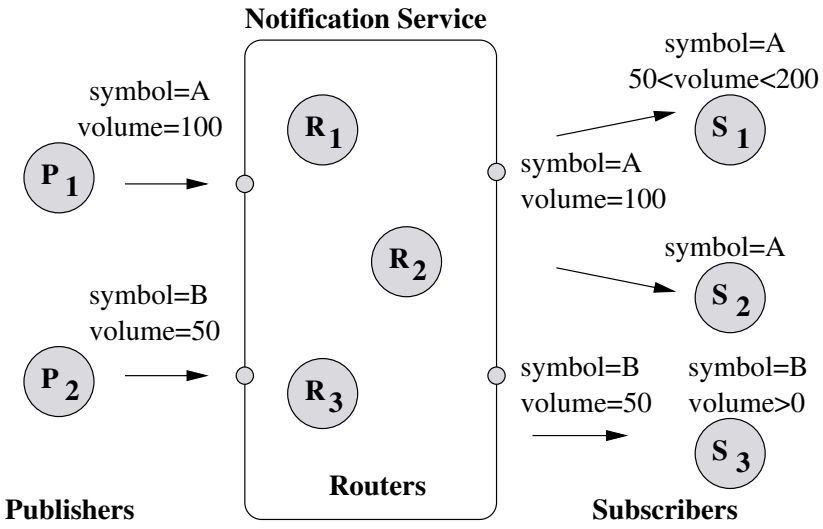


Figure 4.6: Example of content-based publish/subscribe.

Content-based model. The *content-based* model provides a finer granularity, where the subscriptions are defined according to the properties of the metadata that represents the events content. Such conditions are usually specified as a set of n attribute-value pairs, where each attribute a_i consists of a name and a value v_i with a comparison operator. The attributes may be numerical, string or other and the allowed values can be specified as single values or as range of values. Most of subscription languages support all common operators ($\leq$, $<$, $\geq$, $>$, $\neq$, $=$, etc.). Furthermore, an event E_i matches a subscription S_i if and only if all subscription's attributes are satisfied. An example of the content-based model is shown in Figure 4.6. The publisher P_1 publishes the event *symbol = 'A' and volume = '100'*, which satisfies both subscriptions S_1 and S_2. In the same way, the event *symbol = 'B' and volume = '50'*, published by P_2, is notified to S_3 since the event falls in the ranges of the subscription. A generic model to specify the subscriptions is presented

in [77]. Examples of content-based systems are SIENA [21], Gryphon [13], JEDI [38], Elvin [96] and XNet [26].

Although this model is more expressive and flexible, it adds significant complexity to the matching and routing procedure, requiring more sophisticated protocols. As a consequence, it achieves higher expressiveness at the expense of higher computational resource consumption at the routers for the matching procedure [44]. A more complex matching procedure can lead to a lower network consumption of resources as only relevant events are delivered to the subscribers.

4.2.2 Overlay Infrastructure

Most of publish/subscribe systems are built on top of overlay networks. Each node in the overlay can have at least one from the following roles: subscriber registering a subscription, publisher publishing events and router filtering and disseminating the event. The organization of the nodes in the overlay impacts on the efficiency of the notification process. Different infrastructures have been proposed. The earliest ones rely on a centralized architecture, i.e., the notification service is composed of one node (or server) acting as a router thus responsible for the event matching and dissemination. This approach is the simplest one and has the advantage of facilitating the matching procedure. However, since it is limited by the computer resources (processing power) and network bandwidth, it does not scale with the number of subscriptions/publications. In addition, the approach lacks fault-tolerance, i.e., if the server fails, the events will not be delivered to the subscribers.

In order to improve the scalability, decentralized approaches have been proposed. Here, we concentrate on the main two distributed approaches, broker-based and peer-to-peer based overlays.

Broker-based overlay. The event filtering and dissemination is distributed among a set of brokers that communicate with each other. A broker is connected to some other brokers and may have a link to some publishers or/and subscribers. This approach achieves a better scalability compared to the centralized approaches as the number of neighbors for each broker remains reasonable while the size of the system grows. Examples of broker-based overlays are SIENA [21], Gryphon [13], JEDI [38] and XNet [26].

The topology in which the brokers are organized may be flat or hierarchical. In a flat

topology, the brokers are connected to other brokers with no restriction. In a hierarchical topology, the brokers are organized in a spanning tree structure where, normally, the subscribers are connected to the leaves and publishers to the root. The main advantage of this topology is that it simplifies the event and subscription propagation: event dissemination in a top-down fashion from the root to the leaves and subscription advertisement in a bottom-up fashion from the leaves to the root.

Each broker should store and manage all the subscriptions in the system for the event dissemination. This results in large routing tables and complex subscription management. There are some approaches that try to minimize the size of routing tables using subscription aggregation or containment relationship [21, 24].

The major limitation of this type of overlay is that the overlay resides in a fixed infrastructure to filter and route the events. While subscriptions are dynamic, the event routing structure remains rather static. In case of modifications on the overlay, it is necessary to reconfigure the whole structure. Moreover, a broker-based approach requires complex filtering algorithms (e.g., [5, 25, 40]) to match each incoming message against every known subscription.

Peer-to-peer based overlay. A peer-to-peer overlay network is a logical network, built on top of the physical network, which exploits the resources at the edges of the network. In contrast to classical client-server architectures, the peers have symmetric roles, i.e., each node can be a server or a client at the same time. Hence, the peers can communicate directly without passing through intermediary entities in order to share resources (e.g., content, CPU, memory). Peer-to-peer overlays are composed of a large number of distributed, heterogeneous, autonomous, and highly dynamic nodes (e.g., participants may join and leave the system at any time).

The main motivation for using peer-to-peer overlays for event dissemination is to build scalable, self-organizing and fault-tolerant publish/subscribe systems. Several approaches have been proposed to build publish/subscribe on top of peer-to-peer systems. Examples of systems are Scribe [23] and Bayeux [126] that implement topic-based systems on top of Pastry [92] and Tapestry [120], respectively. Examples of content-based systems are Ferry [124] on top of Chord [103] and Meghdoot [58] based on CAN [89].

There are different strategies in order to organize the nodes in the overlay. This organization has an impact on the efficiency of the event dissemination. Thus, a challenge for these systems is to organize the nodes in a structure that allows for efficient distributed

matching and event dissemination.

4.2.3 Content-based Routing Protocols

As aforementioned, in the content-based model, events are defined according to their properties and subscribers express their interests by indicating the type of content that they are interested in, using some predicate language. For each incoming event, a content-based router matches the event contents against the set of subscriptions to identify and route the message to the (sub)set of interested consumers. Efficient event dissemination in content-based publish/subscribe systems is a challenging problem. Indeed, a system that uses an inefficient content-based routing protocol does not scale to large and dynamic subscriber populations.

The simplest approach is to broadcast the event to all the forwarding nodes in the network. In this approach, the router only needs to have the information about the subscriptions of the consumers connected to it, if any. This solution is easy to be deployed and does not require any specific organization of the nodes since each router forwards the event to each of its neighbors. Obviously, this solution is not scalable and suffers from a high message overhead. On the other hand, the routers only forward the events without filtering any message. Hence, the advantage of this solution is that no extra memory is required because the nodes do not store any subscription information.

In order to cope with the message overhead, selective event dissemination algorithms have been proposed to limit the event propagation only to the nodes storing matching subscriptions and to reduce the amount of subscription information to be maintained by each node. Selective event dissemination leads to scalable content routing, in the way that an increase on the publisher/subscriber population will not have significant impact in the performance [12].

One example of selective event dissemination approach is SIENA [21], which attempts to reduce the number of subscriptions in the forwarding nodes by exploiting the containment relationship (the term covering is also commonly used in the literature) among the subscriptions. SIENA organizes the brokers in a spanning tree. The main originality is in the subscription advertisement scheme, where subscriptions are propagated among the brokers only if a subscription with a larger scope has not been propagated already.

Organizing the nodes such that only a subset of nodes stores the information of a certain subscription and only a subset of nodes participates in the event dissemination, can lead to

more scalable content-based routing. Furthermore, the routing process of an event typically involves a large number of intermediary nodes, which sometimes act as forwarders and have no interest in the event.

In order to disseminate quickly the events, subscribers can be organized based on their interests (subscriptions), i.e., subscribers with similar interests are gathered together in the overlay. As a consequence, this can save memory and bandwidth consumption [87]. Also, if the subscribers participates in the event dissemination for the events that they are interested in, the number of messages during the event dissemination can be reduced. Thus, the organization of the subscribers in the system has a strong influence on the routing accuracy, i.e., the number of false positives (a peer receives a message that it is not interested in) and false negatives (a peer does not receive a message that it is interested in) generated in the system. False positives are not critical to the subscribers since they can filter out irrelevant events. However, they affect the event dissemination performance; a high number of false positives mean a high bandwidth and memory consumption. Conversely, false negatives must be avoided as they affect the application consistency.

A straightforward approach for avoiding false positives and false negatives is to organize the subscribers in a tree structure according to containment relationships, such that the subscription of a peer contains the subscriptions of its descendants.

Indeed, if an event matches the containee, it *has* to match the container (this guarantees no false negatives); conversely, if it does not match the container, it cannot match the containee (this guarantees no false positives). Figure 4.7 illustrates the containment graph for a set of subscriptions.

A direct mapping of the containment graph to a tree structure [25] is often inadequate. First, it requires a virtual root with as many children as subscriptions that are not contained in any other subscription. Second, depending on the subscription workload, the resulting tree might be heavily unbalanced with a high variance in the degree of internal nodes. Another approach consists in building one containment tree per dimension and adding a subscription to each tree for which it specifies an attribute filter [6]. This solution tends to produce flat trees with high fan-out and generates a significant number of false positives.

A good approach tries to combine the best of two worlds by organizing the nodes in a bounded-degree height-balanced trees, while preserving the containment relationships that ensure accurate content dissemination. To that end, we propose distributed extensions of the R-tree index structures that will be presented in the next chapter. In the following

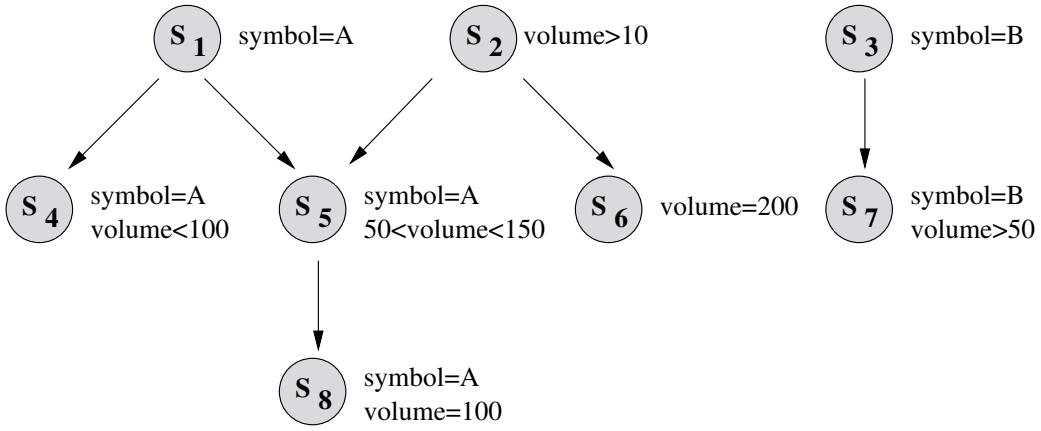


Figure 4.7: Example of a containment graph for a set of subscriptions with two attributes: *symbol* and *volume*.

sections, we present, respectively, the system model used in our studies and the overview of R-trees, which our content-based routing approaches rely on.

4.3 System Model

In our work we consider two types of subscription representations: geometrical representation, where the subscriptions are represented in an n-dimensional space; and space filling curves representation, where the subscriptions are mapped to a 1-dimensional space. Both representations have already been adopted in some publish/subscribe systems (e.g., [112, 119] for geometrical and [79, 100] for space filling curves). In the next subsections we detail both representations.

4.3.1 Spatial Filters

As most other content-based publish/subscribe systems, we assume that an event is a set of attribute-value pairs. Each attribute has a name and a numeric or string value. A subscription is a conjunction of predicates over the attribute values, i.e., $S = f_1 \wedge \ \ldots \ \wedge f_j$, where f_i is defined as a tuple $f_i = (n_i \quad op_i \quad v_i)$ with n_i the name of the attribute, op_i an operator ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$, etc.), and v_i a constant value. For example, a subscription expressed on the attributes a and b may be of the form $(v_i < a < v_j) \ \wedge \ (v_k < b < v_l)$.

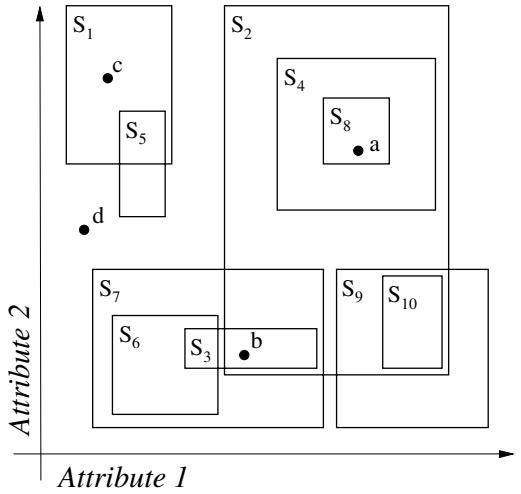


Figure 4.8: Sample subscriptions and events with two attributes. S_i are subscriptions and a, b, c and d events. Certain subscriptions are interested in the events a, b and c in contrast to the event d that does not match any subscription.

Geometrically, these complex filters define poly-space rectangles in the Euclidean space. So, a schema composed by n attributes may be represented in a Euclidean space of n dimensions. The subscriptions correspond to rectangles and events to points. Figure 4.8 shows a set of subscriptions and events defined on a 2-dimensional space with two attributes. Note that, if one attribute is undefined, then the corresponding rectangle is unbounded in the associated dimension. Also, if one attribute is composed by segments of a range value, the subscription is represented as multiple rectangles (each rectangle is represented by the coordinates of the upper left corner and the bottom right corner). In this case, we may consider subscription merging if the subscriber does not desire to store all the rectangles, instead the subscriber will store only the maximum and minimum coordinates of the poly-rectangle that encloses all the rectangles at the price of a loss of accuracy (false positives).

Publish/subscribe systems can take advantage of the property of *subscription containment* in order to reduce the number of subscriptions stored at the forwarding nodes. Reducing the number of subscriptions stored at the nodes allows a better usage of network bandwidth and memory. In addition, the nodes will have fewer subscriptions to match against the events, speeding up the event dissemination.

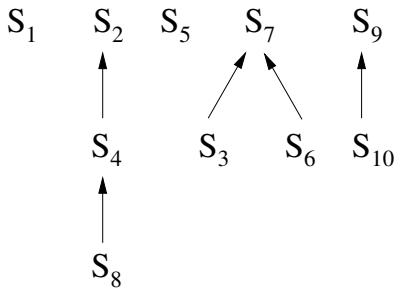


Figure 4.9: Containment graph for the subscriptions of Figure 4.8.

Containment relationship. Containment relationship among subscriptions is defined as follows: subscription S_1 contains another subscription S_2 (written $S_1 \sqsupseteq S_2$) iff any event E_i that matches S_2 also matches S_1. Conversely, we say that S_2 is contained by S_1 and we write $S_2 \sqsubseteq S_1$.

Note that the containment relationship is transitive and defines a partial order. Geometrically, subscription containment corresponds to the enclosure relationships between the poly-space rectangles. Figure 4.9 shows the containment relationships for the sample subscriptions in Figure 4.8.

4.3.2 Space Filling Curves

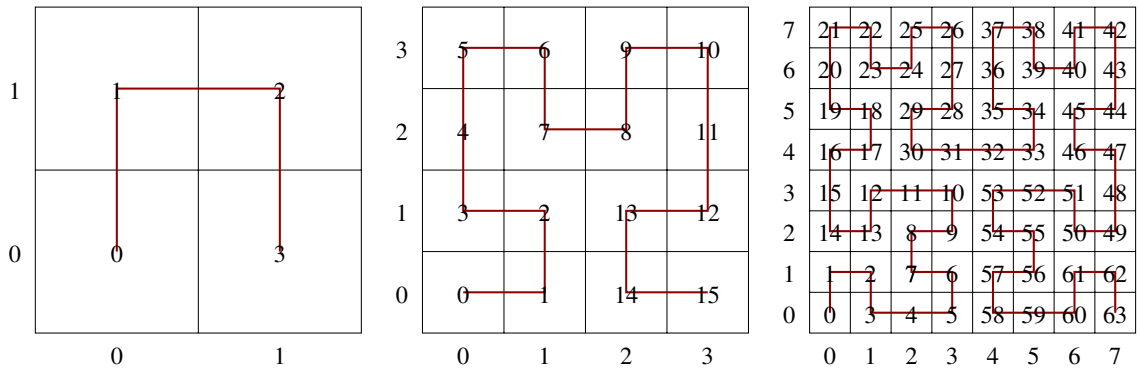


Figure 4.10: Example of Hilbert curve for, respectively, $k = 1$, $k = 2$ and $k = 3$.

Another representation approach is to map multi-dimensional subscriptions and events in one single dimension using space filling curves (SFC). One transformation approach

is the Hilbert space filling curves [60], which maps an n-dimensional space into a 1-dimensional space while preserving the spatial proximity between the subscriptions. Studies have shown that the Hilbert curve outperforms other space filling curves in preserving proximity [76].

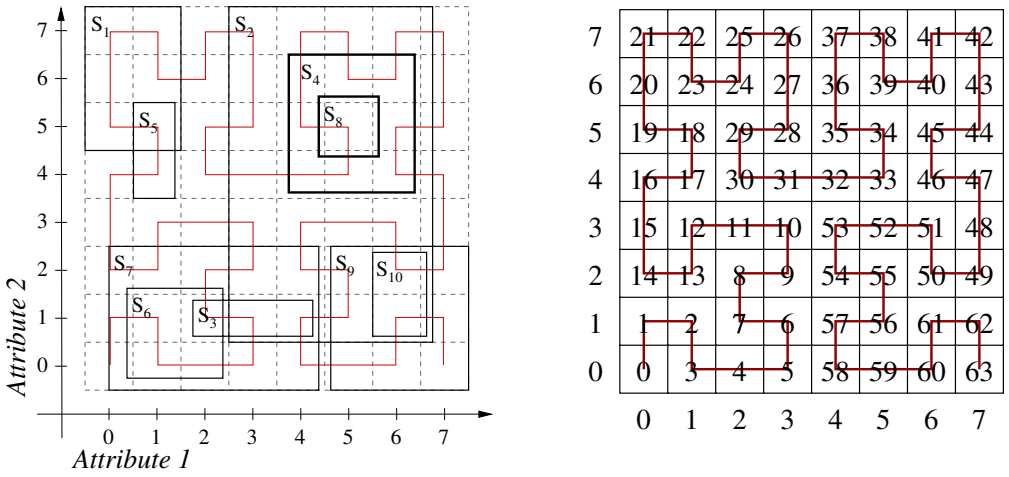


Figure 4.11: Hilbert curve approximation for the subscriptions of Figure 4.8 ($k = 3$).

The Hilbert SFC imposes a linear ordering, partitioning the Euclidean space into grid cells and assigning a single integer value to each cell, thus one dimension. Given the k^{th} order of the curve, the Euclidean space is partitioned into 2^k intervals in each of the n dimensions, generating 2^{kn} cells (where $k \geq 1$ and $n \geq 2$). The k^{th} order of an n-dimensional Hilbert curve maps $[0, 2^k - 1]^n$ coordinates into $[0, 2^{kn} - 1]$ integer values (see Figure 4.10). Extending this notation, we may assume that each cell, for each dimension, is represented by a range value instead a single value. Therefore, a set of coordinates circumscribed by the grid cell is represented by a single integer value, i.e., the k^{th} order of an n-dimensional Hilbert curve maps $[0, r2^k - 1]^n$ coordinates, where r is the same range value for each dimension, into $[0, 2^{kn} - 1]$ integer values. Using this notation, subscriptions are represented as a set of integer range values and events as a single integer value. An example of Hilbert representation from the sample subscriptions of Figure 4.8 is shown in the left hand side of Figure 4.11. In this example, considering the subscription S_4 represented as $\{(4, 4), (6, 6)\}$, using Hilbert SFC of order $k = 3$, S_4 is represented in the form $\{[32, 36], [39, 40], [45, 46]\}$. The curve approximation for $k = 3$ is shown in the right hand side of Figure 4.11.

Preserving proximity between the subscriptions in publish/subscribe systems helps to

minimize the communication cost during event dissemination as similar subscriptions may be placed near to each other in the system. In addition to that, the aforementioned containment relationship between subscriptions may be exploited in order to improve even more the filtering procedure.

Containment relationship. Containment relationship based on Hilbert SFC representation is defined as follows: subscription S_1 contains subscription S_2 or $S_1 \sqsupseteq S_2$, iff for every range of S_2, this range is covered by the ranges of S_1. Thus, if an event E_i matches S_2, it also matches S_1. In the example shown in the left hand side of Figure 4.11, S_4 is represented in the form $\{[32, 36], [39, 40], [45, 46]\}$ and S_8 in the form $\{[34, 34]\}$. As the range $[34, 34]$ from S_8 is covered by the range $[32, 36]$ from S_4, thus $S_4 \sqsupseteq S_8$.

4.4 R-tree Index Structure

R-trees [59] were first introduced as a hierarchical structure in the context of database applications. The main idea is to organize dynamically a set of multi-dimensional geometrical objects in order to quickly retrieve these spatial objects.

An R-tree is a height-balanced tree, where each node in the tree is represented by the smallest poly-space rectangle, called *minimum bounding rectangle* (MBR), enclosing all the rectangles in the subtree rooted by this node. Thus, the MBR of a leaf node is the smallest rectangle that bounds the spatial object; and the MBR of a non-leaf node corresponds to the smallest rectangle that surrounds all the MBRs of its children. In a database context, the leaves point to the database objects. Note that, depending on the spatial objects representation the MBRs may overlap or contain each other.

An R-tree is characterized by the following properties:

- Every non-leaf node has a maximum of M and at least m entries where $m \leq M/2$, except for the root.
- The minimum number of entries in the root node is two, unless it is a leaf node. In this case, it may contain zero or one entry.
- Each entry in a non-leaf node is represented by (mbr,p), where the mbr is the MBR that encloses the MBRs of its child node and p is the pointer to the child node. Each

entry in a leaf node is represented by (mbr,oid), where the mbr is the MBR that spatially encloses the object and oid is the pointer to the object.

- All the leaf nodes are at the same level.
- The height of an R-tree containing N index records is $\lceil log_m(N) \rceil - 1$, where m is the minimum number of entries.
- The maximum number of nodes is the sum of the maximum number of nodes per level, i.e., $\sum_{i=1}^{h_{max}} \lceil N/m_i \rceil = \lceil N/m_1 \rceil + \lceil N/m_2 \rceil + \ldots + 1$.
- The worst space utilization for each node, except the root, is m/M.

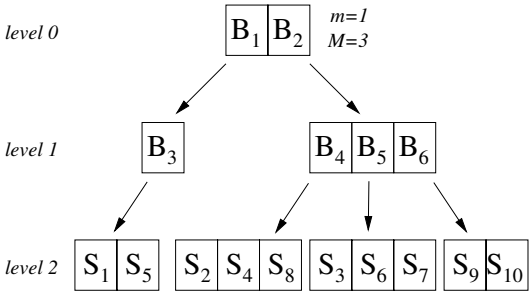


Figure 4.12: R-tree for the subscriptions of Figure 4.8.

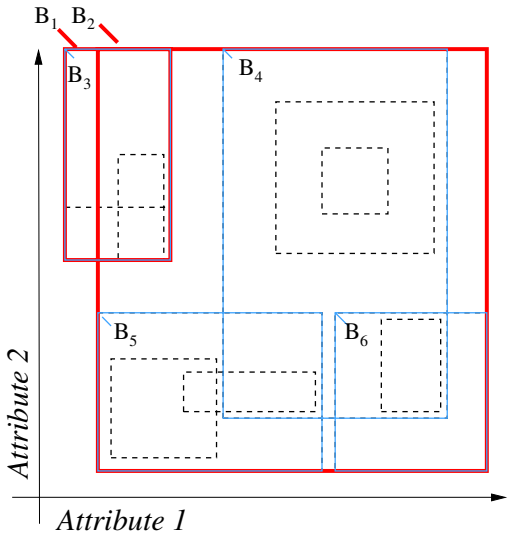


Figure 4.13: Spatial representation of the R-tree of Figure 4.12.

In an R-tree structure, the actual objects are only stored at the leaves of the tree and internal nodes only maintain MBRs. An R-tree constructed from the sample subscriptions of Figure 4.8 is shown in Figure 4.12 (for $m = 1$ and $M = 3$) and its spatial representation in Figure 4.13.[1] Note that all subscriptions are stored at the leaves and the role of internal nodes $B_1, \ldots, B_6$ is to keep track of the bounding rectangles that contain their descendants. In distributed settings, internal nodes must be managed by specific peers in the system.

[1]For simplicity, we illustrate an example with $m = 1$. However, in practice m is greater than 2.

An R-tree is completely dynamic; insertions and deletions of entries may be executed at any time and interleaved with searches. At each new insertion or deletion of an entry, the tree is locally reorganized, thus no periodic reorganization is required.

Search. The query search starts from the root and follows downwards the tree according to the MBRs. Thus, considering a query q, the search is forwarded to the child c of the node n where $n.mbr$ encloses q. Since MBRs may overlap or contain each other, depending on the tree structure, a search may visit more than one subtree. Moreover, representing spatial objects through MBRs may generate false alerts during the search procedure since most of times there is a dead space (i.e., the area between the MBRs that are not enclosed by them) between the MBR of a node and the MBRs of its children. In addition, these false alert regions may be propagated upwards the tree. Therefore, the tree organization, i.e., how the entries are grouped is very important in order to minimize these false alert regions.

Insertion. Insertions in the R-trees are also propagated from the root to an appropriated leaf node where the new entries will be accommodated. When an entry is inserted, all the nodes on the path to the root must update their MBRs; as they may be extended to fit the new entry. Since the nodes may have maximum M entries, if the leaf node is already full, the node is split into two nodes. One child node remains with its parent and the other node will have a new parent. The split is propagated upwards the tree updating the MBRs and splitting non-leaf nodes that become full. Thus, the choice of which branch the new entry is inserted and how the splitting takes place influence the total area of the MBR and as a result, the number of false alerts. In the following subsections, we present different variants of R-trees that try to improve the tree structure through different insertion and splitting strategies.

Deletion. When an entry is deleted from a node, the R-tree must be updated in order to guarantee the minimum number of entries m. If after the deletion, the node has too few entries (less than m) then this node is eliminated and all the remaining entries are reinserted in the tree. The node elimination is propagated upwards the tree, as necessary. In addition, all the MBRs must be updated within the path to the root since it may become smaller.

4.4.1 R-trees: Linear, Quadratic and Exponential Method

In R-trees the choice of the branch in which the new entry will be accommodated is based on the coverage area of the MBR, i.e., the entry is inserted into the node whose MBR needs the smallest enlargement to include the new entry. Minimizing the coverage is important in order to minimize the dead space between MBRs, which may generate false alerts. However, upon an insertion of a new entry, if the children's set becomes contains more than M entries, the children set must split. There are three strategies, in the R-trees, for splitting an overflowing node:

- The *linear* method chooses two children from the overflowing node, such that they are as far as possible, and places each one in a separate node. For each remaining child picked randomly, it is assigned to the node whose MBR will increase the least due to the addition. This method takes linear time.

- The *quadratic* method chooses two children from the overflowing node such that the union of their MBRs would waste the most area and then place each one in a separate node. The remaining MBRs are examined, and the one whose addition maximizes the difference in coverage between the MBRs associated with each node, is added to the node whose coverage is minimized by the addition. This method takes quadratic time.

- The *exponential* method tests exhaustively all possible groupings and the best two groups are chosen with respect to the smallest MBR enlargement. This method takes exponential time.

Guttman [59] suggests that the quadratic method offers a good compromise to achieve reasonable search performance. For comparison purposes, in our work, we consider only the linear and quadratic methods. Furthermore, as will be shown, in the context of publish/subscribe the quadratic method also gives better results compared to the linear method.

4.4.2 R*-trees

R*-trees [14] were proposed in order to improve the performance of R-trees. As previously discussed, R-trees tend to minimize the coverage of the MBRs. R*-trees combine coverage, overlap and margin of each enclosing MBR to achieve a better search performance. As

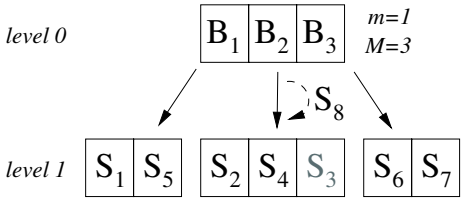


Figure 4.14: Example of splitting method in R*-trees for $M = 3$. Entry S_8 is inserted in the children set of B_2, thus the children set splits.

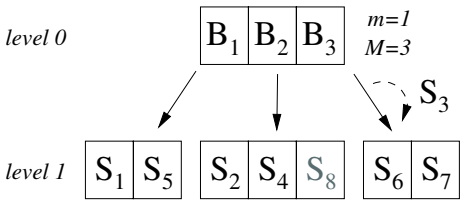


Figure 4.15: The children S_2, S_4 and S_8 remains in the children set of B_4 and S_3 is reinserted in the tree. In this case, S_3 becomes child of B_3.

aforementioned, minimizing the coverage implies minimizing the dead space. Minimizing the overlap between the MBRs avoids that a search query visits several branches of the tree. The last criterion is to minimize the margins, i.e., minimize the sum of the lengths of the MBR edges. Considering a fixed area, a square is supposed to have the smallest margin. Thus, in order to minimize the margins, the shapes of the MBRs are approximate the quadratic shape and propagated to upper levels since quadratic objects may be packed more easily. Whatever parameter is chosen, they all have an impact on the number of false alerts that may be generated during a search query. Conversely, these criteria may be contradictory. Minimizing the margins in order to have more the shapes of the MBRs closer to a quadratic shape, may increase the overlap between them. On the other hand, minimizing the coverage and overlap may change the shape of the MBRs, thus increasing their margins.

Upon the insertion of a new entry, the choice of which branch the entry will be forwarded to is based on coverage and overlap. As in R-trees, the insertion starts from the root and is forwarded to the node that needs the smallest enlargement of its MBR to include the new entry. This procedure is repeated until the last non-leaf node. In order to include the new entry in a leaf node, the R*-tree insertion algorithm examines the overlap between MBRs, i.e., the selected node is the one whose MBR enlargement leads to the smallest overlap increase between the MBRs of the leaf nodes.

In case the node cannot accommodate the new entry because it already has M entries, the node will split. However, instead of splitting immediately, this variant keeps a certain fraction of entries (that is greater than m) in the node and, the remaining entries are reinserted in the tree. As the insertion is non-deterministic, the new entries may be allocated

to a more suitable node. In spite of that, reinsertion is costly, thus only one reinsertion per level is allowed. If a reinsertion already took place and the node overflows, in this case the node will split, creating two new nodes. To determine the entries distribution among the two nodes, R*-trees evaluate a good compromise between coverage, overlap and margin of the MBRs. Figures 4.14 and 4.15 illustrates an example of the split method for a tree (where $m = 1$ and $M = 3$) built from the set of subscriptions of Figure 4.8 (the subscriptions S_9 and S_{10} were not inserted in the tree at this stage). In order to allocate the new subscription S_8, the children of node B_2, already with 3 entries, must split. The entries S_2, S_4 and S_8 are kept in its children set, and subscription S_3 is reinserted in the tree finding a better position as a child of B_3.

4.4.3 Hilbert R-trees

Hilbert R-trees [64] are another variant of R-trees, which impose an ordering on the nodes at the same level in order to improve the proximity of the children set. This variant is based on Hilbert space filling curves [60], which preserve the proximity of the spatial objects.

The entries at the non-leaf nodes store, in addition, a *Hilbert value*. The Hilbert value (hv) is computed from the center coordinate of the MBR of the node and it is only calculated for the leaf nodes. Every non-leaf node is represented by (mbr, p, ℓhv), where mbr is the MBR that encloses the MBRs of its child node, p is the pointer to its child node and ℓhv is the largest Hilbert value among the entries of its child node. Entries at the leaf-nodes are represented in the same manner as in R-trees, by (mbr, oid). Hence, each non-leaf node only keeps track of the largest Hilbert value of its subtree and it does not recalculate it. The Hilbert R-tree variant, with respective Hilbert values, built from the sample subscriptions of Figure 4.8 is presented in Figure 4.16 (for $m = 1$ and $M = 3$).

The insertion of a new entry differs significantly from the other variants as it does not try to minimize the coverage of the MBRs, but it is based only on the Hilbert values. Before starting the insertion procedure, the Hilbert value is calculated from the centroid of the MBR of the new entry. Then, this Hilbert value is used to guide to which subtree the new entry will be forwarded. At each level, the chosen subtree is the one which the node has the smallest Hilbert value that is larger than the Hilbert value of the new entry. When a leaf node is reached, the new entry is inserted in the correct order according to the Hilbert values of the children. In addition, the internal nodes are also ordered by their ℓhv. In this way, the ordering of the nodes at each level of the tree is guaranteed. Note that, when the

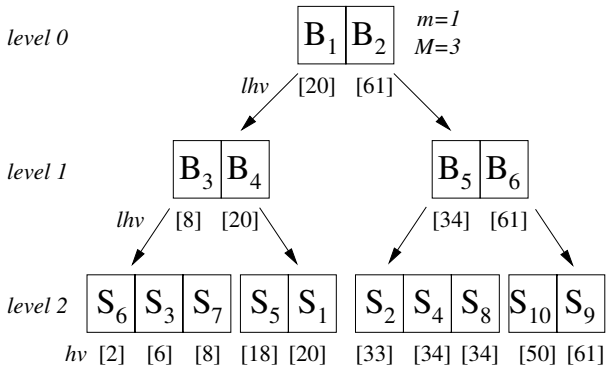


Figure 4.16: Hilbert R-tree for the subscriptions of Figure 4.8.

update is propagated upwards the tree, not only the MBR is updated but, as well, the largest Hilbert value.

When a node overflows, the node does not split immediately, but some of its entries are allocated to sibling nodes. Only if all the sibling nodes are full then the split takes place. Considering s sibling nodes that are full, the split procedure results on $s + 1$ nodes. Where the split will happen depends only on the Hilbert values. Thus, as the nodes are ordered by their Hilbert values it is only necessary to determine the split point in order to distribute evenly the entries among the $s + 1$ nodes.

In this variant, the deletion procedure also uses the concept of sibling nodes. After deleting an entry, if the node becomes underflow it takes some entries from the sibling nodes. In case all the sibling nodes underflow, they will be merged.

4.5 Conclusion

Publish/subscribe is an appealing communication primitive for large scale dynamic networks due to the loosely coupled interaction between the participants. In this paradigm, publishers produce *events* and subscribers express their interests through *subscriptions*; any *event* matching the subscription is delivered to the corresponding subscriber.

Unlike conventional routing, where packets are routed based on a set of attributes such as IP addresses and port numbers, publish/subscribe systems route their messages based on topics or on a representation of their content [45]. In *topic-based* systems, clients subscribe to individual topics and events published on a specific topic are forwarded to all clients

participating in this topic. The *content-based* systems provide a finer granularity, where subscribers specify their interests based on event contents. The topic-based approach lacks of expressiveness, but it can be implemented very efficiently. On the other hand, content-based publish/subscribe is more expressive and flexible, but it requires more sophisticated protocols.

Traditional solutions for content routing are usually based on a fixed infrastructure of reliable brokers. While subscriptions are dynamic, the event routing structure remains mostly static. This approach limits the scalability and routing accuracy with the increase and dynamism of subscription populations. Moreover, this solution introduces single point of failures and bottlenecks.

Another approach to content routing is to design it with a free of brokers infrastructure, and organize subscribers and publishers in a peer-to-peer overlay through which messages flow to interested parties. Several designs of peer-to-peer publish/subscribe systems were proposed [23, 58, 124, 126]. The main advantage of these approaches is their scalability, although most of them suffer from the loss of accuracy (apparition of false negatives or false positives). Hence, for such approaches to be efficient, the overlay must: avoid *false negatives* (a subscriber failing to receive a message it is interested in); minimize the occurrence of *false positives* (a subscriber receiving a message that it is not interested in); *self-adapt* to the dynamic nature of the systems, with peers joining, leaving, and failing; and maintaining the *overlay balanced* in order to provide a publication service time logarithmic in the size of the network.

In the next chapters, we present our Distributed R-trees and Distributed Hilbert R-trees approaches, which address the limitations of content routing in publish/subscribe systems. Our approaches extend R-trees to construct a peer-to-peer overlay for selective dissemination of information.

CHAPTER

5 Distributed R-trees

5.1 Introduction

In this chapter, we present the Distributed R-tree (DR-tree) overlays [15, 16] that are based on R-trees [59] and R*-trees [14] to construct a peer-to-peer network optimized for selective dissemination of information. Distributed R-trees are a class of content-based publish/subscribe overlays where subscribers and publishers are organized in peer-to-peer balanced structures based only on their interests.

Our overlays achieve the efficiency through: 1) organizing subscribers in a distributed and completely decentralized virtual balanced tree, based on the containment relationship of the subscriptions; 2) disseminating quickly the events in the system and minimizing the message overhead; 3) providing a zero risk of false negatives and maintaining a low level of false positives; 4) including self-stabilizing protocols [15] that guarantee consistency despite failures and rapid changes in the peer populations.

Section 5.2 presents the design of our peer-to-peer stabilizing overlays and extends them to embed publish/subscribe systems with complex spatial filters. Our overlays guarantee subscription and publication times that are logarithmic in the size of the network. They are self-organized in a balanced tree with a per node polylogarithmic memory cost.

In Section 5.3 we propose self-stabilization algorithms for our overlays. We show that the recovery time in face of joins/leaves or memory corruptions is also logarithmic in the size of the network. We also prove that our overlays remain correct in spite of transient faults, joins, and leaves, and we analytically study its resistance to churn.

Finally, the Section 5.4 concludes the chapter.

5.2 Distributed R-trees

In this section we extend the R-tree index structures to peer-to-peer content-based spatial filters. That is, subscribers self-organize in a balanced virtual tree overlay based on the semantic relations between their subscriptions. We refer to the resulting distributed structure as *DR-tree*. In order to simplify the presentation we consider that each filter is a rectangle and can be represented using coordinates in a two dimensional event space. The extension to complex filters represented with poly-space rectangles is straightforward.

5.2.1 Overlay Organization

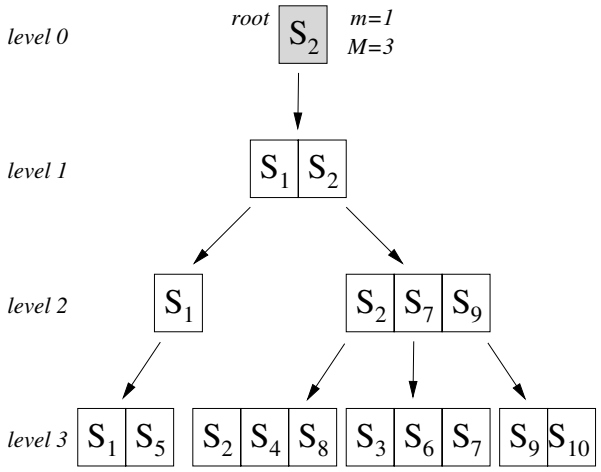


Figure 5.1: DR-tree for the subscriptions of Figure 4.8.

In our peer-to-peer content-based routing overlay, every peer or subscriber subscribes for certain interest, may or not publish events and may participate in the event routing. These subscriptions are mapped onto a logical balanced tree, called DR-tree, in the way that every peer is responsible for, at least, a leaf node of the logical tree, which stores its subscription. In case of a peer registering multiple subscriptions, the peer may choose between being responsible for multiple leaf nodes in the overlay or compute the MBR of the set of subscriptions to register as a single subscription. Depending on the nature of a peer's subscription (also called filter), it may be responsible also for internal nodes of the tree. In this case, the peer has not only the role of subscriber, but also of router participating

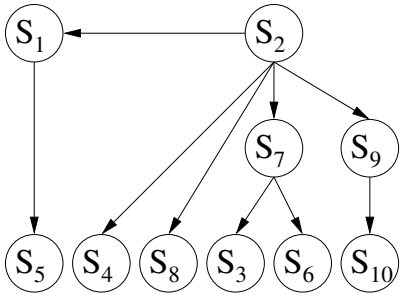


Figure 5.2: Communication links between the peers/subscribers in the system for the of subscriptions of Figure 5.1.

in the event dissemination. A subscriber responsible for an internal node of the tree filters events for all subscribers responsible for internal nodes in its subtree. In order to maintain the balanced nature of the tree, such a subscriber is responsible for an internal node at each level of its subtree. More precisely, an internal node p is recursively its own child in the subtree rooted by p. Therefore, a peer may have different children sets for each level where it is responsible for an internal node and, thus may have to maintain more than one parent link and children set.

Considering again the sample subscriptions of Figure 4.8, a possible logical organization of the subscriptions in the system is shown in Figure 5.1. Some subscriptions are both leaves and internal nodes of the DR-tree. The choice of which subscriptions are promoted as internal nodes will be discussed shortly. The DR-tree has a single virtual root (subscription S_2) that appears at all levels. The physical organization of the subscribers is shown in Figure 5.2. Each node is neighbor of its children and parent in the DR-tree.

Events flow through the tree according to the MBR value of the nodes at each level: an internal node forwards the event to each of its children whose MBR contains the event. An event produced by a node n is disseminated along the interested subtrees for which n is the root; further, it is propagated upwards the root of the DR-tree and down to every sibling subtree encountered on the path to the root. Thus, event dissemination may start from any node in the tree. Consider for instance the production of the event b (see Figure 4.8) by the peer associated to the subscription S_7 in the DR-tree of Figure 5.1. The instance of node S_7 at level 2 sends the event down to child S_3 as its subscription matches the event, and upwards to S_2. Node S_2 checks if the event is contained by the MBR of any of its children at levels 0 and 1 and, as this is not the case, does nothing. Therefore, the event is received

only by S_3, S_7, and S_2, thus producing no false positive and transmitting only 2 messages.

DR-tree structures guarantee that no false negative occurs during event dissemination, i.e., every subscriber receives the events it is interested in. However, the organization of the subscribers has a strong influence on the routing accuracy and the number of false positives in the system. It is, therefore, essential to organize the nodes carefully so as to minimize the occurrence of false positives. To this end, we organize the subscriptions in the DR-tree structure while preserving existing containment relationships. In particular, we want to preserve the following property:

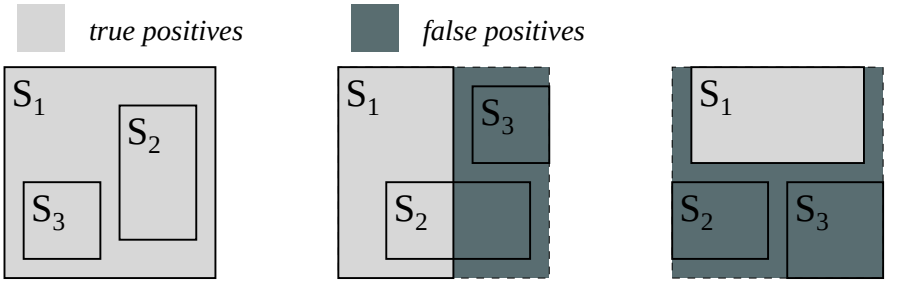
Property 5.2.1 (Weak Containment Awareness) *Given two filters S_1 and S_2 with $S_1 \sqsubseteq S_2$, then (the topmost instance of) S_1 is not an ancestor of (the topmost instance of) S_2 in the DR-tree.*

This property guarantees that a containee filter will not be a parent of a container filter, as it would degrade routing accuracy. In addition, it is desirable to implement a stronger variant of the containment awareness property:

Property 5.2.2 (Strong Containment Awareness) *Given two filters S_1 and S_2 with $S_1 \sqsubseteq S_2$, then either (the topmost instance of) S_2 is an ancestor or sibling of (the topmost instance of) S_1 in the DR-tree, or there exists S_3 such that $S_1 \sqsubseteq S_3$, $S_2 \not\sqsubseteq S_3$, $S_3 \not\sqsubseteq S_2$, and (the topmost instance of) S_3 is an ancestor or sibling of (the topmost instance of) S_1 in the DR-tree.*

This property would ensure that a containee filter is a descendant of its containers. Because of the height-balancing mechanism, it might not be possible to register a containee deep enough in the tree as child of one of its container; in that case, it can be inserted as a sibling of the container. The second clause of the property deals with the case of a filter having two container filters that do not cover each other (remember that the containment relationship is a partial order). Therefore, the containee may become a descendant of either of its container. This case is illustrated in Figure 4.8, with S_{10} being contained in both S_2 and S_9. The DR-tree of Figure 5.1 preserves the strong containment awareness property but, in the general case, the order of node insertion and removal may lead to sub-optimal configurations in which this property is occasionally violated.

In order to preserve the containment awareness properties and minimize the likeliness for false positives, we elect as root of a subtree the node whose current MBR is largest,



1: containment. **2:** intersecting MBRs. **3:** disjoint MBRs.

Figure 5.3: Principle of root election. In all cases, S_1 is the best candidate to be elected as root.

i.e., which provides most coverage over the MBR of the new root. If one filter covers all the others, then it trivially becomes the new root (case 1 in Figure 5.3): the containment awareness properties are preserved and there is no occurrence of false positives. If filters intersect or are disjoint, we elect the node with the largest MBR (cases 2 and 3 in Figure 5.3) in order to minimize the size of the area corresponding to false positives. As we shall discuss later, it might be desirable to *not* elect the node with the largest MBR when the distribution of events in not uniform.

5.2.2 Overlay Maintenance

Data Structures. Each node p in the overlay maintains constant non-corruptible data representing its subscription:

$$filter_p = S_p = ((\underline{x}_p, \underline{y}_p), (\overline{x}_p, \overline{y}_p))$$

where $\underline{x}_p$ and $\underline{y}_p$ represent the minimal abscissa/ordinate, and $\overline{x}_p$ and $\overline{y}_p$ the maximal abscissa/ordinate of the rectangle that circumscribes the filter. Additionally, each node p maintains the following variables:

- C_p^l is the children set of p at level l. This set is periodically updated upon insertion of new subscriptions or existing children leave the structure at level l.

- $mbr_p^l = ((\underline{x}_p^l, \underline{y}_p^l), (\overline{x}_p^l, \overline{y}_p^l))$ represents the minimum bounding rectangle that includes

all the MBRs of all children at level l, and is computed as:

$$mbr_p^l = (((\min_{q \in \mathcal{C}_p^l}(\underline{x}_q^{l+1}), \min_{q \in \mathcal{C}_p^l}(\underline{y}_q^{l+1})), (\max_{q \in \mathcal{C}_p^l}(\overline{x}_q^{l+1}), \max_{q \in \mathcal{C}_p^l}(\overline{y}_q^{l+1})))$$

The MBR of a leaf node is identical to its subscription.

- $parent_p^l$ is the parent of node p at level l in the tree. The parent of the DR-tree root node is the node itself.

Joins. We assume that, at connection time, a subscriber (responsible for the node q) invokes an oracle that accurately provides a subscriber responsible for a node already in the structure (Figure 5.6). The join process may start from any node, but the odds of finding a good position for the new subscription are best when starting from the root. Therefore, the subscription is recursively redirected upward the tree until it reaches the root. The detailed pseudo-code of the join process is shown in Figures 5.4 and 5.5. The implementation of functions *ChooseBestChild* (select the subtree in which to insert a new node) and *SplitNode* (separate a leaf set into two sets and return both parents) are not shown as they depend on the splitting method and type of structure being used. As we have previously presented, DR-trees support three R-tree variants: linear, quadratic, *R**.

Upon reception of a connection request, a node s already in the structure chooses in its children set the child whose interests are closest to the new subscription (as determined by comparing MBRs). In addition, it pushes the request to the chosen child. This downward propagation process stops at the last non-leaf level (lines 1-6 in Figure 5.5). Having neighbors with similar interests helps minimize the occurrence of false positives. Our DR-tree structures support two variants for selecting the best branches when traversing down the tree to register a new subscription: 1) *R*: we choose the subtree that needs the least enlargement of its MBR to insert the new subscription; upon tie, we select the subtree with the smallest MBR [59]; 2) R^*: we proceed as above until we reach the last non-leaf nodes; then, we insert the new subscription in the node that needs the least overlap enlargement; upon tie, we select the node whose MBR needs the least area enlargement [14].

Assume that the join request reaches node p at the last non-leaf level. If the number of children alive of p is less than M, then p adds the new subscription q to its children set and it adjusts its MBR in order to include the new subscription. In case of MBR adjustment, p pushes the update upwards until it reaches the root (lines 7-10 in Figure 5.5). On the other

1: **ComputeMBR**$(p, l) \equiv$
2: $mbr_p^l \leftarrow ((\min_{q \in \mathcal{C}_p^l}(\underline{x}_q^{l+1}), \min_{q \in \mathcal{C}_p^l}(\underline{y}_q^{l+1})), (\max_{q \in \mathcal{C}_p^l}(\overline{x}_q^{l+1}), \max_{q \in \mathcal{C}_p^l}(\overline{y}_q^{l+1})))$

3: **IsBetterMBR**$(p, q, l) \equiv$
4: **return** $|mbr_q^{l+1}| > |mbr_p^{l+1}|$

5: **IsGoodMBR**$(p, l) \equiv$
6: **return** $mbr_p^l = ((\min_{q \in \mathcal{C}_p^l}(\underline{x}_q^{l+1}), \min_{q \in \mathcal{C}_p^l}(\underline{y}_q^{l+1})), (\max_{q \in \mathcal{C}_p^l}(\overline{x}_q^{l+1}), \max_{q \in \mathcal{C}_p^l}(\overline{y}_q^{l+1})))$

7: **ChooseBestParent**$(p, l) \equiv$
8: **select** $q \in \mathcal{C}_p^l, mbr_q^{l+1} = max|mbr_{\mathcal{C}_p^l}^{l+1}|$
9: $parent_q^l \leftarrow parent_p^l$
10: **forall** $s \in \mathcal{C}_p^l$ **do** $parent_s^{l+1} \leftarrow q$
11: $\mathcal{C}_q^l \leftarrow \mathcal{C}_p^l$
12: ComputeMBR(q, l)

13: **AdjustParent**$(p, q, l) \equiv$
14: **if** IsBetterMBR(p, q, l) **then**
15: $parent_q^l \leftarrow parent_p^l$
16: **forall** $s \in \mathcal{C}_p^l$ **do** $parent_s^{l+1} \leftarrow q$
17: $\mathcal{C}_q^l \leftarrow \mathcal{C}_p^l$
18: ComputeMBR(q, l)
19: **if** $\neg$ IsRoot(q, l) **then**
20: AdjustParent($parent_q^l, q, l-1$)
21: **else if** $\neg$ IsGoodMBR(p, l) **then**
22: ComputeMBR(p, l)
23: **if** $\neg$ IsRoot(p, l) **then**
24: send CHECK_MBR$(l-1)$ to $parent_p^l$

25: **AdjustChildren**$(p, q, l) \equiv$
26: $mbr_p^l \leftarrow mbr_p^l \bigcup mbr_q$
27: $\mathcal{C}_p^l \leftarrow \mathcal{C}_p^l \bigcup \{q\}$
28: $parent_q^{l+1} \leftarrow p$

Figure 5.4: Functions used by the Join, Leave, and Repair modules

hand, if the children set is bigger than M, p executes a split-children module that divides its children set in two groups, each having at least m elements (note that m must be chosen such that $M \geq 2m$). The invocation of this module aims at preserving the maximal and

1: **upon receive** JOIN(q, l) at node p at level l
2: $n \leftarrow$ ChooseBestChild(p, $filter_q, l$)
3: **if** $\neg$ IsLeaf(n,$l+1$) || ($l_p \neq l$) **then**
4: send JOIN $(q, l+1)$ to n
5: **else**
6: send ADD_CHILD (q, l) to p

7: **upon receive** ADD_CHILD(q, l) at node p
8: **if** $|\mathcal{C}_p^l| < M$ **then**
9: AdjustChildren(p, q, l)
10: AdjustParent(p, q, l)
11: **else**
12: $(left, right) \leftarrow$ SplitNode(p, q, l)
13: **if** IsRoot($left, l$) **then**
14: CreateRoot($left, right$)
15: **else**
16: send ADD_CHILD $(right, l-1)$ to $parent_{left}^l$

Figure 5.5: Join Phase requested by q and executed at node p

1: **upon receive** INITIATE_NEW_CONNECTION at l
2: send INITIATE_NEW_CONNECTION to $q, \forall q \in \mathcal{C}_p^l$
3: $parent_p^l \leftarrow p$
4: send JOIN (p, new_level) to GetContactNode(p, $\&new\_level$)

Figure 5.6: (Re)connection of node p

the minimal bounds on the nodes degrees (lines 11-16 in Figure 5.5). One of the subtree returned by the split procedure stays as a child of the node p, and this process adjusts its MBR, accordingly. A new root is created for the other subtree and is pushed backwards to p's parent. If the size of the parent's children set is less than M, then the parent adds the new root of the subtree to its children list. Otherwise, the parent recursively invokes the split-children module. Note that this process eventually stops with the split of the root, which generates the creation of two subtrees and the election of a new root.

DR-tree structure supports three original methods for splitting a children set, as presented in the previous chapter:

- The *linear* method [59] chooses two children from the overflowing node such that they are as far apart as possible and assign the remaining children, randomly chosen,

to the nodes based on the MBR coverage.

- The *quadratic* method [59] chooses two child seeds such that the union of their MBRs would generate the largest dead space area. The remaining nodes are assigned also based on the MBR coverage, but not randomly.

- The *R*-tree* splitting method [14] attempts to reduce not only the coverage, but also the overlap. Instead of just splitting the node when it overflows, it also tries to allocate some entries to a better suited node through reinsertion.

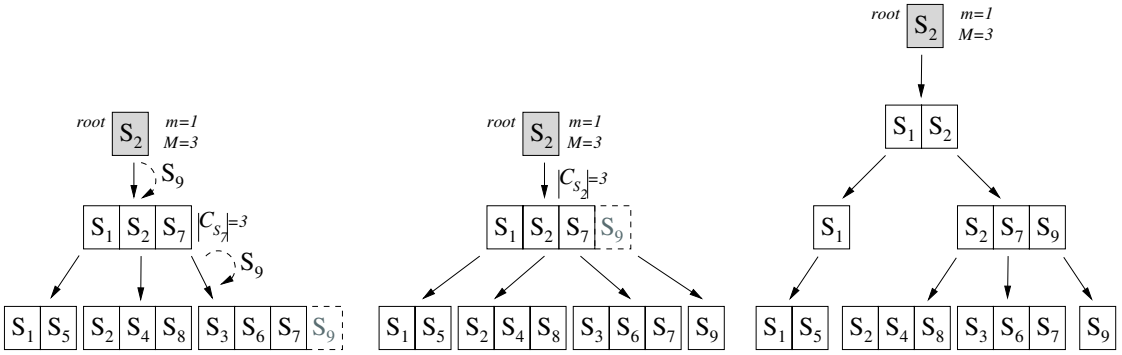


Figure 5.7: Example of the join procedure (using quadratic splitting method) for the set of subscriptions of Figure 4.8.

Figure 5.7 illustrates an example of the join procedure of node S_9 for the subscriptions of Figure 4.8. The new subscription S_9 is pushed downwards the tree according to the MBR coverage until it reaches the last non-leaf node S_7. Since S_7 has already 3 children, its children set should split (based on quadratic method) in order to allocate the new subscription. Thus, the node S_7 keeps in its children set the nodes S_3, S_6 and itself, and the node S_9 is added in a different children set with parent S_9 as well. The new parent S_9 is pushed upwards in the tree level. Thus, the children set of the root S_2 also must split once it overflows with the new parent S_9. Finally, two children set rooted by S_1 and S_2, respectively, are created.

Controlled Departures. We now describe the repair algorithm (Figure 5.8, 5.9 and 5.12) executed whenever a subscriber leaves properly the system by sending a LEAVE message to

```
1: RepairChildren(q, l) ≡
2:    n ← parent_q^l
3:    C_q^l ← C_q^l \ {q}
4:    if |C_q^l| < m then
5:       send INITIATE_NEW_CONNECTION to all s ∈ C_q^l
6:       parent_q^l ← ∅
7:    else
8:       ChooseBestParent(q,l )
9:    if n = q ∧ ¬ IsRoot(q,l) then
10:      RepairChildren(q, l − 1)
```

Figure 5.8: Repair the subtree of leaving node q

```
1: upon receive LEAVE(q, l) at node p
2:    if p = q then
3:       RepairChildren(q, height)
4:       send LEAVE(q, l − 1) to parent_q^l
5:    else
6:       send CHECK_STRUCTURE(l) to p
```

Figure 5.9: Leave Phase executed at node p

the parent p of its topmost instance q in the DR-tree (Figure 5.9) and executing *RepairChildren* (Figure 5.8). Upon receiving a leave message, p checks its children set (lines 2-3 in Figure 5.12). Since the size of the children set may drop below m after q leaves, the subtree that contains the departing node q must be repaired. In this case, the children set must be reinserted in the tree. Hence, for each child of p, the whole subtree rooted by this child is reinserted in the tree at the same level l and p is removed from the tree (lines 11-15 in Figure 5.12). On the other hand, if the size of the children set remains between m and M, the MBR of p must be updated since it may become smaller (lines 26 in Figure 5.12) and it may happen that another node (part of the p's children set) exchange position with p (lines 16-19 in Figure 5.12). After updating its children set and its MBR, the node p sends a message CHECK_STRUCTURE to its parent. This process is recursively redirected upward the tree until it reaches the root.

Since the leave message is sent to the parent p of the topmost instance of q, the subtree rooted at q must be repaired as well. Therefore, before q leaves the system, it checks each of its children set at each level starting from the leaves. If the children set will become

smaller than m, when q leaves, it sends a INITIATE_NEW_CONNECTION message for each child (lines 4-6 in Figure 5.8). Otherwise, a new parent must be elected and updated on the path to the node p (lines 7-10 in Figure 5.8). Note that the MBRs are also updated.

An example of the leave procedure is shown in Figure 5.10. Upon the departure of node S_9, the children sets in the right subtree become less than 2. Note that the DR-tree in this example has $m = 2$ and $M = 4$. In this way, the node S_{10} and the subtree rooted by S_7 are reinserted in the tree.

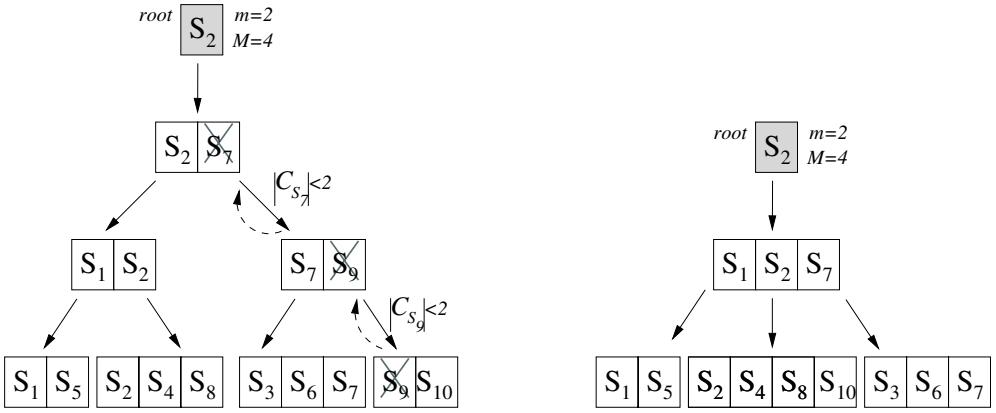


Figure 5.10: Example of the leave procedure for the set of subscriptions of Figure 4.8.

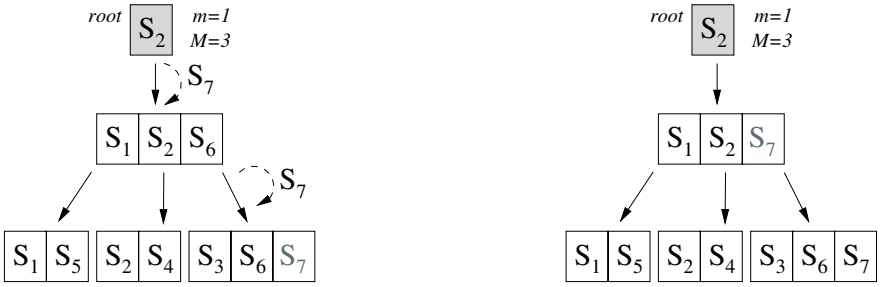


Figure 5.11: Example of the dynamic reorganization for the set of subscriptions of Figure 4.8.

Dynamic Reorganizations. There are two situations where nodes may dynamically reorganize to improve the accuracy of the underlying DR-tree structure. First, each internal node in the tree periodically checks if it is the best cover for subtree, i.e., if it has the biggest

MBR of the children set. If one of its children provides better coverage (e.g., because its MBR has grown after the insertion of a new node), then the nodes exchange their positions. This scenario can occur during join, splitting, and if the tree is corrupted (to be discussed later).

Second, under biased event workloads, it may happen that the organization of the DR-tree (computed statically so as to minimize MBR coverage) may perform poorly because small false positive regions are hit by many events while larger areas see none. If a child is interested in many events that its parent does not care about, then both nodes exchange their positions.

An example of dynamic reorganization upon join of the node S_7 is illustrated in Figure 5.11. Since S_7 provides better coverage than S_6, they change positions and S_7 becomes the new parent of the subtree.

5.3 Self-stabilizing Distributed R-trees

The implementation of publish/subscribe systems in dynamic environments where continuous service has to be guaranteed despite churn (frequent connections/disconnections) requires self-stabilizing algorithms in order to ensure correct operation in all circumstances.

The occurrence of faults can change the state of the system. Thus, the self-stabilization algorithm (the concept was first introduced by Dijkstra [41]) is responsible to converge from any arbitrary state to the desired behavior in a finite number of steps [42]. In contrast to that, upon transient faults, a system that is not self-stabilizing may never leave the illegitimate state.

Therefore, self-stabilizing publish/subscribe systems can automatically recover after transient faults (e.g., node's failure, message losses or corruption). Many self-stabilizing algorithms for publish/subscribe systems have been proposed in the literature [62, 63, 78, 99, 115]. Similar to our work, [34] proposes a reconfigurable publish-subscribe system over a tree-based distributed hash table in order to cope with nodes failures in dynamic environments. In the following we present our self-stabilization algorithms for our DR-tree overlays.

1: **upon receive** CHECK_STRUCTURE(l) at node p
2: **while** $\exists s \in \mathcal{C}_p^l, parent_s^{l+1} \neq p$ **do**
3: $\mathcal{C}_p^l \leftarrow \mathcal{C}_p^l \setminus \{s\}$
4: **if** IsRoot(p, l) **then**
5: **if** $|C_p^l| = 1$ **then**
6: $s \in \mathcal{C}_p^l$ **do** $parent_s^{l+1} \leftarrow \varnothing$
7: **else**
8: **if** $\nexists s \in \mathcal{C}_p^l, s = p$ **then**
9: ChooseBestParent(p,l)
10: **else**
11: **if** $|C_p^l| < m$ **then**
12: send INITIATE_NEW_CONNECTION to all $s \in \mathcal{C}_p^l$
13: $t \leftarrow parent_p^l$
14: $parent_p^l \leftarrow \varnothing$
15: send CHECK_STRUCTURE($l-1$) to t
16: **else**
17: **if** $\nexists s \in \mathcal{C}_p^l, s = p$ **then**
18: ChooseBestParent(p,l)
19: send CHECK_STRUCTURE($l-1$) to $parent_p^l$
20: **if** $|C_p^l| > M$ **then**
21: $(left, right) \leftarrow$ SplitNode(p, q, l)
22: **if** IsRoot($left, l$) **then**
23: CreateRoot($left, right$)
24: **else**
25: send ADD_CHILD $(right, l-1)$ to $parent_{left}^l$
26: CHECK_COVER(l) to p

Figure 5.12: Repair DR-tree structure at node p

5.3.1 Overlay Stabilization

The overlay stabilization process implements the self-stabilization of the mbr variables and the tree structure, and checks that the overlay respects the DR-tree specification. This verification is performed periodically due to the dynamism of the environment. That is, at each node in the DR-tree, the following events are triggered periodically at each level of the tree: CHECK_MBR, CHECK_COVER, CHECK_PARENT, CHECK_STRUCTURE. Note that the time interval that the events are triggered may be adapted according to each level, i.e., nodes situated at high level may trigger the events more often than the leaf nodes once the peers responsible for these nodes have higher degree.

Correction of the MBR values. In a correct state, the MBR of a leaf node equals its filter while the MBR of a non-leaf node is the smallest rectangle that covers the MBRs of its children. Upon the reception of a CHECK_MBR event at level l, each node checks the correctness of its MBR value and repairs it in case of anomaly (see Figure 5.13). In case of MBR update, the node sends the CHECK_MBR message to its parent at level $l-1$ and the message is recursively propagated upwards the tree until it reaches the root (lines 9-10 in Figure 5.13). This propagation is optional in the algorithm. If the rate of fails in the structure is high, then the event is frequently executed at all levels, consequently, the propagation of the event becomes redundant.

```
1: upon receive CHECK_MBR(l) at node p
2:    update ← false
3:    if IsLeaf(p, l) ∧mbr_p^l ≠ filter_p then
4:       mbr_p^l ← filter_p
5:       update ← true
6:    if ¬ IsLeaf(p, l) ∧¬IsGoodMBR(p, l) then
7:       ComputeMBR(p, l)
8:       update ← true
9:    if ¬ IsRoot(p, l) ∧ update then
10:         send CHECK_MBR(l − 1) to parent_p^l
```

Figure 5.13: Repair MBR at node p

Correction of the cover. Due to some modifications in the tree structure, a child of a node may better cover the node subtree than the node itself because the child has a bigger MBR than the node at the same level. In that case, the node and the child exchange their roles. Also, the MBR of the new parent must be updated, as well as the MBRs and the parents of all ancestor nodes on the path to the root. This correction is performed by the module proposed in Figure 5.14. Note that the update of the ancestors (lines 10-11 in Figure 5.14) is necessary only in stable environments where the event is not triggered frequently.

Correction of the DR-tree structure. Transitory faults or uncontrolled departures may have a dramatic impact on the DR-tree structure. Therefore, we reinforce the system by adding modules that deal with the different scenarios of corruption. The DR-tree structure is corrupted if: (a) the parent or the children set is corrupted (i.e., the parent does not appear

1: **upon receive** CHECK_COVER(l) at node p
2: **if** $\exists q \in \mathcal{C}_p^l$, IsBetterMBR($p, q, l$) **then**
3: **if** $\neg$ IsRoot(p, l) **then**
4: $parent_q^l \leftarrow parent_p^l$
5: **else**
6: $parent_q^l \leftarrow q$
7: **forall** $s \in \mathcal{C}_p^l$ **do** $parent_s^{l+1} \leftarrow q$
8: $\mathcal{C}_q^l \leftarrow \mathcal{C}_p^l$
9: ComputeMBR(q, l)
10: **if** $\neg$ IsRoot(q, l) **then**
11: AdjustParent($parent_q^l, q, l-1$)

Figure 5.14: Repair Cover at node p

in the parent variable of its children or it does not appear in the children list of its parent); (b) the size of the children set drops under the limit m or becomes bigger than M. Each one of these situations is corrected by one of the modules shown in Figures 5.12 and 5.15 as explained in the following. Note that the insertion of new nodes in the structure cannot create corruptions of the DR-tree state.

A transient fault may corrupt the $parent$ variable. In order to stabilize the DR-tree structure in this respect, each node verifies via module 5.15 if it appears in the list of children of its parent and if it has a parent. If that is not the case, then the node initiates a join process. Note that the node will be reinserted in the tree at the same level.

1: **upon receive** CHECK_PARENT(l) at node p
2: **if** $p \notin \mathcal{C}_{parent_p^l}^{l-1} || GetParent(p, l) = \varnothing$ **then**
3: send JOIN (p, l) to GetContactNode(p, l)

Figure 5.15: Repair Parent at node p

By specification, each node has to have between m and M children. In order to avoid underloaded (less than m children) or overloaded (more than M children) nodes resulting in an imbalanced tree, each node verifies periodically if its structure is correct by executing CHECK_STRUCTURE. Thus, when a children set drops below m, due to uncontrolled departures, the children set is reinserted in the tree (lines 11-15 in Figure 5.12). On the other hand, if the children set becomes bigger than M, the parent splits the children set as in the join phase (lines 20-25 in Figure 5.12).

Persistent event dissemination. During the reorganization of the tree events may not be delivered to interested parties due to controlled/uncontrolled departures. In order to improve the reliability of the message dissemination, one can use various well-known techniques (e.g., buffering messages in the peers).

Each peer may have one queue for each communication channel with its neighbor. Once the structure is stabilized, the events may be re-sent to the subscribers. Thus, subscribers may request from each other a list of notifications they have sent and, if a subscriber does not find a specific event, it requests its retransmission. However, the time interval that an event remains stored in the peers has a strong influence on the reliability of the system. Three classes of persistence were proposed in [7], where each strategy has a different impact in the memory size required in the system. They are classified as follows:

0**-persistence:** the events do not remain available in the network, expiring as soon as they are disseminated. One example of an application in this class is a stock quote. Thus, only subscribers available at the very moment of the event dissemination are considered for the event delivery. Since the publication rate is typically high in such applications, missing events are not considered critical.

Δ**-persistence:** each event expires after $\Delta > 0$ from the instant that it is published. After the time interval expires, a garbage collection is performed removing all expired notifications. Daily news can be viewed as an example of Δ-persistence application. In this case, the subscribers have 1 day to visualize the notification. Note that high values for Δ may generate out-of-date notifications.

∞**-persistence:** each event remains permanently available in the system. When a subscriber registers a new subscription, it will receive all the events previously published that match the subscription. An example of an application based on this classification is a digital library. Upon the arrival of a new subscriber, all the catalog updates previously published will be notified to the new user. Obviously, this implementation requires unlimited memory to store all the events published in the system.

5.3.2 Overlay correctness

In this section we present the correctness of the computed overlay. We first show that join and controlled departure operations do not corrupt the DR-tree structure. Then, we show the stabilization of the structure once the local variables are corrupted by a transient fault or disconnection of subscribers without notification.

Definition 5.3.1 *The DR-tree is in a legal state iff the following conditions are verified:*

- *each non-root and non-leaf node in the tree*[1] *has at least* m *and at most* M *children;*
- *all the leaf nodes are at the same level;*
- *for each node* p *in the DR-tree overlay, the parent and children variables are coherent:*
  - *if* p *is the parent of node* q *at level* l *then* q *belongs to the children set of* p *at* $l-1$*;*
  - *if* q *is the child of* p *at level* l *then* q *has parent variable set to* p *for level* $l+1$*;*
- *for each node* p *at level* l *there is no child* q *such that* q *offers a better cover for the subtree of* p *at* l*;*
- *the MBR value of each non-leaf node* p *at level* l *is the union of the MBR values of its children at level* $l+1$.

Definition 5.3.2 (legitimate configuration) *Let S be the system executing the algorithms described in section 5.2. The system is in a legitimate configuration c iff the virtual structure defined by the* $parent$ *variables and the children sets is a legal DR-tree.*

Lemma 5.3.1 *In a legitimate configuration the height of the DR-tree is* $O(\log_m(N))$ *while the space complexity for the structure maintenance is* $O(M \log^2(N)/\log(m))$.

Proof. Since the number of children for each node is at least m and there are N nodes in the network, then the height of the tree is $O(log_m(N))$. A peer, in the worst case, may be responsible for one node at each level of the tree and for each level corresponding node maintains at most M children. Since the height of the tree is $O(log_m(N))$ and the cost to store each child is $O(log(N))$, then the space needed to store all the children at all levels is $O(M \log^2(N)/\log(m))$.

Lemma 5.3.2 (stabilization after joins) *Let* c *be a legitimate configuration and let* q *be a new subscription joining the system in* c*. Let* c' *be the new configuration of the system after* q *executes the join operation (Figure 5.5).* c' *is a legitimate configuration and the stabilization time of the structure after the insertion of a new node is* $O(log_m(N))$.

[1] Note that a peer can be responsible for several nodes in the tree.

Proof. Once a node q joins the network, the join request is forwarded downward to the last non-leaf node p. In order to guarantee correctness of the MBRs and the covers (the node q may better cover its parent p), they are updated (if necessary) during upward traversal of the tree from p to the root. Since the height of the tree is $O(log_m(N))$ (Lemma 5.3.1), it takes $\log_m(N)$ steps to choose the children set in which the new node will be inserted and; $\log_m(N) + 1$ steps to update and/or split all the children set along the subtree considering the worst case where the root is split generating a new level. Thus, the stabilization is reached in $3\log_m(N) + 1$ steps. Overall, the join process does not introduce anomalies in the DR-tree structure and the structure eventually stabilizes to a legal configuration.

Lemma 5.3.3 (stabilization after controlled departures) *Let c be a legitimate configuration and let q be a subscriber leaving the system via a controlled departure in c. Let c' be the new configuration of the system after q executes the controlled departure operation (Figure 5.8). c' is a legitimate configuration and the complexity is $O(m\log_m^2(N))$.*

Proof. The controlled departure operation consists in informing the parent the intention to leave. However, before q sends a message to its parent p, q must advertise its children set at each level so they can elect a new parent or rejoin the system. After q leaves the system, p must update its children set and its MBR; and forward the update upward the tree until it reaches the root. Considering the worst case scenario, all the children set along the subtree become less than m and they should be reinserted in the tree. Since the join procedure for each node takes $O(\log_m(N))$ and the height of the tree is $O(log_m(N))$, then in the worst case the stabilization takes $(m-1)(3/2\log_m^2(N) + \log_m(N))$ steps.

Lemma 5.3.4 (stabilization after uncontrolled departures) *Let c a legitimate configuration and let q be a node leaving the system via an uncontrolled departure (failure) in c. The system reaches a legitimate configuration c' in a finite number of steps and the complexity is $O(M\log_m^2(N))$.*

Proof. Since the system is designed to be self-stabilizing, a node that leaves the system without notification is detected by its children and parent. Each node in the tree periodically triggers the event CHECK_PARENT and CHECK_STRUCTURE. Thus, after triggering the event CHECK_PARENT, q's children will detect the departure of q and will initiate a join process. If the child is a non leaf node then the whole subtree is reinserted as in a

controlled leave operation. Finally, once the departure is detected by q's parent through the event CHECK_STRUCTURE, the rest of the stabilization follows the Lemma 5.3.3. Consider the worst case scenario where q is the root of the tree and it appears at all levels in the tree. Then, all the children sets at all levels in the subtree will be reinserted in the tree. Since the join procedure for each node takes $O(\log_m(N))$ and the height of the tree is $O(log_m(N))$, then the in the worst case the stabilization takes $(M-1)(3/2\log_m^2(N)+\log_m(N))$ steps.

Lemma 5.3.5 (stabilization after memory corruption) *Let c be an initial arbitrary configuration of the system. The system reaches a legitimate configuration c' in a finite number of steps.*

Proof. In an arbitrary configuration the variables mbr and $parent$ and the children sets may be corrupted. Each node periodically triggers the events CHECK_MBR, CHECK_COVER, CHECK_PARENT and CHECK_STRUCTURE. Once these events are triggered the local variables are corrected via the modules shown in Figures 5.13, 5.14, 5.15 and 5.12. Note that the correction of the variable $parent$ is done via an insertion of the faulty process. The correctness of the join operation follows from Lemma 5.3.2. The children set is repaired following Figure 5.12. If the root of a subtree has a child which provides better cover for the subtree then the two nodes exchange their positions in the tree by executing the module shown in Figure 5.14.

The DR-tree integrates modules that repair the overlay as soon as a corruption is detected. The recover process is totally dependent on the value of the "timeout" and the stabilization time of the structure. As shown in the previous lemmas, for most of the faults the recovery time is $O(\log_m^2(N))$. Studies have showed that, in a peer-to-peer systems, the longer a node stays available in the network, the higher is the probability that it remains connected [55, 93]. In order to avoid a significant number of rejoin operations under high churn, the structure organization may be based also on the time that the peer has been connected in the network. Hence, peers with high availability may be placed near to the root, upon tie, we remain the organization based on the MBR value. The implication of this approach is that the number of false positives in the system may increase, however the reorganization cost will minimize under high uncontrolled departures.

However, in environments prone to high churn the structure may never be able to self-repair. Therefore, it is interesting to study the limits of our overlay. The following lemma

computes the bound on the expected time the DR-tree gets disconnected due to frequent departures. We recall that joins have no impact on the overlay connectivity.

Lemma 5.3.6 (DR-tree churn resistance) *Let Δ be an interval of time during which no stabilization operation is triggered and let λ be the rate of departures.*[2] *The expected time before the DR-tree disconnects is $\frac{\Delta}{N} e^{\frac{m(N-\Delta\lambda)^2}{N4\Delta\lambda}}$.*

Proof. The proof uses the technique proposed in [74] for the Multi-Chord system. Let p be an internal node in the system and let λ the rate of leaves. In a steady configuration, p has at least m children. The average number of leaves occurred in the children sets of p during the interval Δ is $\mu = \Delta\lambda\frac{m}{N}$, where $\Delta\lambda$ is the average number of leaves occurring in the system. By applying the Chernoff bound, the probability that more than $(1+\delta)\mu$ events occur is: $Pr(X > (1+\delta)\mu) < e^{-\mu\frac{\delta^2}{4}}$. The probability that more than m leave events produced in the children sets of p is: $Pr(X > m) < Ne^{-\frac{(m-\mu)^2}{4\mu}}$.

Given an interval Δ where the rate of leaves is λ, the probability that more than m leave events occur in the children sets of any node in the system is: $p(\Delta) \leq NPr(X > m) < Ne^{-\frac{(m-\mu)^2}{4\mu}}$.

Overall, the mean time before the DR-tree disconnects is: $T_d = \frac{\Delta}{p(\Delta)} > \frac{\Delta}{N} e^{\frac{\frac{m}{N}(N-\Delta\lambda)^2}{4\Delta\lambda}}$.

5.4 Conclusion

In this chapter we have studied a class of distributed R-trees and their applicability to build content-based publish/subscribe overlays. Our study has focused on the properties of the resulting topology and the accuracy of event dissemination (occurrence of false positives and false negatives).

Distributed R-trees, proposed in this chapter, are a decentralized implementation of the R-tree structure and its variants, which, were first introduced in the context of databases [59], as response to the problem of finding adequate storage structures for complex database objects.

These self-stabilizing peer-to-peer overlays are well adapted to publish/subscribe systems with complex subscriptions (multi-dimensional) as geometrical representation captures well the semantics of the subscriptions. Also our overlays can cope with the dynamism of the system. The overlays are designed such that they eradicate false negatives

[2] We consider arrivals and departures modeled by a Poisson distribution.

and drastically drop the number of false positives. Moreover, organizing the peers based on their interests minimizes the amount of false positives in the system and events can be quickly disseminated.

DR-trees are fault-tolerant overlays designed for dynamic systems. The overlays provide logarithmic guarantees for the publish/subscribe operations and use only polylogarithmic memory per node.

In addition, we have proved the correctness of our overlay under static and dynamic assumptions. Note that DR-trees generalize P-trees [36], which are the dynamic version of B+-trees. B+-trees are ancestors of R-trees designed to handle ranges and inequalities.

The experimental evaluation of the Distributed R-tree overlays will be presented in Chapter 7.

CHAPTER

6 Distributed Hilbert R-trees

6.1 Introduction

In this chapter, we propose three different Distributed Hilbert R-tree overlays (*DHR-tree*), which are based on Hilbert space filling curves [60]. As DR-trees, DHR-trees are also a class of content-based publish/subscribe overlays where publishers and subscribers self-organize in a balanced tree peer-to-peer overlay based on the similarity between the subscriptions.

The overlays mainly differ by the data structure that each peer stores (geometrically and/or Hilbert-based) and the strategies used to build the tree (join and split methods). We show that, there is a tradeoff between the routing accuracy and the cost in space to index the subscriptions.

The rest of this chapter is organized as follows. Section 6.2 describes the three DHR-tree overlays: DHR-tree Center, DHR-tree Min/Max and DHR-tree Segment. As the join, leave and self-stabilizing algorithms are similar to the DR-trees (see Chapter 5), we only present examples to illustrate the functions *ChooseBestChild* (select the subtree to insert a new subscription) and *SplitNode* (split a children set into two groups). The summary and discussion of the overlays are presented in Section 6.3.

6.2 Distributed Hilbert R-tree Overlays

As in DR-trees, we assume that each peer subscribes for certain interest, which this subscription is stored in the leaf node of the tree. In addition, a peer may have the role of forwarder thus being responsible for certain internal nodes of the tree. Each node p in the

overlay maintains the following variables:

- $filter_p = S_p$ represents the subscription to which every peer in the system subscribes.
- mbr_p^l or mbp_p^l represents the minimum bounding rectangle (MBR) or the minimum bounding polytope (MBP) that includes all the MBRs/MBPs of all children at level l. The MBP is equivalent to the MBR thus we will show in the Subsection 6.2.3 that in a tree built completely based on Hilbert space filling curves (we refer to as Hilbert SFC), the MBP is not necessarily represented geometrically as a poly-rectangle but as a polytope in the Euclidean space.
- C_p^l is the children set of p at level l. Note that the subscribers responsible for the internal nodes have to maintain different children sets for each level that it is present.
- $parent_p^l$ is the parent of the node p at level l in the tree. Note that the parent of the DR-tree root node is the node itself.

6.2.1 Distributed Hilbert R-trees Center

This variant extends the Hilbert R-trees [64] presented in Section 4.4. The subscriptions are organized in an R-tree structure based on Hilbert values, which are used to guarantee that similar subscriptions are gathered together in the same subtree. As aforementioned, each node in the tree keeps track of its subscription and MBR. Each node stores, in addition, the Hilbert value associated to the centroid coordinate of the subscription. The three variables are computed as follows:

- $S_p = ((\underline{x}_p, \underline{y}_p), (\overline{x}_p, \overline{y}_p))$, where $\underline{x}_p$ and $\underline{y}_p$ represent the minimal abscissa/ordinate, and $\overline{x}_p$ and $\overline{y}_p$ the maximal abscissa/ordinate of the subscription.
- $mbr_p^l = ((\underline{x}_p^l, \underline{y}_p^l), (\overline{x}_p^l, \overline{y}_p^l))$ includes all the MBRs of all its children at level l, and, as in DR-trees, is computed as:

$$mbr_p^l = (((\min_{q \in \mathcal{C}_p^l}(\underline{x}_q^{l+1}), \min_{q \in \mathcal{C}_p^l}(\underline{y}_q^{l+1})), (\max_{q \in \mathcal{C}_p^l}(\overline{x}_q^{l+1}), \max_{q \in \mathcal{C}_p^l}(\overline{y}_q^{l+1}))))$$

- ℓhv_p^l represents the Hilbert value used to guarantee the ordering of the nodes at level l. The ℓhv of a non-leaf node is the largest Hilbert value among its children: $\ell hv_p^l =$

$\max_{q \in \mathcal{C}_p^l}(\ell hv_q^{l+1})$. The ℓhv is computed only for the leaf-nodes as the Hilbert value of the center of the subscription: $\ell hv_p^l = hv((\frac{\underline{x}_p^l + \overline{x}_p^l}{2}), (\frac{\underline{y}_p^l + \overline{y}_p^l}{2}))$. Note that depending on the order of the space filling curve, the curve may not intersect the center of the subscription. In this case, the Hilbert value stored at the leaf node is the one nearest to the center.

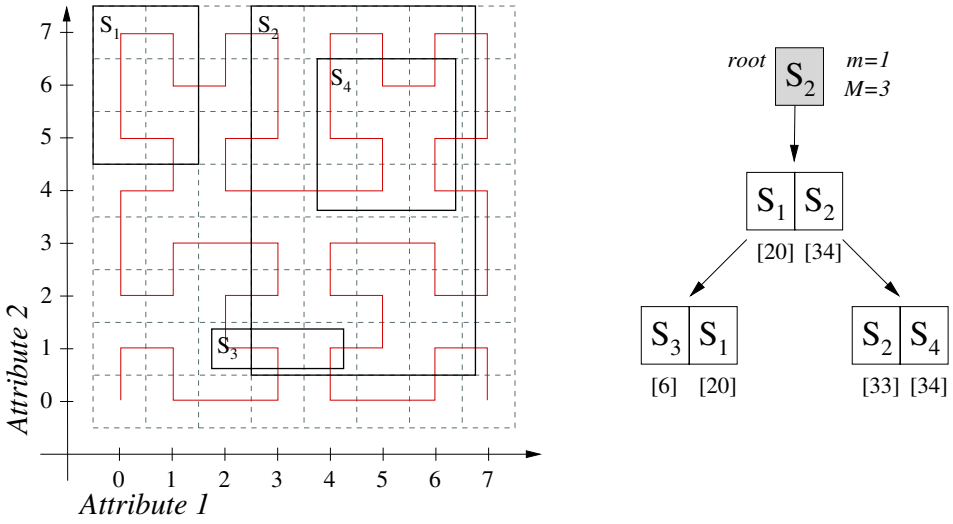


Figure 6.1: Example of splitting method in DHR-tree center.

Overlay Organization. The insertion of a new node in the system is only based on the Hilbert value used to preserve the proximity between the nodes at the same level. Preserving the proximity, i.e., organizing the subscriptions based on their similarity helps to minimize the occurrences of false positives. Given a joining node q and a node p already in the structure, node p forwards to the child whose $\ell hv_{s \in \mathcal{C}_p^l}^{l+1}$ is the smallest among the children set and $\ell hv_{s \in \mathcal{C}_p^l}^{l+1} \geq \ell hv_q$. If the ℓhv of the new node is bigger than the largest ℓhv among all nodes in the tree, the node is allocated in the children set of the parent node with the largest ℓhv. In this case, the ℓhv of the parent must be updated. Upon tie, the joining node is redirected to the branch that requires the smallest enlargement of the MBR. Even if there are several subtree choices, the node is redirected to the node with the biggest MBR.

The splitting method in this variant is straightforward. Since, at each level the nodes are ordered based on the Hilbert values it is only necessary to define where the splitting

takes place in order to evenly distribute the nodes among the groups. Once the overflowing node is split into two groups, the parent for each subtree is the one whose current MBR is the largest to preserve the containment relationship. An example of the splitting procedure is shown in Figure 6.1. The left hand side of the Figure illustrates four subscriptions that will split into two groups (where $m = 1$ and $M = 3$). The right hand side of the figure presents the DHR-tree Center after the splitting. In this example, one group is composed by S_3 and S_1 as their Hilbert values are the smallest and the other group is composed by S_2 and S_4 as their Hilbert values are the biggest.

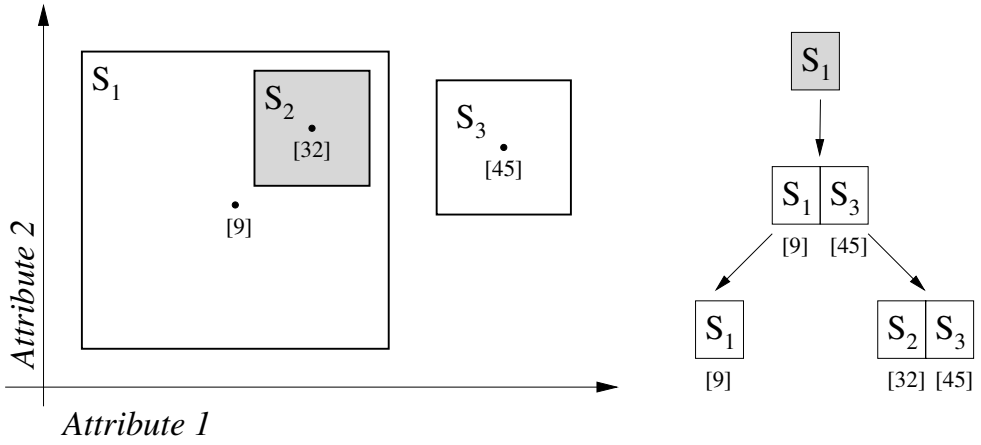


Figure 6.2: Example of subscriptions with containment relationship that is not preserved in the overlay.

Even though the root election preserves the containment relationships, the insertion and splitting procedures may not guarantee that two subscriptions that contain each other are allocated in the same subtree. An example is illustrated in Figure 6.2. Consider the subscriptions S_1 and S_3 already in the structure and a joining subscription S_2. The subscription S_1 contains S_2, however S_2 is inserted in the subtree rooted by S_3 as ℓhv_{S_3} is the smallest Hilbert value greater than ℓhv_{S_2}.

On the other hand, the joining and the splitting methods are only based on a single integer value ℓhv_p^l, which is much less complex than using n-dimensional computations as in the original R-trees. Figure 6.3 shows the overlay (re-)organization for the set of subscriptions of Figure 4.8.

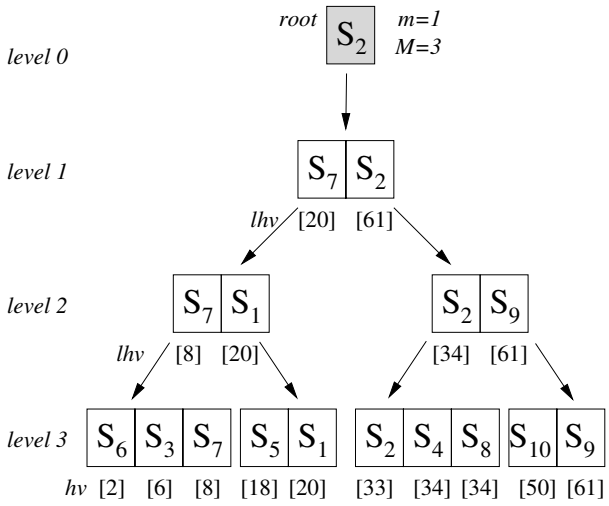


Figure 6.3: DHR-tree center for the subscriptions of Figure 4.8.

Event Dissemination. In order to avoid false negatives, the event dissemination is based on the MBR and not on the Hilbert value. When a node receives an event, it forwards the event to each child, which its MBR contains the event (i.e., event falls in the range of the MBR for each of the dimensions), unless the message was already received by that child. Therefore, the Hilbert value is not considered in the event dissemination and only used during the overlay organization.

6.2.2 Distributed Hilbert R-trees Min/Max

This variant aims to minimize the amount of memory used to store the MBR and the matching complexity at internal nodes. The subscriptions and MBRs are represented as:

- $S_p = ((\underline{x}_p, \underline{y}_p), (\overline{x}_p, \overline{y}_p))$, where $\underline{x}_p$ and $\underline{y}_p$ represent the minimal abscissa/ordinate, and $\overline{x}_p$ and $\overline{y}_p$ the maximal abscissa/ordinate of the rectangle that circumscribes the filter.

- $mbr_p^l = (\underline{hv}_p^l, \overline{hv}_p^l)$ represents the smallest and the largest Hilbert value among its children set:

$$mbr_p^l = ((\min_{q \in \mathcal{C}_p^l}(\underline{hv}_q^{l+1}), \max_{q \in \mathcal{C}_p^l}(\overline{hv}_q^{l+1}))$$

The MBR of a leaf node is computed as the smallest and the largest Hilbert value from all the coordinates of its subscription.

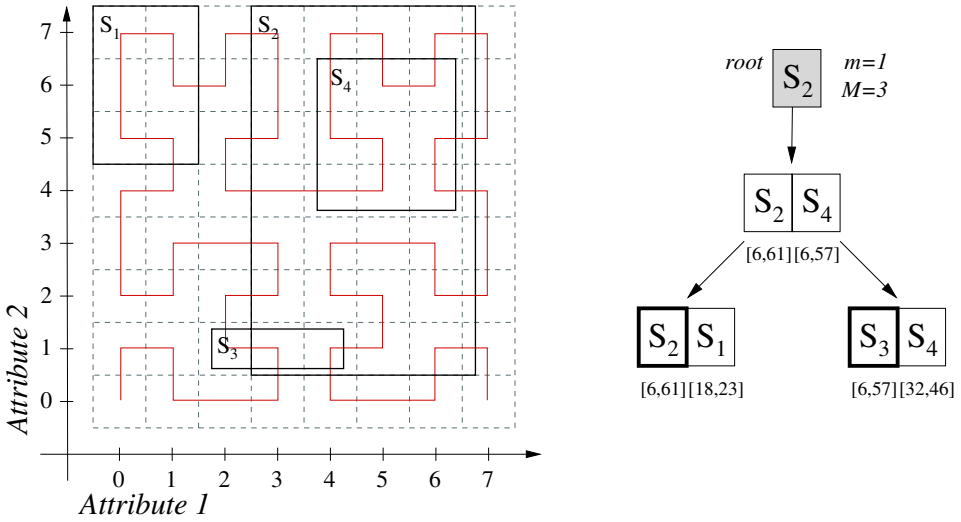


Figure 6.4: Example of splitting method in DHR-tree min/max.

Overlay Organization. The join procedure is only based on the *min/max* Hilbert values at each level and the chosen subtree is the one whose distance between the *min/max* values of the joining node q and the node p, already in the tree, is the smallest. Upon tie, we do not choose the child with the biggest distance between the *min/max* values in order to avoid that all subscriptions are inserted in the same branch. This strategy is also used in the splitting method. First, two children s and r (also called seed), from the overflowing node, such that $\underline{hv}_s^{l+1}$ and $\overline{hv}_r^{l+1}$ are the extremities, are placed in separate groups. If the same node has the smallest and highest Hilbert value, we choose a second highest Hilbert value among the children set. Then, each remaining child is assigned in the group for which the distance between the *min/max* values between the child and the seed is the smallest. The splitting procedure is illustrated in Figure 6.4. The left hand side of the Figure illustrates four subscriptions that will split into two groups (where $m = 1$ and $M = 3$) and the right hand side the DHR-tree Min/Max resulted from the splitting. In this case, one group is composed by S_2 and S_1, where S_2 is the seed node and the other group is composed by S_3 and S_4 with S_3 being the seed node for that group.

The new parent of the subtree is the one with the biggest distance between the $\underline{hv}_p^l$ and $\overline{hv}_p^l$ values of its MBR. Note that the biggest distance does not guarantee the largest MBR in terms of area. Figure 6.5 presents the resulting DHR-tree Min/Max for the set of subscriptions of Figure 4.8. We can observe that this variant may not guarantee that subscriptions with containment relationship are placed in the same subtree. For example, the subscription S_8 is inserted in the children set of subscription S_1. Considering the root election, a subscription with large scope is more likely to have bigger distance between $\underline{hv}_p^l$ and $\overline{hv}_p^l$, which may help to preserve the containment relationship.

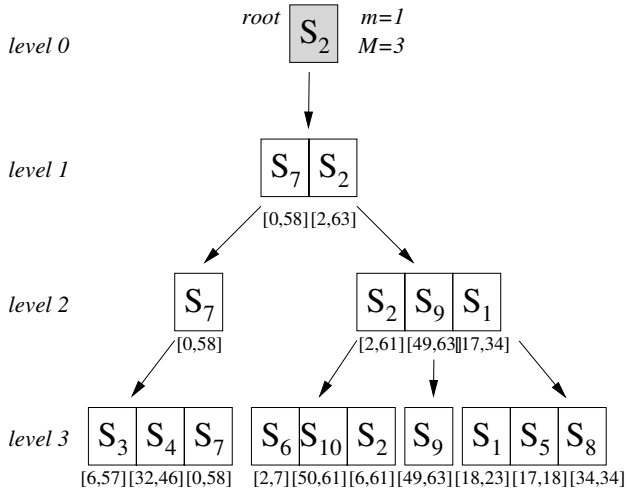


Figure 6.5: DHR-tree min/max for the subscriptions of Figure 4.8.

Event Dissemination. The coordinate of an event e is also associated to a Hilbert value hv_e. Thus, an event is forwarded to the child p whose hv_e falls in the range $\underline{hv}_p^l$ and $\overline{hv}_p^l$. As the MBR is computed from all the coordinates contained in the subscription, there are no false negatives. Of course, this variant may produces higher amount of false positives compared with the overlays that are based on the MBR for the event dissemination. However, the memory cost of indexing the MBRs is not only reduced, but the matching procedure is also less complex.

6.2.3 Distributed Hilbert R-trees Segment

In this variant, all the information stored by the nodes is transformed in 1-dimensional using Hilbert SFC. Each node stores the following information:

- $S_p = ((\underline{hv}_1, \overline{hv}_1), ..., (\underline{hv}_i, \overline{hv}_i))$ is composed of a finite set of range values and each range value represents a cluster (i.e., a group of consecutive points in the curve) in the Hilbert curve.
- $mbp_p^l = ((\underline{hv}_1^l, \overline{hv}_1^l), ..., (\underline{hv}_i^l, \overline{hv}_i^l))$ is also composed of a set of Hilbert range values that includes all the MBPs from the children set at level l. The MBP is computed as follows:

$$mbp_p^l = \cup_{q \in \mathcal{C}_p^l}^{l+1} S_q$$

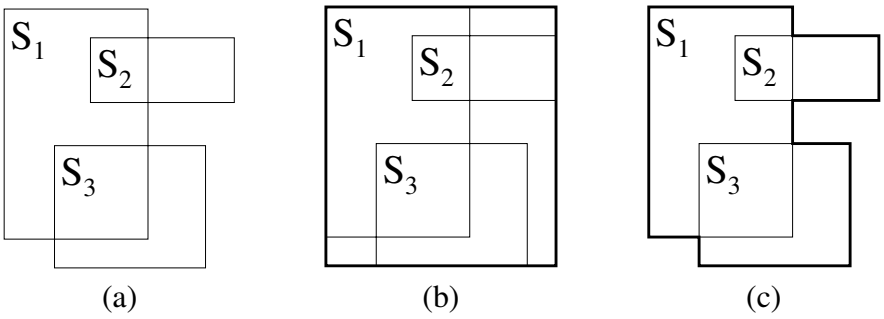


Figure 6.6: Difference between grouping subscriptions through MBR and MBP. (a) Set of subscriptions to be grouped. (b) The bold rectangle represents the MBR that encloses the set of subscriptions. (c) The bold polygon represents the MBP that encloses the set of subscriptions.

An example of the *minimum bounding polytope* (MBP) is shown in Figure 6.6. Consider a set of subscriptions shown in Figure 6.6(a). In the DR-trees the three subscriptions are grouped and the MBR is computed as the minimum bounding rectangle that includes all the subscriptions (see Figure 6.6(b)). Using Hilbert SFC to represent the subscriptions, the resulting MBP is the union of the set of segments from each subscription (see Figure 6.6(c)). We can observe that, using Hilbert SFC representation, there is no dead space, which may generate false positives. Comparing MBR and MBP to group the same set of

subscriptions, MBP may achieve better results in terms of false positives. However, identifying the set of subscriptions to accomplish an efficient merging is not trivial using Hilbert SFC representation. The algorithms used to merge similar subscriptions will be presented in the overlay organization discussion.

Another issue in this representation is the scalability for higher dimensions. The number of segments to represent a subscription considerably increases with high dimensions and large range values [76]. Moreover, the number of ranges to represent the MBPs may increase significantly at the upper levels of the trees. We propose two strategies to minimize the number of segments to be stored by the peers. The first approach is to represent a range value by a single Hilbert value (i.e., a set of coordinates is represented by the same Hilbert value) in order to find a good compromise between memory cost and routing accuracy. Another approach, which may be combined with the first, is to limit the number of segments of the subscriptions and MBPs. The ranges with the smallest distance (false positive area) are merged until the desired number of segments is reached.

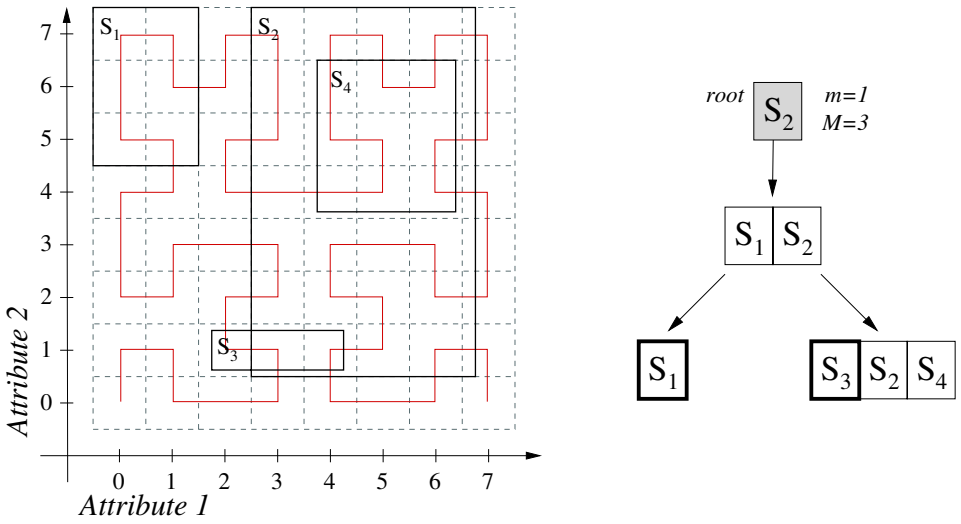


Figure 6.7: Example of splitting method in DHR-tree segment.

Overlay Organization. A joining node will be inserted in the branch for which the MBP of the last non-leaf node is the nearest (in terms of spatial proximity) with the new subscription. The new subscriber q is allocated in the children set of the node p, where the new subscription overlaps the most with the MBP of the node, i.e., the biggest range size

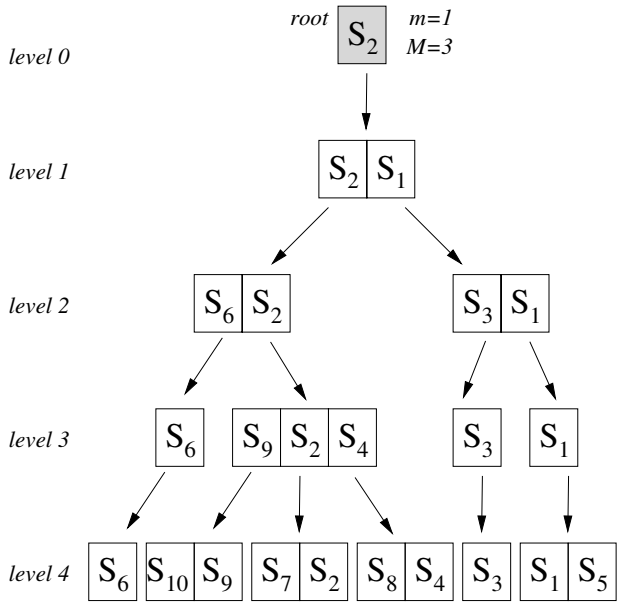


Figure 6.8: DHR-tree segment for the subscriptions of Figure 4.8.

resulted from the intersection of the MBP and the subscription. If the subscription and the MBP are disjoint, the chosen branch is the one for which the resulting MBP (i.e., the union of the subscription and the current MBP) has the same number of, or less, segments than the current MBP. If there is no node to forward the request, the chosen subtree is the one whose union of the distance between the segments is the smallest. Upon tie, the joining node is redirected to the node with the biggest MBP area, computed from the union of the ranges.

We propose two variants for splitting method. In the first strategy, two nodes (also called seed) are selected, such that the union of their MBPs has the biggest distance between their segments and they are assigned in separate groups. The remaining children are assigned to the group in the same way as in the join procedure. The new parent of the subtree is the one with the biggest MBP area in order to preserve the containment relationship. The second strategy is based on R*-trees. Instead of splitting immediately the overflowing node, some entries are reinserted in the tree to be better allocated as the join is nondeterministic. An example of the splitting strategy is shown in Figure 6.7. The left hand side of the figure illustrates four subscriptions that will split into two groups (where $m = 1$ and $M = 3$) and the right hand side the DHR-tree segment resulted from the splitting. The

subscriptions S_1 and S_3 are picked as seed and assigned to separate groups. The remaining subscriptions S_2 and S_4 are assigned to the group with S_3. Figure 6.8 presents the overlay organization for the set of subscriptions of Figure 4.8.

Event Dissemination. The event is represented in 1-dimensional space and its propagation is based on the MBP. Thus, a node forwards the event to each child whose MBP contains the event, i.e., the event falls in one of the range values. Considering the 1-to-1 transformation, i.e., one coordinate is represented by one Hilbert value; the event dissemination is equivalent to the overlays that use n-dimensional representations. However, the matching cost is much smaller considering a single dimension strategy. On the other hand, if we consider either a limited number of segments or an n-to-1 transformation (a set of coordinates represented by a single Hilbert value) the routing inaccuracy increases. In the first case, the false positives area (i.e., the distance between two merged range values) may be propagated at upper levels in the tree, and in the former case the number of false positives increases with the size of the range to be represented by a single value and the number of dimensions.

6.3 Conclusion

In this chapter we have proposed three Distributed Hilbert R-tree overlays to build content-based publish/subscribe systems: DHR-tree Center, DHR-tree Min/Max and DHR-tree Segment. The overlays mainly differ in the subscription information and the mechanism used to organize the subscriptions in the overlay. All three overlays are based on the Hilbert space filling curves, which maps an n-dimensional space into 1-dimensional space while preserving the proximity between the subscriptions. By using Hilbert space filling curves to organize the subscriptions in the overlay, similar subscriptions are allocated close to each other in the tree with simpler operations when compared to n-dimensional operations.

As aforementioned, the bases of the main operations (join and leave) in DHR-trees are the same as in DR-trees and this class of overlays relies on the self-stabilizing algorithms, presented in Section 5.3. In this way, DHR-trees are self-organizing and scalable for large number of subscribers and provide logarithmic guarantees for the overlay organization and event dissemination. The experimental evaluation of the Distributed Hilbert R-tree overlays will be presented in Chapter 7.

CHAPTER

7 Evaluation and Related Work

7.1 Introduction

In this chapter, we present the evaluation of the performance of our overlays: DR-trees (quadratic, linear and R*) and DHR-trees (center, min/max and segment). As we have discussed previously, the routing accuracy affects the efficiency of the system. We evaluate the accuracy through the number of false negatives (a subscriber does not receive a message that it is interested in) and false positives (a subscriber receives a message that it is not interested in) generated in the system. False negatives are very critical to the consistency of the system. As aforementioned, our overlays do not generate false negatives as the event dissemination is based on the MBRs/MBPs. Thus, all subscribers interested for a certain event will be notified. False positives do not affect the consistency, but a high rate may degrade the performance, consuming high bandwidth and memory.

The rest of this chapter is organized as follows. Section 7.2 presents the experimental setup used in the evaluation of our DR-tree and DHR-tree overlays. In Section 7.3, we first evaluate the performance of DR-tree overlays in terms of false positives ratio, and then the self-stabilizing algorithm for DR-trees. In addition, at the end of this section, we propose and evaluate two optimization strategies in our overlays under skewed event workloads. In Section 7.4 we evaluate the performance of DHR-tree overlays based on false positives ratio. Section 7.5 presents some related work. We finalize the chapter with a summary of the experimental evaluation in Section 7.6.

Parameter	**Values**
Number of subscriptions	$1,000$, $2,500$, $5,000$, $[10,000]$, $25,000$, $50,000$
Number of events	$[2,500]$
Subscription distribution	uniform, uniform-25:75, uniform-10:90, Zipf, [Zipf-25:75], Zipf-10:90
Event distribution	[uniform], Zipf
Dimensions	2, [3], [4], 6, 8, 10, 12
Tree degree (m, M)	$(2,5)$, $[(5,10)]$, $(10,20)$, $(15,30)$, $(20,40)$

Table 7.1: Parameters used for the experiments.

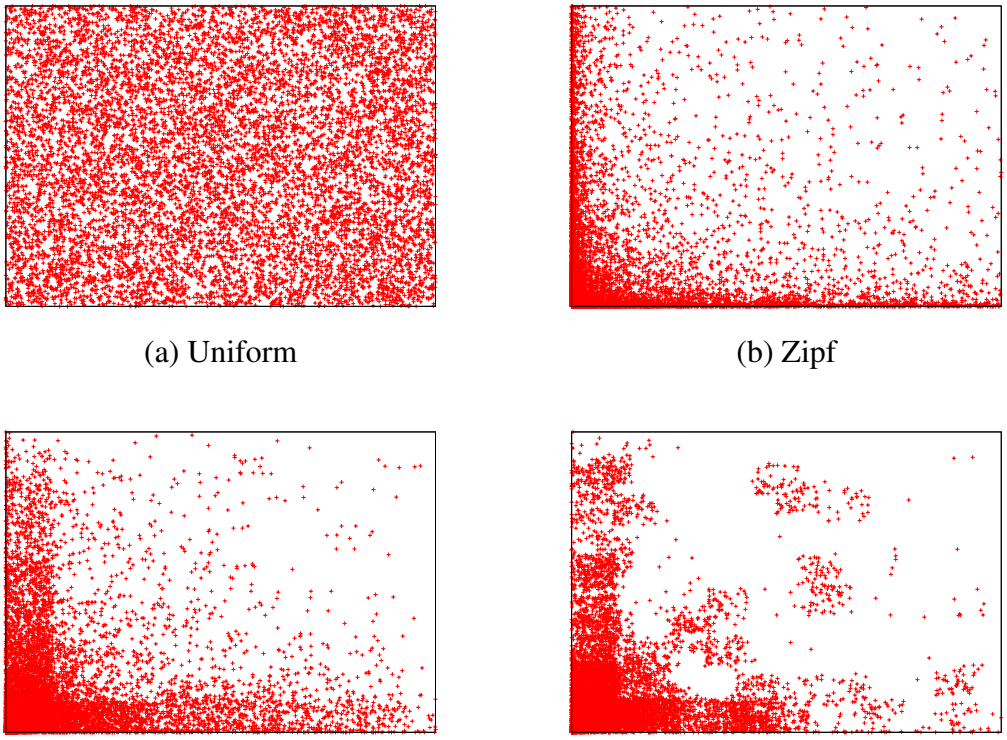
7.2 Experimental Setup

Subscriptions are defined as a set of d attribute-range pairs, each of which corresponds to a dimension. The range specifies the set of values that the subscriber is interested in. Without loss of generality, we used range values between 0 and $1,000$ for DR-trees and between 0 and $1,023$ (10 bits) for DHR-trees. Note that a range may represent a single value. Events are points in the d-dimensional space.

We analyzed the performance of the system under uniform and skewed subscription workloads; and with a uniform event distribution. Skew is simulated using a power-law distribution (Zipf with $\alpha = 1$), as is often done when evaluating publish/subscribe systems (e.g., [6, 119]), and is applied to the origin of subscriptions only: their size is always chosen according to a uniform distribution.

The subscriptions and events are composed by a set of predicates over the attributes that corresponds to the tree dimensionality. In the simulations, we considered different set of predicates with only numerical attributes.

To model and observe the influence of containment relationships, we have generated some subscription sets with a given ratio of container/containee subscriptions. Given a ratio of X:Y, we have first generated $X\%$ of the subscription population according to the current distribution. For each subscription in the remaining $Y\%$, we have taken the following steps: select a random subscription S from the current set; generate a uniform random subscription S' such that $S \supseteq S'$; and insert S' in the set. This method guarantees that at least $Y\%$ of the subscriptions are containees. We considered uniform and Zipf distributions, as well as two X:Y ratios: 25%:75% and 10%:90%. Figure 7.1 shows the distribution of the origins of the subscriptions used in the experiments for $d = 2$ (one can



(a) Uniform (b) Zipf

(c) Zipf(25:75) (d) Zipf(10:90)

Figure 7.1: Subscription distribution (only the bottom left corner of subscriptions is plotted). The abscissa/ordinate in the graphs represent the abscissa/ordinate in the event space considering range values between 0 and $1,000$.

clearly observe containment relationships in the last graph (d)).

In the experiments, we evaluated the efficiency of our approach in terms of *false positives ratio*, i.e., the percentage of the subscribers or peers in the system that receive events that do not match their interests. For simplicity, we assume that events are injected at the root. Note that this assumption is equivalent to having each event with at least one interested subscriber being produced by a publisher with a matching subscription, i.e., publishers never experience false positives locally.

Obviously, an event that does not match a single subscription is expected to show a lower false positive ratio than an event with many interested subscribers, because the latter is likely to be propagated deeper in the tree. Note that, as leaves have an MBR equal to their subscriptions and a node forwards an event to each of its children whose MBR contains the

event, only internal nodes can experience false positives. We do not consider false negatives since our structures do not produce any.

The number of events was fixed in all simulations to $2,500$ for computing false positive ratios. We have used the parameters shown in Table 7.1 (default values are in square brackets). For each simulation, we have varied the values of the parameter to be observed and fixed the remaining ones to their default values. We have fixed the default number of dimensions for DR-trees to 4 dimensions and for DHR-trees to 3 dimensions.

7.3 Distributed R-trees

This section describes the results of our evaluation and comparison of the different DR-tree variants presented earlier.

7.3.1 Overlay Structure

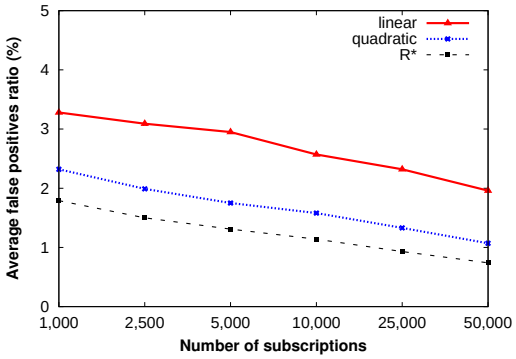


Figure 7.2: False positives ratio for different subscription set sizes.

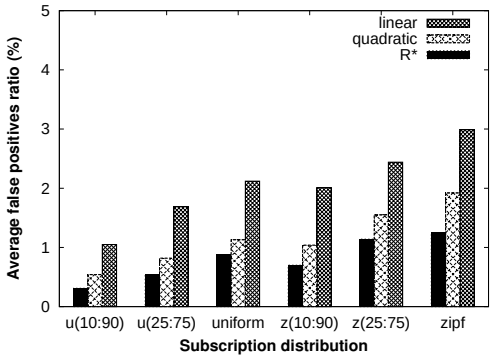


Figure 7.3: False positives ratio for different subscription distributions.

We measured the false positives ratio for different sizes of subscription sets. Figure 7.2 shows that the average false positives ratio is less than 5% and slightly decreases with the size of the subscription set. Comparing the three splitting methods, we observe that R* presents the best results because it reinserts nodes in case of overflow instead of splitting immediately. This may improve the containment relationship along the tree and, consequently, the routing accuracy because R-trees are known to be highly susceptible to the order in which entries are inserted.

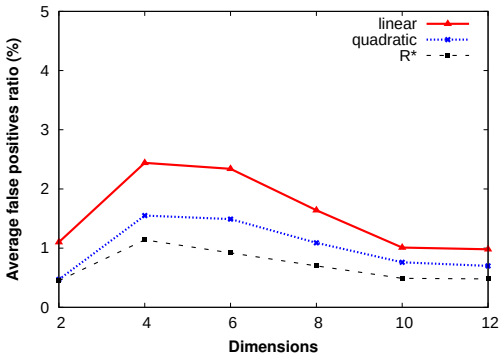


Figure 7.4: False positives ratio for different dimensions.

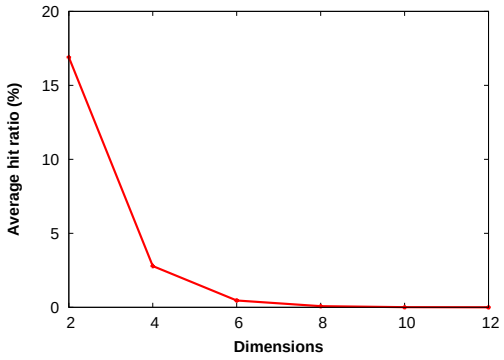


Figure 7.5: Hit ratio for different dimensions (quadratic splitting).

Figure 7.3 shows the average false positives ratio when varying the distribution of the subscriptions. We observe that better results are obtained for subscriptions with high containment relationship, which confirms that our trees do indeed preserve and take advantage of containment relationships. Accuracy is also slightly better with a uniform subscription distribution. Under a Zipf distribution, the event travels more subtrees, which explains the higher amount of false positives.

Figure 7.4 illustrates the average false positives ratio for different dimensions. Surprisingly, accuracy improves with the number of dimensions. This is due to the fact that less nodes are interested in the event, as can be seen in Figure 7.5. The hit ratio is the percentage of subscribers in the system that are interested in the event. In this way, we have performed the same experiment taking into account only events that match at least one subscription, as presented in the Figure 7.6. The average false positives ratio slightly increases though remaining less than 5%. Comparing the results shown in Figures 7.4 and 7.6, the average of false positives ratio for 4 dimensions is almost the same in both cases. This is due to the fact that most of the events, considering 4 dimensions, match at least one subscription. Also, we have plotted the average false positives ratio as a function of the number of hits for each experiment (i.e., each event disseminated) with three dimensions (2, 4 and 8). The results are shown in Figures 7.7, 7.8 and 7.9 for, respectively, R*, quadratic and linear splitting methods. We observe now that the false positives ratio actually increases with the dimension but remains reasonably small, never reaching 10% even for linear.

As discussed before, our approach differs from original R-trees in that some subscriptions may appear at the same time at the different levels in the logical tree. Thus, the degree

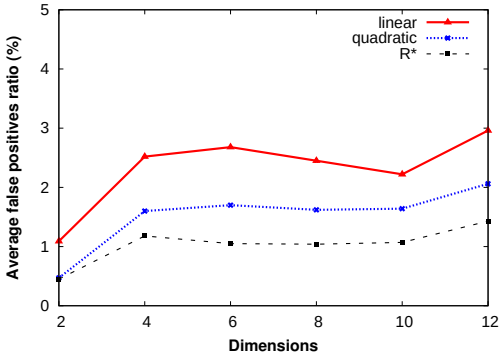


Figure 7.6: False positives ratio for different dimensions considering only events that match at least one subscription.

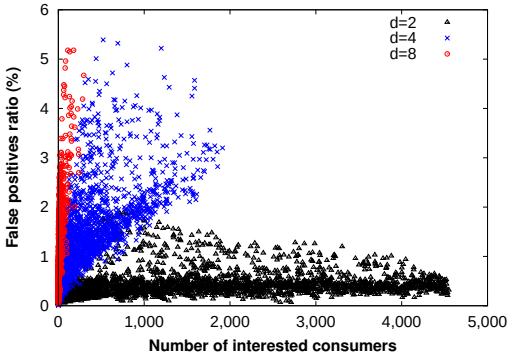


Figure 7.7: False positives ratio (R^*) vs. hit ratio for different dimensions (one point corresponds to one experiment).

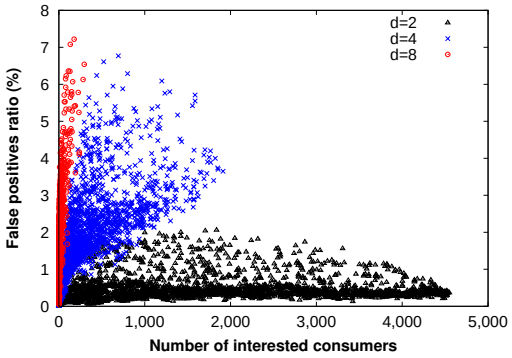


Figure 7.8: False positives ratio (quadratic) vs. hit ratio for different dimensions (one point corresponds to one experiment).

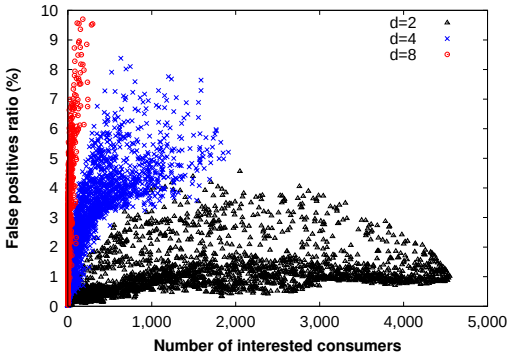


Figure 7.9: False positives ratio (linear) vs. hit ratio for different dimensions (one point corresponds to one experiment).

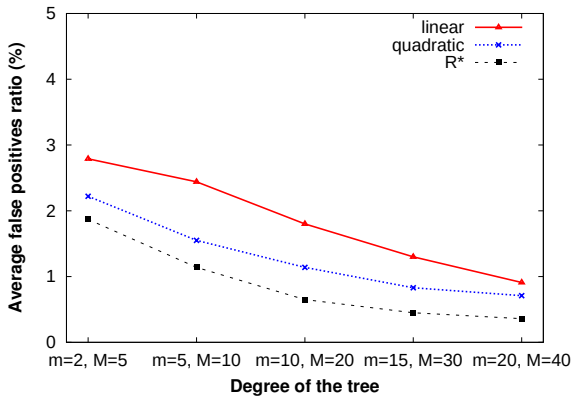


Figure 7.10: False positives ratio for different degrees of the tree.

	linear			*quadratic*			*R**		
Degree	**Max**	**Avg**	**Var**	**Max**	**Avg**	**Var**	**Max**	**Avg**	**Var**
m=2, M=5	19	4.61	5.08	20	4.59	5.09	25	3.91	8.47
m=5, M=10	28	7.93	10.66	29	7.99	10.98	30	7.49	16.27
m=10, M=20	37	14.82	23.15	51	14.92	25.40	42	15.05	32.87
m=15, M=30	50	21.69	33.52	51	21.91	35.70	57	22.28	56.29
m=20, M=40	60	28.39	54.02	68	28.02	55.38	77	30.00	81.10

Table 7.2: Degree statistics.

of a subscriber varies depending on the position and number of occurrences of its subscription in the tree, in addition to the values of m and M. Figure 7.10 presents simulation results for different degrees between $M = 5$ and $M = 40$, where $m = M/2$; and Table 7.2 shows the maximum, average, and variance of the degree of subscribers (peers) responsible for internal nodes. We observe a clear tradeoff between accuracy and subscriber's degree: increasing the degree improves accuracy. In the studied scenario, a value of $M = 20$ appears to be a good compromise.

For comparison purposes, a tree built as a direct mapping of the containment graph using the same $10,000$ subscriptions (Zipf-25:75) would have a virtual root node with approximately $2,000$ children and would obviously not be height-balanced.

7.3.2 Overlay Stabilization

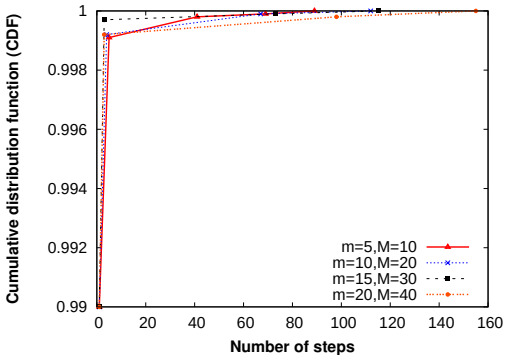


Figure 7.11: Cumulative distribution of the number of steps to stabilize the overlay (controlled departures).

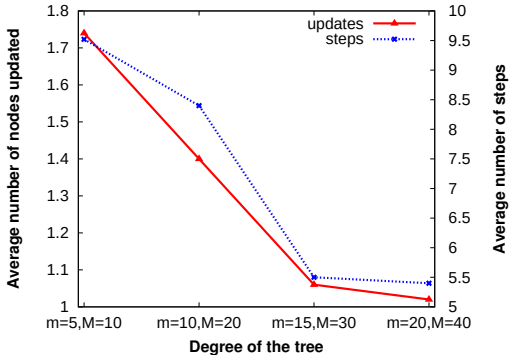


Figure 7.12: Average number of nodes updated and steps for different degrees of the tree (controlled departures).

For the evaluation of the self-stabilization algorithms, we considered the default experimental setup of Table 7.1 and the quadratic splitting method. We have evaluated the number of steps necessary to stabilize the overlay; the number of nodes updated during the stabilization process; and the number of internal nodes underloaded after uncontrolled departures and arrivals of nodes in the system. The overlay stabilization time includes the steps to check the branch of the departing node and to reinsert the nodes and update their ancestors. To compute the number of nodes updated after controlled/uncontrolled departures, we have considered the nodes reinserted in the tree, the new parent in case of reorganization, and the nodes that have updated its MBR.

Stabilization after controlled departures. In order to analyze the overlay stabilization process after controlled leaves, we performed a wide range of experiments by observing the departure of individual nodes. For each experiment, we removed $1,000$ random subscriptions (out of the set of $10,000$ subscriptions) from the system at a time. We then computed the average number of steps and of nodes updated as a result of the departures. Results were obtained by computing the average over 20 simulations.

Figure 7.11 shows the number of steps necessary to stabilize the DR-tree after each departure. For all different node's degree, more than 99% of the nodes that left the system required less than 5 steps to stabilize the DR-tree (where the height of the tree is $h = 4$).

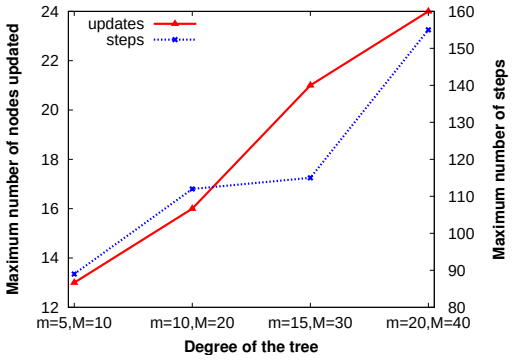


Figure 7.13: Maximum number of nodes updated and steps for different degrees of the tree (controlled departures).

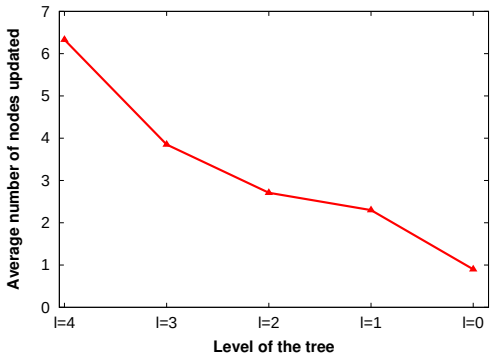


Figure 7.14: Average number of nodes updated at different levels of the tree (controlled departures).

In most of the scenarios, the children set does not become underloaded after a controlled departure, which explains this low value.

Figure 7.12 presents the average number of nodes updated and number of steps to stabilize the overlay when nodes leave the system properly as a function of the tree degree. For both experiments, we observe that better results are obtained for higher degrees. This is due to the fact that the height of the tree decreases when increasing the degree and, consequently, nodes are less frequently underloaded after a departure.

We have also plotted results considering only the maximum number of steps and updates over all simulations, as shown in Figure 7.13. We observe that, for both cases, the results increase for larger children sets as more nodes will be reinserted in the tree when nodes become underloaded. Consequently, more nodes in the tree are affected by the reinsertion of those nodes.

Finally, Figure 7.14 presents the average number of nodes updated for each level of the tree. Note, that for a tree built from a set of $10,000$ subscriptions, the height of the tree is $h = 4$. Obviously, the number of nodes updated is higher at the lower levels as the number of nodes increases.

Stabilization after uncontrolled departures. We have introduced $10,000$ uncontrolled node departures (chosen randomly among the initial set of subscriptions) and $10,000$ additional node arrivals to evaluate stabilization under churn. We considered a Poisson model

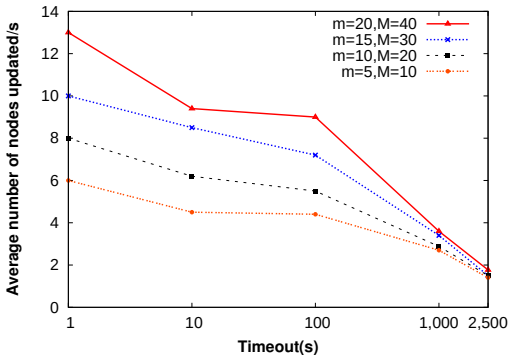


Figure 7.15: Average number of nodes updated per second for different timeout values.

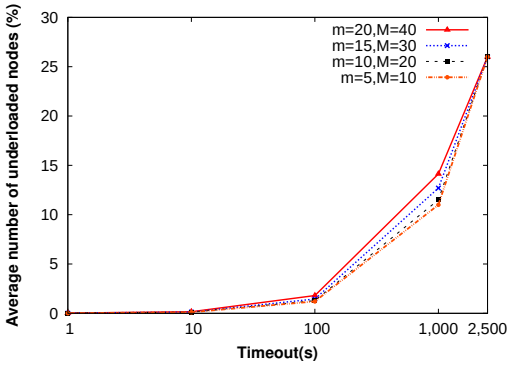


Figure 7.16: Underloaded nodes ratio for different timeout values.

for node arrivals and uncontrolled departures [72] with rate $\lambda = 1s$ and we varied the timeout interval Δ of the stabilization modules: $1s$, $10s$, $100s$, $1,000s$, and $2,500s$. This means that, during each stabilization interval, there are on expectation Δ departures and Δ arrivals. For all the simulations, events to check the correctness of the tree were triggered by the leaves and propagated upwards the tree (this is equivalent to having every node locally triggered the events without propagation).

Figure 7.15 presents the average number of nodes updated per second when varying the timeout. Unsurprisingly, we observe that more nodes need to be updated when increasing the degree. Indeed, as the children set grows, more nodes must be reinserted in the tree when a node becomes underloaded. Note that, upon arrivals of new subscriptions, nodes that have corrupted variables due to uncontrolled departures are updated before the timeout.

Figure 7.16 illustrates the average ratio of underloaded nodes for different timeout values. Considering the initial setup of $10,000$ subscriptions, we built a DR-tree for each degree configuration. The resulting tree had the following number of internal nodes: $1,670$ ($m = 5, M = 10$), 783 ($m = 10, M = 20$), 524 ($m = 15, M = 30$), 382 ($m = 20, M = 40$). The ratio of underloaded nodes is computed as a percentage of these values. As expected, we observe better results for small timeouts because the structure is corrected more frequently. The results do not show a significant difference for varying node degrees, because there is a correlation between the number of underloaded nodes and the number of internal nodes in the tree.

We have finally analyzed the evolution of the minimum node degree in the tree over

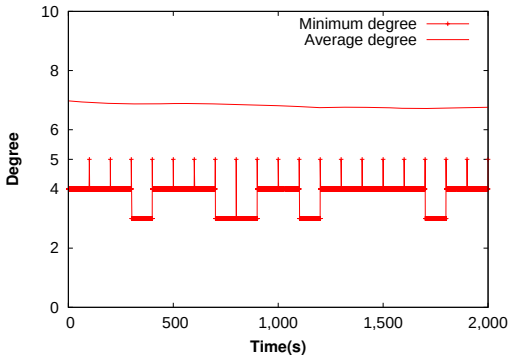


Figure 7.17: Minimum children set in the tree for timeout $100s$ ($m = 5, M = 10$).

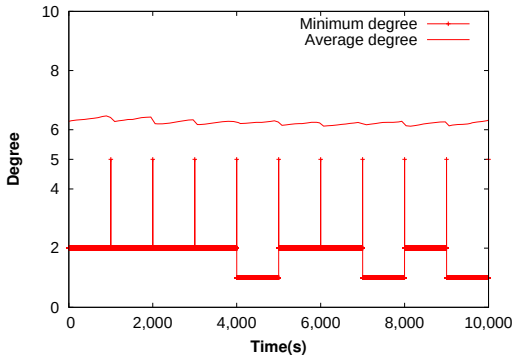


Figure 7.18: Minimum children set in the tree for timeout $1,000s$ ($m = 5, M = 10$).

time to verify that self-stabilization works as expected under churn. We can observe in Figure 7.17, for timeout $100s$, that the average degree is approximately 7 and the minimum children set before correction of the tree varies between 4 and 3. However, after every timeout, the structure is corrected and the minimum degree reverts to $m = 5$. We also tested for timeout $1,000s$, as shown in Figure 7.18. In this scenario, the average degree is above 6 and the minimum children set before the execution of the self-stabilization modules varies between 2 and 1. Again, we can observe that the structure is corrected with minimum degree $m = 5$ after every timeout.

7.3.3 Overlay Optimization

Under biased event workloads, it may happen that the organization of the DR-tree, based on minimizing the MBR coverage, may perform poorly because small "false positive" regions are hit by many events while larger areas see none. To deal with such situations, we have introduced two optimization mechanisms for minimizing the number of false positives.

Static. The first optimization takes place when inserting a new subscription in the tree. The choice of which branch the new subscription will be inserted is no longer only based on the MBR coverage, but also on the event history that each node keeps track of. We choose the branch for which the MBR enlargement covers the smallest number of events (instead of the smallest area) in order to minimize the total number of false positives in the structure. Upon tie, we revert to the criterion of the smallest MBR enlargement.

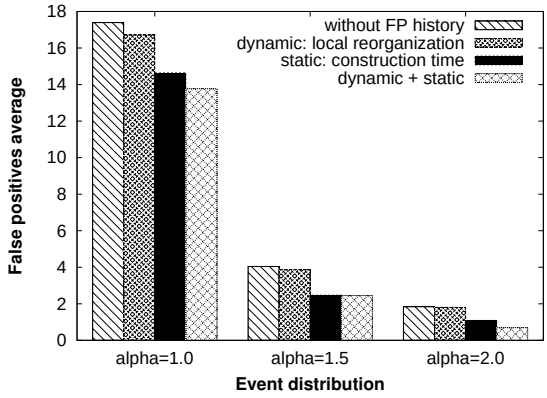


Figure 7.19: False positive average for different event distribution.

Dynamic. The second mechanism is based on local reorganizations of the DR-tree, where parent election is performed according to the number of hits (event that matches a subscription, or "true positives") instead of the largest MBR coverage. Each node computes locally its number of hits and false positives. When a child experiences significantly more hits than its parent, the former replaces the latter in the hierarchy.

In order to evaluate these two strategies, we have measured the false positives average for different event distribution, i.e., we have generated the events according to the Zipf distribution varying the value of α (see Figure 7.19). For the evaluation, we considered the default experimental setup of Table 7.1 and quadratic splitting method. We observe that the combination of static (construction time) and dynamic (local reorganization of the DR-tree) mechanisms performs better for skewed event distribution than the DR-tree built without false positives history. Moreover, the static mechanism alone presents better results than the dynamic one. This shows that one can already significantly improve the routing accuracy under bias event workload with just a simple modification to the node insertion algorithm.

7.4 Distributed Hilbert R-trees

As in the DR-trees, we also evaluate the routing accuracy through the number of false positives generated in the overlays. We have compared the three different approaches: DHR-tree Min/Max, DHR-tree Center and DHR-tree Segment. For the DHR-tree Segment,

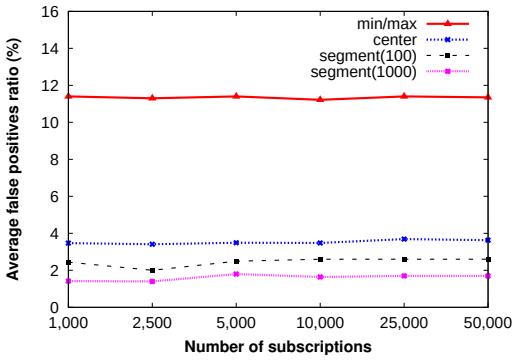


Figure 7.20: False positives ratio for different subscription set sizes.

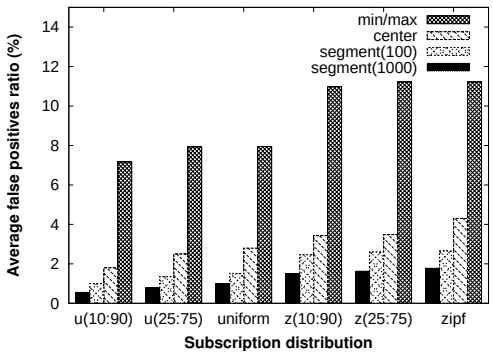


Figure 7.21: False positives ratio for different subscription distributions.

for most cases, we have evaluated the overlay considering a limited number of segments (i.e., number of discrete ranges, each of which represents a segment of the curve) of the subscriptions and MBPs. We have considered two values, 100 and $1,000$ segments.

First we have measured the routing accuracy for different subscription sets, as shown in Figure 7.20. We observe that DHR-tree Segment presents better results than DHR-tree Center and DHR-tree Min/Max. Obviously, minimizing the number of segments to represent the data structures increases inaccuracy, what explains the higher average false positives ratio for the DHR-tree Segment with 100 segment set size compared with $1,000$ segments. Unsurprisingly, DHR-tree Min/Max presents high average of false positives ratio compared with the other approaches. This is due to the fact that the event dissemination is based only on the min/max Hilbert values of the MBR. Thus, minimizing the memory cost to index the MBRs increases the inaccuracy during the event dissemination. For each overlay, the average false positives ratio remains nearly the same even if the number of subscriptions increases. This is because using Hilbert space filling curves, the containment relationship is not preserved efficiently, i.e., subscriptions with containment relationship may not be allocated in the same subtree.

Figure 7.21 shows the average false positives ratio when varying the subscription distribution. As in DR-trees, better results are obtained for subscriptions with high containment relationship (10:90) and under uniform distribution, which confirms that DHR-tree overlays also preserve the containment relationship. As in the previous experiment, the DHR-tree Segment with $1,000$ segments presents better results.

We have performed simulations varying the degree of the nodes between $M = 5$ and

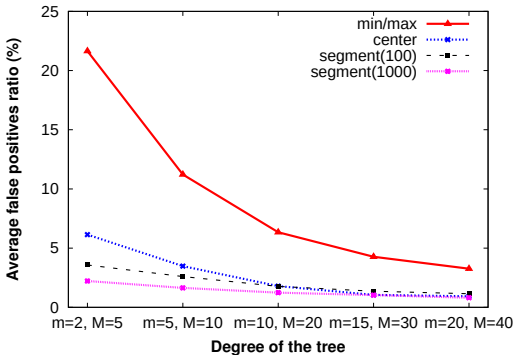


Figure 7.22: False positives ratio for different degrees of the tree.

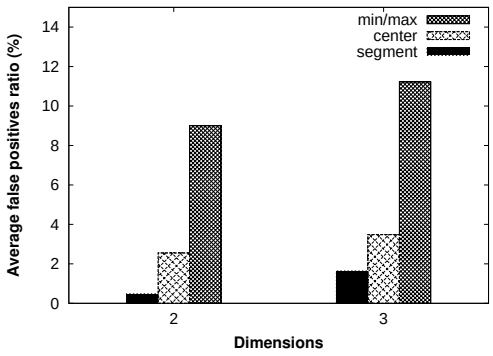


Figure 7.23: False positives ratio for different dimensions.

$M = 40$ (remember that $m = M/2$). Figure 7.22 illustrates the average false positives ratio for the different values of m and M. As can be observed, varying the degree in the DHR-tree Min/Max overlay affects the routing accuracy significantly. Considering a value of $M > 20$ for DHR-tree Min/Max, the routing accuracy improves significantly, almost reaching 5%. In contrast to DHR-tree Min/Max, the results from the DHR-tree Segment (100 and $1,000$) variant remain almost constant. Even though the event travels fewer subtrees, the routing inaccuracy increases as the number of segments to represent the MBPs is limited.

We have also plotted the average number of false positives for 2 and 3 dimensions in Figure 7.23. For DHR-tree Segment, we have fixed the number of segments to 500 for 2 dimensions and $1,000$ for 3 dimensions. We can see that the average slightly increases but remains less than 5% for the DHR-tree Center and the DHR-tree Segment variants.

In order to compare the tradeoff between the memory cost to store the segments in the DHR-tree Segment overlay and the routing accuracy, we have performed simulations varying the number of segments. Figures 7.24 and 7.25 present the average false positives ratio varying the segment set size, respectively, for 2 and 3 dimensions. For 2 dimensions, the results are roughly the same while for 3 dimensions, the ratio increases with smaller numbers of segments. Fixing 50 segments to represent the subscriptions and MBPs, the ratio remains very low in both experiments and the memory cost to index remains low.

As presented in the Section 4.3, we may consider a set of coordinates in the Euclidean space to be represented by a single Hilbert value in order to minimize the memory cost. Considering the range values between 0 and $1,023$ we have varied the order of the Hilbert

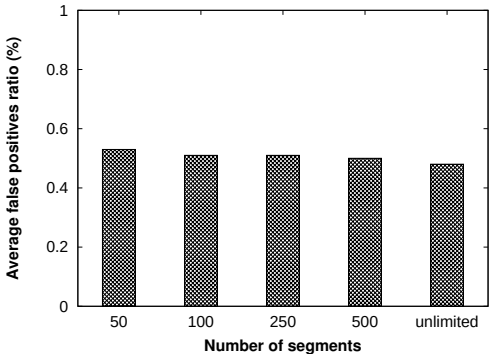


Figure 7.24: False positives ratio varying the number of segments (2 dimensions).

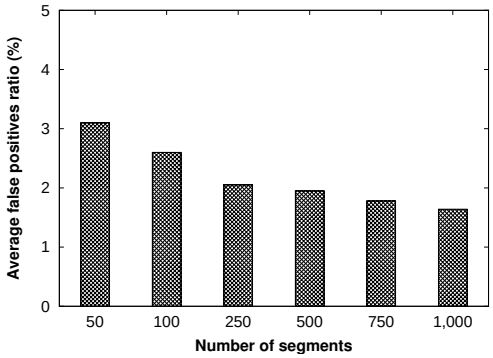


Figure 7.25: False positives ratio varying the number of segments (3 dimensions).

Figure 7.26: False positives ratio for different order of Hilbert curve (2 dimensions).

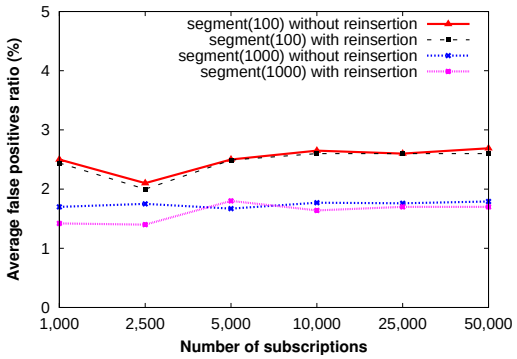


Figure 7.27: False positives ratio for different splitting method (3 dimensions).

curve used in the transformation. This means that considering a low order of Hilbert curve implies in a larger range of values to be covered by the same Hilbert value. Figure 7.26 illustrates the routing accuracy by varying the order (k) of Hilbert curve between 2 and 10. Note that we have performed the experiments for 2 dimensions and we consider unlimited number of segments for the subscriptions and MBPs. Obviously, the number of false positives is directly related to the granularity of the Hilbert curve, i.e., space partitioning. We observe that $k = 4$ (Hilbert values from 0 to 255) is a good compromise as the average remains less than 5% and the number of segments per subscription/MBP is reasonably low. Note that the events were mapped using the same order of the Hilbert curve as for the subscriptions.

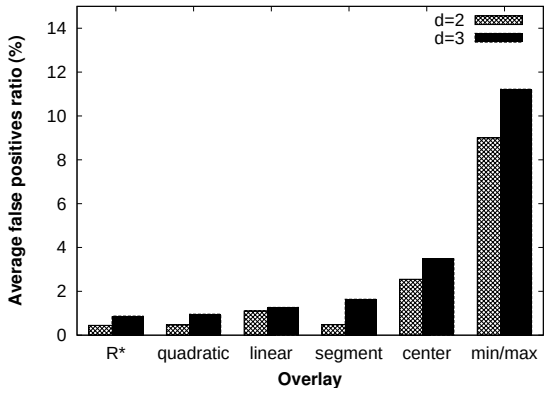


Figure 7.28: False positives ratio for different overlays.

Finally, we have compared the two different splitting methods of the DHR-tree Segment: with or without reinsertion of nodes. Figure 7.27 illustrates the results for 100 and $1,000$ segment set sizes. Both set sizes show only little improvement when performing reinsertion during the splitting method, which does not justify the cost of reinsertion.

In order to compare, DR-tree and DHR-tree overlays, we have performed simulations with $10,000$ subscriptions and range values between 0 and $1,023$ for 2 and 3 dimensions. As in the previous experiment, we have fixed the number of segments, in the DHR-tree Segment variant, to 500 for 2 dimensions and $1,000$ for 3 dimensions. Results are presented in Figure 7.28. As expected, DR-trees present in general better results than DHR-trees. Moreover, R* outperforms all other variants, which confirms the tradeoff between routing accuracy and memory cost to index the data structures. Except for the DHR-tree Min/Max,

all variants present an average false positives ratio less than 5%. For the 2 dimensions experiment, the DHR-tree Segment variant shows better results than the linear. In addition, the results from DHR-tree Segment show that our merging algorithm is efficient at maintaining a low false positives ratio. The DHR-tree Center variant presents higher average compared with DHR-tree Segment and DR-trees. The reason is that the insertion and splitting are based only on the Hilbert value of the centroid of the subscriptions, which may violate the containment relationship properties.

7.5 Related Work

Publish/subscribe over structured peer-to-peer systems has been widely addressed in recent years.

TERA [8] is a topic-based publish/system built on top of peer-to-peer systems. This approach clusters the subscribers with similar interests in order to confine the traffic. Many other topic-based approaches have been proposed e.g., Scribe [23] and Bayeux [126]. These publish/subscribe systems leverage DHT, Pastry [92] and Tapestry [120] respectively, in order to build scalable and fault-tolerant publish/subscribe systems. The advantage of using DHTs is to build scalable, self-organizing and fault-tolerant publish/subscribe systems. However, in contrast to content-based systems, topic-based systems lack on expressiveness to specify the subscriptions.

Tam et al. [105] extend Scribe to support content-based systems by organizing the content into several topics. For each subscription and for each event, they build a set of topics and submit to Scribe. This approach does not fully provide the query semantics of a traditional content-based approach limiting the expressiveness and flexibility of the subscriptions. Moreover, false positives are likely to occur.

We focus on content-based publish/subscribe approaches built on top of structured peer-to-peer systems. One of the techniques used in some of these approaches is the *rendezvous-based routing* (e.g., [3, 4, 11, 27, 84, 125]). The main principle of this technique is to identify particular nodes where subscriptions meet publications. Once the event meets the node responsible for managing the dissemination structure for this event, the event delivery starts from this node as the root of a spanning tree reaching all the interested subscribers. The main motivation of this approach is that a controlled subscription distribution allows better load balance in terms of subscription storage and management (insertion, removal and

update), thus avoiding redundant matching operations to be performed by several nodes. The main drawback of such solutions is the high load reported on the *rendezvous* nodes since they centralize all the filtering performed in the system. Hence, rendezvous nodes may become bottlenecks and single point of failures. In contrast, a totally distributed (i.e., decentralized) approach, where every peer in the system participates in the matching and event dissemination, spreads the load in the system.

Willow [91] uses a virtual tree, where each leaf in the tree is associated to a set of attributes and values. The internal nodes are responsible for the attributes of their children. The routing procedure leverages the DHT, the event is sent to the node with the identifier nearest to a specified key in terms of an XOR metric. Similarly, Wang et al. [110] propose to use a spanning tree in which each leaf is responsible for an identifier range and the root is in charge of the entire identifier space. The event forwarding is similar to prefix-forwarding, but in addition, for each entry in the routing table, there is a filter associated to the identifier. Also, covering relation is considered to reduce the number of subscriptions stored by each peer. These approaches have the disadvantage that the notification starts at the root of the spanning tree, thus depending on the publication rate, the root node may become overloaded.

HyperCBR [22] relies on a multidimensional space topology (simulations are done using a CAN-based implementation). The space is divided into multiple partitions under responsibility of nodes. In a two dimensional space, subscriptions are propagated along the entire column, thus distributed among different nodes. The events are disseminated along a row and if it intersects a matching subscription, the event will be disseminated along the respective column. The main drawback of this approach is that it can lead to a high amount of false positives.

HOMED [29] and Mirinae [28] use a peer-to-peer hypercube-based overlay. Peers are organized based on their interests in order that only interested peers participate in the dissemination of an event. Similarly, Meghdoot [58] uses the CAN infrastructure. In this system, the subscriptions composed by multiple predicates are partitioned and distributed onto CAN nodes. These approaches have two main drawbacks: the lack of scalability for publish/subscribe systems that require complex subscriptions and the large number of false positives/negatives. The first problem was addressed by using multi-dimensional spaces. Terpstra et al. [106] propose to partition the event space among the peers in the system, but they broadcast the events and the subscriptions to all the peers in the system. Consequently,

the number of false positives is in the order of the number of subscriptions in the system.

Other approaches have been proposed to improve subscription representation on DHT-based systems, more precisely, to support range predicates. Chord is used in [108] to build a content-based publish/subscribe system. The values of each attribute are hashed and stored at the corresponding Chord node. The matching procedure is done for each attribute separately. The main disadvantage of this approach is that the number of nodes that will store a subscription is proportional to the number of attributes in the system and to the interval of the range value for each attribute. Hence, the total number of subscriptions in the system can be critical to the performance, generating high traffic in the network and high matching rate.

One of the techniques that focus on minimizing the false positives/false negatives is the organization of subscribers based on their similarity [6, 25, 109, 118]. In the first two systems, subscriptions form unbalanced trees and the publication complexity is strongly dependent on the subscription distribution. Contrary to these approaches, our structures are balanced and offer logarithmic guarantees (subscription, unsubscription, event dissemination).

One approach that tries to balance the workload among the nodes is proposed in [118]. A hierarchical tree is used to partition the subscriptions based on the attributes or their values. Similar subscriptions are grouped at the same branch. The matching tree is divided into a set of subtrees distributed among the peers in the system. Another similar approach is presented by [117], where the content space is partitioned and mapped to a k-d tree. The node responsible for a partition will store all the subscriptions that overlap the partition. In case a subscription overlaps n partitions, the subscription will be split into n sub-subscriptions and stored each of them in the node responsible for the corresponding partition. The main drawback of these approaches is that skewed subscription distribution may load some peers with high amount of subscriptions.

Sub-2-Sub [109] is constructed on top of an epidemic semantic-based group membership. Nodes with similar interests are organized in a gossip-based overlay. For the event dissemination, an event is disseminated through a probabilistic balanced tree to all interested parties. The main drawback of this approach is that depending on the subscription distribution, it can lead to nodes having high degree. In contrast, our approaches provide bounded degree per node in the tree.

Another approach is the subscription merging [37, 71], which also groups subscriptions

based on their similarity and creates a new subscription containing the set. This new subscription is similar to the MBR and it is used in the matching procedure to determine if the event must be disseminated in the group. The merging algorithm used to identify the groups impacts on the number of false positives generated in the system. However, the merging problem was proved to be NP-hard [37]. An optimization of subscription merging is presented by Ouksel *et al.* [83] where they propose a Monte Carlo type algorithm. Contrary to our approach, the algorithm introduces false negatives due to the probabilistic nature of the algorithm.

Similar to our overlays, Wang et al. [112, 113] propose a broker-based publish/subscribe based on R-trees. Subscriptions and events are represented geometrically, i.e., events as points in a d-dimensional space and subscriptions as hyper-rectangles. This representation is more flexible than the previous ones and captures well the semantics of the subscriptions. They propose two routing approaches: event space partitioning (ESP) and filter set partitioning (FST). In the ESP, the d-dimensional space is partitioned into n disjoint subspaces, where each server is responsible for one subspace. Thus, a subscription that intersects the subspace is stored in the corresponding server. The main disadvantage is that if there is a high overlap of the subscriptions in multiple subspaces, the subscription is replicated in multiple servers, thus increasing the total of subscriptions in the network also the traffic generated by the event forwarding. The second approach, FST, assigns each subscription to a single server and similar subscriptions are grouped together in the same server. For this approach they suggest to use R-trees assigning the subscriptions at leaf nodes and the root has as many children as servers in the system. Thus, each subtree is under responsibility of one server and uses the MBRs to filter the event. Conversely to our overlays, this approach is not appropriated to large-scale systems being limited to a fixed set of subscriptions.

Publish/subscribe over unstructured peer-to-peer overlays is only marginally addressed and with little success only. This is due to the fact that unstructured peer-to-peer overlays do not rely on a structure that would facilitate the routing since they use inefficient routing strategies. Costa et al. [35] propose a hybrid approach, which uses deterministic and/or probabilistic event routing. In case the subscription information is available, the algorithm uses deterministic event routing by routing the event along the link that transmitted the matching subscription. If no subscription information is available, the events are forwarded to a randomly chosen subset of the available links. They propose that the links of the connectivity graph can be easily built as an unstructured peer-to-peer overlay.

Another approach based on unstructured peer-to-peer overlays is presented in [10]. The main idea is to build event distribution lists via subscription flooding. The active subscriptions are periodically broadcast in the network in order to refresh the subscription information. Thus, the subscriptions of faulty nodes will no more remain, being removed after a time-out.

The main drawback of these approaches relies on the main characteristics of unstructured peer-to-peer overlays, the uncontrolled organization of the nodes in the system. Thus, subscribers with similar interests may be located logically far away from each other in the network. As a consequence, these approaches may generate high message overhead during event dissemination. Moreover, non-deterministic approaches, as presented in [35], can lead to unsuccessful event deliveries if the TTL reaches zero before the event reaches all the interested subscribers.

7.6 Conclusion

We have implemented and analyzed the performance of our DR-tree and DHR-tree overlays. We have evaluated the performance in terms of false positives and false negatives. By construction of the MBRs, the R-tree structure does not produce false negatives during dissemination, i.e., all the subscribers that have subscribed for an event will receive the event.

Except to DHR-tree Min/Max overlay, our experiments show that the false positives rate is in the order of $2 - 4\%$ with most workloads. Overall, DR-trees present better results than DHR-trees since the n-dimensional MBRs are more appropriate in order to preserve the containment relationship between the subscriptions. On the other hand, using Hilbert space filling curves may reduce the computation costs during the overlay organization and the matching procedure. In addition to that, the cost in space to index the data structures.

We have also evaluated the effectiveness of our self-stabilizing algorithms and proposed two optimization strategies for the DR-tree overlays considering skewed event workloads. Note, that the optimizations can be easily adapted to the DHR-tree Center overlay.

At the end of this chapter, we have presented some related work focusing on content-based publish/subscribe systems built on top of structured peer-to-peer systems. We have presented the main characteristics of these systems and compared to our overlays.

Conclusion

CHAPTER 8

Conclusion

In large scale distributed environments, huge amounts of information are exchanged and accessed by a large and dynamic number of participants. In distributed settings, the information may not be distributed uniformly and may have different request popularity, with some information being accessed frequently while other never being accessed. This may cause bottlenecks and overload some nodes responsible for storing the information and forwarding the request. Another challenge in distributed environments is to efficiently disseminate information. For efficient information dissemination, it is important to disseminate useful information to the users taking into account their interests. Diffusion of information that is not interesting to the users consumes bandwidth and may overload some nodes responsible to route the information. In this thesis, we focused on developing efficient load balancing mechanisms when the information is requested with different popularity and efficient information dissemination considering different user's interests.

8.1 Contributions

The contributions of this thesis can be summarized as follows:

Balancing the Communication Load in Structured Overlays. Several approaches have been proposed to balance the communication load in peer-to-peer systems. However, most of these approaches focus on the uneven distribution of the objects and nodes in the overlay. Few of them aim to balance the traffic resulting from the skewed popularity of the objects. First, we have studied the impact, through simulations, of the biased popularity of the objects. We have shown that with a Zipf-like distribution of the requests in the system, the

load at the nodes responsible for the requested information or object and the routing load along the lookup paths also exhibits a Zipf-like distribution. The routing load at the lookup path is very intense at the nodes close to the destination and decreases with the hop distance from the destination node.

We have proposed to balance the routing and request load through, respectively, dynamic routing table reorganization and adaptive caching. We dynamically reorganize the ”long range neighbors” in the routing tables of the nodes that have a high request load in order to reduce the routing load, thus increasing the routing load of the nodes that do not hold popular objects. As this mechanism only balances the routing load of the nodes in the system, the request traffic at the nodes that hold popular objects remains high. For that reason, we proposed caching as complementary solution, where the most popular objects are cached along the lookup path. Our approach is easy to deploy and can be applied to any DHT that provides flexibility on the neighbor selection to build the routing tables. We have shown that our routing table reorganization significantly improves the distribution of the traffic in the system. The combined caching approach aims to achieve a global traffic load balancing. We have demonstrated that the cache does not require much storage capacity to be effective once the most popular objects may remain permanently in the cache. Extra messages are introduced in the system by our caching algorithm but the amount is negligible if we compare with the number of requests issued. In the studied scenarii, we have shown that using routing table reorganization approach combined with caching most of the nodes have the same load. Moreover, depending on the degree of the popularity of the objects (e.g., requests follow a Zipf-like distribution with $\alpha = 0.5$) the routing table reorganization is satisfactory to balance the trafic load, being unnecessary to complement with caching.

Content-based Publish/Subscribe Overlays. It is not trivial to create and maintain an efficient peer-to-peer overlay for selective content dissemination. Most approaches suffer from loss of routing accuracy (resulting in false negatives and false positives) and poor latency during the event dissemination.

We have proposed two classes of overlays, Distributed R-trees (DR-trees) and Distributed Hilbert R-trees (DHR-trees), which are a class of content-based publish/subscribe where publishers and subscribers are organized in a peer-to-peer network. Our overlays extend the R-tree structure and its variants, where subscribers are organized in a decentralized virtual tree based on their interests (containment relationship) in order to quickly

disseminate the events and minimize the message overhead. Moreover, we have shown that our overlays do not produce false negatives and maintain a low rate of false positives.

We have shown that join and control departures do not corrupt the structures. We have implemented self-stabilizing algorithms that guarantee correctness despite failures and changes in the peer population. Our overlays provide logarithmic guarantees for the publish/subscribe operations (subscription, unsubscription and event dissemination).

The main difference between the two approaches lies in the data structure that each peer stores. With the DR-tree approach, the subscriptions are represented geometrically in a n-dimensional Euclidean space. In the DHR-tree approach, we have proposed two subscription representations. In the first representation, subscriptions are represented by an n-dimensional space associated with Hilbert values. Another representation maps the subscriptions in a 1-dimensional space using Hilbert SFC. As we have presented, Hilbert Space Filling Curves (Hilbert SFC) preserves best the proximity compared to other space filling curves.

Our experiments have shown that our overlays exhibit a low rate of false positives. DR-trees outperform DHR-trees in terms of false positives, more precisely, R*-trees presented the best results. This is explained by the fact that using MBRs to group similar subscriptions tends to preserve more efficiently the containment relationship than using Hilbert space filling curves. Moreover, we have shown that there is a tradeoff between memory cost for indexing data structures and routing accuracy. DHR-tree Center and DHR-tree Min/Max provide much less complex overlay organizations (join and leave procedures) if compared to the other overlays. This is due to the fact that the overlay organization relies only on one/two Hilbert value(s) facilitating the criteria for the join and leave procedures. However, this approach does not preserve the containment relationship among the nodes as efficient as DR-trees.

8.2 Discussions and Future Directions

Balancing the Communication Load in Structured Overlays. As previously mentioned, our load balancing solution can be adapted to any system that provides neighbor selection flexibility such as Pastry [92]. However, not all structured peer-to-peer systems provide such flexibility for neighbor selection when building the routing tables. There are several directions for future work in our load balancing strategy. One direction is to

extend our protocol to systems that provide route selection flexibility and not neighbor selection flexibility such as CAN [89]. The load information remains stored in the routing tables, however the nodes may choose the next hop with the lowest load. The tradeoff of this approach is that the lookup path may become longer since the least loaded node is not necessarily the closest to the destination. Another direction is to provide a more generic solution adaptable to other peer-to-peer systems taking into account different characteristics of the overlay (e.g. node's degree).

Content-based Publish/Subscribe Overlays. The implementation of our publish/subscribe overlays has been evaluated by the means of simulations. We have shown that our overlays maintain a low false positive ratio when using synthetic subscriptions and events with different distributions. We have proved that our approach is efficient in theory, however we could imagine to test with real data. One of the interesting directions is to evaluate our overlays in terms of false positives using the data gathered from real life systems (e.g., Yahoo! Alerts).

Another interesting issue is to consider the subscriber capacities to organize the overlay rather than only relying on the subscription properties (containment relationship). In environments where the resources are scarce (CPU, memory and bandwidth), the decision of electing a new parent for a subtree based on the node's capacities may be crucial to avoid degradation of performance rather than using only containment relationship. The implication of this strategy is that the number of false positives in the system may increase as the subscription of the new root may not cover the subscriptions of its children.

Bibliography

[1] K. Aberer, A. Datta, and M. Hauswirth. Multifaceted simultaneous load balancing in DHT-based P2P systems: A new game with old balls and bins. In *Self-* Properties in Complex Information Systems, Hot Topics Series, Lecture Notes in Computer Science 3460, Springer Verlag*, pages 373–391, 2005.

[2] K. Aberer, M. Hauswirth, M. Punceva, and R. Schmidt. Improving data access in P2P systems. *IEEE Internet Computing*, 6(1):58–67, 2002.

[3] I. Aekaterinidis and P. Triantafillou. Pastrystrings: A comprehensive content-based publish/subscribe DHT network. In *Proceedings of International Conference on Distributed Computing Systems*, page 23, 2006.

[4] J.P. Ahulló, P.G. López, and A.F.G. Skarmeta. CAPS: content-based publish/subscribe services for peer-to-peer systems. In *Proceedings of International Conference on Distributed Event-Based Systems*, 2008.

[5] M. Altinel and M.J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of International Conference on Very Large Databases*, pages 53–64, 2000.

[6] E. Anceaume, M. Gradinariu, A.K. Datta, G. Simon, and A. Virgillito. A semantic overlay for self-* peer to peer publish/subscribe. In *Proceedings of International Conference on Distributed Computing Systems*, page 22, 2006.

[7] R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito. On the modelling of publish/subscribe communication systems. *Concurrency and Computation: Practice and Experience*, 17(12):1471–1495, 2005.

[8] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni. TERA: topic-based event routing for peer-to-peer architectures. In *Proceedings of International Conference on Distributed Event-based Systems*, pages 2–13, 2007.

[9] R. Baldoni, M. Contenti, and A. Virgillito. The evolution of publish/subscribe communication systems. *Future Directions of Distributed Computing*, 2584:137–141, 2003.

[10] R. Baldoni, G. Cortese, F. Morabito, L. Querzoni, S. Tucci Piergiovanni, and A. Virgillito. On the accuracy of event distribution lists for publish/subscribe in dynamic distributed systems. In *Proceedings of International Conference on Distributed Computing Systems*, page 31, 2006.

[11] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Proceedings of International Conference on Distributed Computing Systems*, pages 437–446, 2005.

[12] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, MIDLAB 1/2006 - Dipartimento di Informatica e Sistemistica, Università di Roma la Sapienza, 2006.

[13] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of International Conference on Distributed Computing Systems*, pages 262–272, 1999.

[14] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD*, pages 322–331, 1990.

[15] S. Bianchi, A.K. Datta, P. Felber, and M. Gradinariu. Stabilizing peer-to-peer spatial filters. In *Proceedings of International Conference on Distributed Computing Systems*, page 27, 2007.

[16] S. Bianchi, P. Felber, and M. Gradinariu. Content-based publish/subscribe using distributed R-trees. In *Proceedings of International Conference on Parallel and Distributed Computing*, pages 537–548, 2007.

[17] S. Bianchi, S. Serbu, P. Felber, and P. Kropf. Adaptive load balancing for DHT lookups. In *Proceedings of International Conference on Computer Communications and Networks*, pages 411–418, 2006.

[18] M. Bienkowski, M. Korzeniowski, and F. Meyer auf der Heide. Dynamic load balancing in distributed hash tables. In *Proceedings of İnternational Workshop on Peer-to-Peer Systems*, pages 217–225, 2005.

[19] L. Breslau, P. Cao, G. Phillips L. Fan, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom*, pages 126–134, 1999.

[20] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Proceedings of International Workshop on Peer-to-Peer Systems*, pages 80–87, 2003.

[21] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[22] S. Castelli, P. Costa, and G.P. Picco. HyperCBR: Large-scale content-based routing in a multidimensional space. In *Proceedings of IEEE Infocom*, 2008.

[23] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, 2002.

[24] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of International Conference on Very Large Databases*, pages 826–837, 2002.

[25] R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Proceedings of Euro-Par*, pages 1194–1204, 2005.

[26] R. Chand and P. Felber. Scalable distribution of xml content with XNet. *IEEE Transactions on Parallel and Distributed Systems*, 19(4):447–461, 2008.

[27] S.-C. Lo Y.-T. Chiu. Design of content-based publish/subscribe systems over structured overlay networks. *IEICE Transactions on Information and Systems*, E91-D(5):1504–1511, 2008.

[28] Y. Choi and D. Park. Mirinae: A peer-to-peer overlay network for large-scale content-based publish/subscribe systems. In *Proceedings of International Workshop on Network and Operating System for Digital Audio and Video*, pages 105–110, 2005.

[29] Y. Choi, K. Park, and D. Park. HOMED: a peer-to-peer overlay architecture for large-scale content-based publish/subscribe system. In *Proceedings of International Conference on Very Large Databases*, pages 20–25, 2004.

[30] J. Chu, K. Labonte, and B.N. Levine. An evaluation of chord using traces of peer-to-peer file sharing. *Proceedings of ACM SIGMETRICS Performance Evaluation Review*, 32(1):432–433, 2004.

[31] B.-G. Chun, B.Y. Zhao, and J.D. Kubiatowicz. Impact of neighbor selection on performance and resilience of structured P2P networks. In *Proceedings of International Workshop on Peer-to-Peer Systems*, pages 264–274, 2005.

[32] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[33] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of ACM SIGCOMM*, pages 177–190, 2002.

[34] P. Costa and D. Frey. Publish-subscribe tree maintenance over a DHT. In *Proceedings of International Workshop on Distributed Event-Based Systems*, pages 414–420, 2005.

[35] P. Costa and G.P. Picco. Semi-probabilistic content-based publish-subscribe. In *Proceedings of International Conference on Distributed Computing Systems*, pages 575–585, 2005.

[36] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of International Workshop on the Web and Databases*, pages 25–30, 2004.

[37] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE TKDE*, 15(1):174–191, 2003.

[38] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[39] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of Symposium on Operating Systems Principles*, pages 202–215, 2001.

[40] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proceedings of International Conference on Data Engineering*, page 341, 2002.

[41] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[42] S. Dolev. *Self-stabilization*. MIT Press, 2000.

[43] A. Oram (ed.). *Peer-to-peer: harnessing the power of disruptive technology*. O'Reilly, 2001.

[44] P.Th. Eugster, P. Felber, R. Guerraoui, and S.B. Handurukande. Event systems: How to have your cake and eat it too. In *Proceedings of International Conference on Distributed Computing Systems*, pages 625–632, 2002.

[45] P.Th. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[46] I.T. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Proceedings of International Workshop on Peer-to-Peer Systems*, pages 118–128, 2003.

[47] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of International Conference on Very Large Data Bases*, pages 444–455, 2004.

[48] G. Giakkoupis and V. Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In *Proceedings of Symposium on Principles of Distributed Computing*, pages 302–311, 2005.

[49] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *Proceedings of IEEE Infocom*, pages 119–130, 2004.

[50] Gnutella. http://www.gnutella.com/.

[51] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *Proceedings of IEEE Infocom*, pages 2253–2262, 2004.

[52] B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. In *Proceedings of IEEE Infocom*, pages 596–606, 2005.

[53] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *Proceedings of International Conference on Distributed Computing Systems*, pages 360–369, 2004.

[54] Object Management Group. CORBA event service specification, version 1.1, 2001.

[55] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of Symposium on Operating Systems Principles*, pages 314–329, 2003.

[56] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM*, pages 381–394, 2003.

[57] A. Gupta, P. Dinda, and F. E. Bustamante. Distributed popularity indices. In *Proceedings of ACM SIGCOMM*, 2005.

[58] A. Gupta, O.D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *Proceedings of Middleware*, pages 254–273, 2004.

[59] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*, pages 47–57, 1984.

[60] D. Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. *Mathematische Annalen*, 38(2):459–460, 1891.

[61] IETF-RFC:3174. Secure hash algorithm 1 (SHA1), 2001.

[62] M. A. Jaeger, G. Mühl, M. Werner, and H. Parzyjegla. Reconfiguring self-stabilizing publish/subscribe systems. In *Proceedings of International Workshop on Distributed Systems: Operations and Management*, pages 233–238, 2006.

[63] M.A. Jaeger. Self-organizing publish/subscribe. In *Proceedings of International doctoral symposium on Middleware*, pages 1–5, 2005.

[64] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of International Conference on Very Large Databases*, pages 500–509, 1994.

[65] D. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures*, pages 36–43, 2004.

[66] A. Rao Karthik, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured P2P systems. In *Proceedings of International Workshop on Peer-to-Peer Systems*, pages 68–79, 2003.

[67] K. Kenthapadi and G.S. Manku. Decentralized algorithms using both local and random probes for P2P load balancing. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures*, pages 135–144, 2005.

[68] A. Klemm, C. Lindemann, M.K. Vernon, and O.P. Waldhorst. Characterizing the query behavior in peer-to-peer file sharing systems. In *Proceedings of Internet Measurement Conference*, pages 55–67, 2004.

[69] A. Kumar, J. Xu, and E.W. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *Proceedings of IEEE Infocom*, pages 1162–1173, 2005.

[70] J. Ledlie and M. Seltzer. Distributed, secure load balancing with skew, heterogeneity, and churn. In *Proceedings of IEEE Infocom*, pages 1419–1430, 2005.

[71] G. Li, S. Hou, and H. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *Proceedings of International Conference on Distributed Computing Systems*, pages 447–457, 2005.

[72] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of Symposium on Principles of Distributed Computing*, pages 233–242, 2002.

[73] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of International Conference on Supercomputing*, pages 84–95, 2002.

[74] N. Lynch and I. Stoica. Multichord: A resilient namespace management algorithm. Technical report, Technical Memo MIT-LCS-TR-936, CSAIL, Massachusetts Institute of Technology, Cambridge, 2004.

[75] G.S. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *Proceedings of Symposium on Principles of Distributed Computing*, pages 197–205, 2004.

[76] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[77] G. Mühl. Generic constraints for content-based publish/subscribe. In *Proceedings of International Conference on Cooperative Information Systems*, pages 211–225, 2001.

[78] G. Mühl, M. Jaeger, K. Herrmann, T. Weis, A. Ulbrich, and L. Fiege. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. *Lecture Notes in Computer Science*, 3648/2005:664–674, 2005.

[79] V. Muthusamy and H.-A. Jacobsen. Small scale peer-to-peer publish/subscribe. In *Proceedings of Workshop on Peer-to-Peer Knowledge Management*, 2005.

[80] M. Naor and U. Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *Proceedings of Symposium on Parallel algorithms and architectures*, pages 50–59, 2003.

[81] Napster. http://www.napster.com/.

[82] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. *ACM Special Interest Group on Operating Systems*, 27(5):58–68, 1993.

[83] A. Ouksel, O. Jurca, I. Podnar, and K. Aberer. Efficient probabilistic subsumption checking for content-based publish/subscribe systems. In *Proceedings of Middleware*, pages 121–140, 2006.

[84] G. Perng, C. Wang, and M. Reiter. Providing content-based services in a peer-to-peer environment. In *Proceedings of International Conference on Very Large Databases*, pages 74–79, 2004.

[85] C. G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[86] M. Punceva and K. Aberer. Efficient search in structured peer-to-peer systems: Binary v.s. k-ary unbalanced tree structures. In *Proceedings of International Workshop on Databases, Information Systems and Peer-to-Peer*, 2003.

[87] C. Raiciu, D.S. Rosenblum, and M. Handley. Revisiting content-based publish/subscribe. In *Proceedings of International Conference on Distributed Computing Systems*, page 19, 2006.

[88] V. Ramasubramanian and E.G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of Networked System Design and Implementation*, pages 99–112, 2004.

[89] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[90] S. Ratnasamy, I. Stoica, and S. Shenker. Routing algorithms for DHTs: Some open questions. In *Proceedings of International Workshop on Peer-to-Peer Systems*, pages 45–52, 2002.

[91] R. Renesse and A. Bozdog. Willow: DHT, aggregation, and publish/subscribe in one protocol. In *Proceedings of International Workshop on Peer-to-Peer Systems*, pages 173–183, 2004.

[92] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[93] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, pages 156–170, 2002.

[94] N. Sarshar, P.O. Boykin, and V.P. Roychowdhury. Percolation search in power law networks: Making unstructured peer-to-peer networks scalable. In *Proceedings of International Conference on Peer-to-Peer Computing*, pages 2–9, 2004.

[95] B. Segall and D. Arnold. Elvin has left the building: a publish/subscribe notification service with quenching. In *Proceedings of Australian UNIX and Open Systems Users Group Conference*, 1997.

[96] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proceedings of AUUG2k*, 2000.

[97] S. Serbu, S. Bianchi, P. Kropf, and P. Felber. Dynamic load sharing in peer-to-peer systems: When some peers are more equal than others. In *Proceedings of Montreal Conference on eTechnologies*, pages 149–156, 2006.

[98] S. Serbu, S. Bianchi, P. Kropf, and P. Felber. Dynamic load sharing in peer-to-peer systems: When some peers are more equal than others. *IEEE Internet Computing*, 11(4):53–61, 2007.

[99] Z. Shen and S. Tirthapura. Self-stabilizing routing in publish-subscribe systems. In *Proceedings of International Workshop on Distributed Event-Based Systems*, pages 92–97, 2004.

[100] Z. Shen and S. Tirthapura. Approximate covering detection among content-based subscriptions using space filling curves. In *Proceedings of International Conference on Distributed Computing Systems*, page 2, 2007.

[101] C. Shirky. What is P2P ... and what isn't, http://pragmatic.oreilly.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html., 2000.

[102] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. White paper, Carnegie Mellon University, 2001.

[103] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, 2001.

[104] G. Swart. Spreading the load using consistent hashing: A preliminary report. In *Proceedings of Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 169–176, 2004.

[105] D. Tam, R. Azimi, and H. A. Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Proceedings of DBISP2P*, pages 138–152, 2003.

[106] W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A.P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of International Conference on Very Large Databases*, pages 1–8, 2003.

[107] S. Tewari and L. Kleinrock. Analysis of search and replication in unstructured peer-to-peer networks. *Proceedings of ACM SIGMETRICS Performance Evaluation Review*, 33(1):404–405, 2005.

[108] P. Triantafillou and I. Aekaterinidis. Content-based publish/subscribe over structured P2P networks. In *Proceedings of International Conference on Very Large Databases*, pages 104–109, 2004.

[109] S. Voulgaris, E. Riviere, A.-M. Kermarrec, and M. van Steen. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *Proceedings of International Workshop on Peer-to-Peer Systems*, 2006.

[110] J. Wang, B. Jin, J.W., and J. Li. A reliable content-based routing protocol over structured peer-to-peer networks. In *Proceedings of On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*, pages 373–390, 2004.

[111] X. Wang, Y. Zhang, X. Li, and D. Loguinov. On zone-balancing of peer-to-peer networks: Analysis of random node join. In *Proceedings of ACM SIGMETRICS*, pages 211–222, 2004.

[112] Y.-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H.J. Wang. Subscription partitioning and routing in content-based publish/subscribe networks. In *Proceedings of International Symposium on Distributed Computing*, pages 28–30, 2002.

[113] Y.-M. Wang, L. Qiu, C. Verbowski, D. Achlioptas, G. Das, and P. Larson. Summary-based routing for content-based event distribution networks. *SIGCOMM Computer Commununications Review*, 34(5):59–74, 2004.

[114] J. Xu, A. Kumar, and X. Yu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 22(1):151–163, 2004.

[115] Z. Xu and P.K. Srimani. Self-stabilizing publish/subscribe protocol for P2P networks. In *Proceedings of International Workshop on Distributed Computing*, pages 129–140, 2005.

[116] H. Yamamoto, D. Maruta, and Y. Oie. Replication methods for load balancing on distributed storages in P2P networks. In *Proceedings of Symposium on Applications and the Internet*, pages 264–271, 2005.

[117] X. Yang and Y. Hu. A DHT-based infrastructure for content-based publish/subscribe services. In *Proceedings of International Conference on Peer-to-Peer Computing*, pages 185–192, 2007.

[118] C. Zhang, A. Krishnamurthy, R.Y. Wang, and J.P. Singh. Combining flexibility and scalability in a peer-to-peer publish/subscribe system. In *Proceedings of Middleware*, pages 102–123, 2005.

[119] R. Zhang and Y. C. Hu. HYPER: A hybrid approach to efficient content-based publish/subscribe. In *Proceedings of International Conference on Distributed Computing Systems*, pages 427–436, 2005.

[120] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[121] M. Zhong and K. Shen. Popularity-biased random walks for peer-to-peer search under the square-root principle. In *Proceedings of International Workshop on Peer-to-Peer Systems*, 2006.

[122] Y. Zhu and Y. Hu. Towards efficient load balancing in structured P2P systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, pages 20–29, 2004.

[123] Y. Zhu and Y. Hu. Efficient, proximity-aware load balancing for DHT-based P2P systems. *IEEE Transactions on Parallel and Distributed Systems*, 16(4):349–361, 2005.

[124] Y. Zhu and Y. Hu. Ferry: an architecture for content-based publish/subscribe services on P2P networks. In *Proceedings of International Conference on Parallel Processing*, pages 14–17, 2005.

[125] Y. Zhu and Y. Hu. Ferry: A P2P-based architecture for content-based publish/subscribe services. *IEEE Transactions on Parallel and Distributed Systems*, 18(5):672–685, 2007.

[126] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, and J.D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of International Workshop on Network and Operating System for Digital Audio and Video*, pages 11–20, 2001.

[127] H. Zhuge, X. Sun, J. Liu, E. Yao, and X. Chen. A scalable P2P platform for the knowledge grid. *IEEE Transactions on Knowledge and Data Engineering*, 17(12):1721–1736, 2005.

APPENDIX

A Publications

[1] S. Bianchi, P. Felber and M. Gradinariu. Stabilizing Distributed R-trees for Peer-to-Peer Content Routing. *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 2008.

[2] S. Serbu, S. Bianchi, P. Kropf and P. Felber. Dynamic Load Sharing in Peer-to-Peer Systems: When some Peers are more Equal than others. *IEEE Internet Computing Special Issue on Resource Allocation*, 11(4):53-61, 2007.

[3] S. Bianchi, P. Felber, and M. Gradinariu. Content-based Publish/Subscribe using Distributed R-trees. In *Proceedings of International Conference on Parallel and Distributed Computing*, pages 537-548, 2007.

[4] S. Bianchi, A.K. Datta, P. Felber, and M. Gradinariu. Stabilizing Dynamic R-tree based Spatial Filters. In *Proceedings of International Conference on Distributed Computing Systems*, pages 27-27, 2007.

[5] S. Bianchi, S. Serbu, P. Felber and P. Kropf. Adaptative Load Balancing for DHT Lookups. In *Proceedings of International Conference on Computer Communications and Networks*, pages 411-418, 2006.

[6] S. Serbu, S. Bianchi, P. Kropf and P. Felber. Dynamic Load Sharing in Peer-to-Peer Systems: When some Peers are more Equal than Others. In *Proceedings of Montreal Conference on eTechnologies*, pages 149-156, 2006. Nominated for Best Paper Award.