Infrastructureless Data Dissemination: A Distributed Hash Table Based Publish/Subscribe System

by

Vinod Muthusamy

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Infrastructureless Data Dissemination: A Distributed Hash Table Based Publish/Subscribe System

Vinod Muthusamy

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2005

Peer-to-peer networks can offer benefits to distributed content-based publish/subscribe data dissemination systems. In particular, since a peer-to-peer network's aggregate resources grows as the number of participants increases, scalability can be achieved without managing or deploying additional infrastructure. This thesis proposes an efficient algorithm for supporting publish/subscribe subscriptions that specify a range of interest. The algorithm is built over the Pastry distributed hash table and is completely decentralized. Load balance is addressed by subscription delegation away from overloaded peers, and a bottom up tree search technique that avoids root hotspots. As well, fault-tolerance is achieved with a light-weight replication scheme that quickly detects and recovers from faults. Simulations support the scalability and fault-tolerance properties of the algorithm.

Acknowledgements

I would like to acknowledge several individuals who have made this thesis possible. First, my supervisor Hans-Arno Jacobsen has been invaluable in guiding this research and providing valuable feedback along the way. I would also like to thank Microsoft Research for allowing use of their Pastry simulator, and in particular Antony Rowstron, Miguel Castro, and Manuel Costa for helping me setup the simulator and answer any questions I had. Finally, my colleagues in the Middleware Systems Research Group, including Milenko Petrovic and Eli Fidler have provided me with different perspectives and helped me gain new insights on this research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet is increasingly being used as a communications medium for many diverse applications. Distributed applications often rely on some middleware primitives in order to communicate with processes distributed across the network. Some such primitives include remote procedure call (RPC), and message passing. Expanding on the message passing model, publish/subscribe is a powerful event dissemination model that can be used as a communications layer to build many complex applications, such as selective information dissemination. and workflow management. Publish/Subscribe is a data dissemination model whose scalability [1, 2] and ability to decouple data producers from consumers lends itself well to large-scale Internet applications. An innovation in the publish/subscribe paradigm is that processes no longer communicate with each other based on their addresses. Instead processes specify properties of the *data* they are interested in receiving, and any data that enters the system is automatically delivered to all interested processes by the middleware.

A real-world application of the publish/subscribe model is IBM's use of a publish/subscribe system to deliver real-time tennis match scores and commentary to millions of users from all over the world [3]. Another possible application is an auction service such as eBay in which a publish/subscribe system can be used to notify an auctioneer when he is outbid on an item, or to inform a seller of all bids on her item. This application consists of millions of users and the

dissemination of possibly thousands of data items per second.

Common to these applications is the selective dissemination of data to a very large number of geographically scattered users.

1.1 Motivation

Large-scale publish/subscribe applications currently require the resources of a company such as IBM to deploy and administer. Such applications could reasonably require dozens or hundreds of servers placed at strategic points across the globe in order to handle the immense load, and trained personnel are needed to carefully monitor the network for bottlenecks and strategically deploy servers and network bandwidth to resolve the bottlenecks. For example, many popular Web sites on the Internet use the Akamai [4] content distribution infrastructure to cache the Web site contents across the Internet. The size of the Akamai network—"more than 14000 servers ... in 65+ countries" [4]—gives an indication of the extent of Akamai's infrastructure management task.

Our proposed design virtually eliminates publish/subscribe infrastructure costs. Instead of requiring dedicated servers, the users' resources are automatically exploited to achieve infrastructureless scalability. In addition, the design self-organizes to adapt to bottlenecks and faults so no personnel is required to administer the network. Infrastructure-less scalability is achieved by employing a new generation of peer-to-peer (P2P) networks based on a distributed hash table (DHT) [5, 6].

Traditional distributed publish/subscribe systems such as Siena [2], Gryphon [7], JEDI [8], and Rebeca [9] all build an application level overlay broker network. The configuration of the overlay plays an important role in the performance of the system. For instance, an overlay that is very different from the underlying physical network will have a worse bandwidth utilization and latency for delivering events than an overlay that maps closely to the physical topology. As well, the overlay configuration can affect the routing state stored at each broker.

Despite the importance of the overlay topology, the mapping between the broker and physical network is typically performed manually by a network deployer. This task becomes more difficult as the number of brokers in the network becomes large. Furthermore, the topology is often static, and is not adapted to changing usage patterns. In addition, the organization of the broker network as a tree—as in JEDI—results in every node being a single point of failure and a bottleneck.

So, while existing distributed publish/subscribe systems have good scalability properties, they require the purchase of infrastructure (the broker machines and network access) and the administration of this infrastructure. P2P networks, on the other hand, promise infrastructureless scalability without any administration. For this reason, we propose to build a publish/subscribe system over a P2P network.

1.2 Applications

Most existing large-scale publish/subscribe applications can benefit from our design. However, an infrastructureless scheme opens the door to applications that would have otherwise been infeasible. One such class of applications is "grassroots" applications involving users with little resources or expertise. For example, a small shareware developer can benefit from distributing software updates to his users using a publish/subscribe system, but he might not have the resources to purchase and manage such a system. Another class of applications is those that require lots of resources for a short period of time. A prototypical example is disseminating real-time results and play-by-play of Olympics events to those interested. This scenario can have millions of users around the globe and a high rate of data dissemination, and requires a massive publish/subscribe network. However as this network is only used for a few weeks, it might not be cost-effective to deploy such a system. Yet another class of applications is those where the users do not have the expertise to, or do not want to bother with the trouble of, deploying a publish/subscribe network. Consider a corporation that routinely needs to deliver

memos to appropriate employees. A publish/subscribe system is an improvement over using mailing lists as it allows more fine-grained delivery of memos to only those users that are interested in them.

The common theme here is that a publish/subscribe system running on a self-configuring, infrastructureless P2P network lowers the bar on the costs of using a publish/subscribe system, and this can lead to applications that are currently infeasible for reasons of financial cost, technical expertise, or administrative inconvenience.

1.3 Problem statement

This work addresses the distributed publish/subscribe problem in which data from information producers must be routed to all interested users in the network. Chapter 2 discusses the publish/subscribe matching problem in more detail. While existing solutions to this problem exist [2, 9, 8, 10, 7], they all require a dedicated broker network infrastructure. It is both expensive and difficult to deploy and maintain such a network.

This thesis proposes a distributed publish/subscribe solution that is scalable without requiring dedicated infrastructure. In particular, we implement publish/subscribe semantics on top of a DHT peer-to-peer overlay network. DHTs provide a scalable network overlay without requiring any dedicated infrastructure. The challenge is that the DHT interface only supports exact name lookups. Using such an interface, it is simple to develop a publish/subscribe matching algorithm for exact match semantics. However, it is challenging to develop a distributed algorithm that supports range interest specifications in the subscription language. As we will see, despite these obstacles, DHTs still provide desirable properties that we would like to exploit. The problem solved by this thesis is the development of an algorithm that supports true content-based publish/subscribe matching semantics on top of the DHT interface.

In addition to the publish/subscribe matching problem, we also must address publish/subscribe multicast delivery of publications. Existing publish/subscribe multicast algorithms

typically build a static multicast tree. This is sufficient in traditional algorithms where it is assumed that the broker network is relatively stable and reliable. However, in our case, the broker network is made up of the peers in a dynamic P2P network, where the peers enter and leave the network frequently, requiring frequent changes to the multicast tree. The P2P network is also unreliable, and failures in the multicast tree are common. Hence errors in the tree must be detected and fixed quickly. The unreliable network also necessitates developing fault-tolerance mechanisms to recover from faults.

In summary, this thesis attempts to develop a scalable infrastructureless distributed publish/subscribe system. To achieve this, a DHT network is used, but implementing a content-based publish/subscribe algorithm using a limited DHT interface is not straightforward.

1.4 Contributions

The main contribution of this work is the development of an algorithm to perform distributed content-based publish/subscribe matching and data dissemination using a DHT interface. We develop three algorithms, based on mapping the publish/subscribe matching problem into a multidimensional indexing problem. The first is able to match one publish/subscribe attribute at a time, while the second one can match all attributes simultaneously. The third algorithm removes a limitation of the second one that required a global fixed schema of attributes in the system. The matching algorithm developed is able to search a distributed tree without a root overload problem present in most tree based indexing algorithms. As this algorithm runs on a distributed system, a load-balancing mechanism is used by peers to move subscriptions away from overloaded nodes in a decentralized fashion. Due to the dynamic nature of the peer-to-peer network the system is build on, a light-weight replication scheme is devised to detect and recover from faults. In addition to this active fault-tolerance mechanism, which detect faults when publication or subscription traverses the network, the system also recovers from faults passively, when there is no traffic in the network, using soft state and beaconing. The algorithm

is extensively evaluated in a simulation environment. The evaluation shows that the network load of the algorithm scales well, and it it delivers quality service to the users by correctly delivering a high percentage of publication quickly.

1.5 Outline

In order to keep the thesis self-contained, Chapter 2 presents background information on the publish/subscribe model, P2P networks, and multidimensional indexing algorithms. This background is necessary to understand the concepts this work builds on. Chapter 3 describes related work in building publish/subscribe systems and databases over a DHT. The related work provides some context for this research by comparing it to similar systems. In Chapter 4, the distributed publish/subscribe matching algorithm is developed, and Chapter 5 completes the design by describing distributed publish/subscribe issues that are not related to the matching problem. Chapter 6 evaluates the algorithms presented in the previous two chapters, and Chapter 7 completes the thesis with some concluding remarks and discussion of future work.

Chapter 2

Background

In this thesis we implement a distributed publish/subscribe system over a P2P DHT substrate by first mapping the publish/subscribe problem into a multidimensional indexing problem. To this end, in order to keep the thesis self-contained, this chapter provides background on the publish/subscribe model, touching on the semantics and benefits of the model; P2P architectures, focusing on structured DHT networks; and briefly describes some relevant multidimensional indexing algorithms.

2.1 Publish/Subscribe

2.1.1 Publish/Subscribe model

Publish/Subscribe is a data dissemination model that has many useful properties. There are three main entities in the publish/subscribe model: the *publisher*, *subscriber*, and *broker*. The publisher is the data producer, the subscriber the data consumer and the broker mediates between the publisher and subscriber. These entities are only logical and do not have to correspond to physical components. For example, it is possible for a computer to be a publisher, subscriber, and broker.

A popular example of the publish/subscribe model is a stock quote information dissemi-

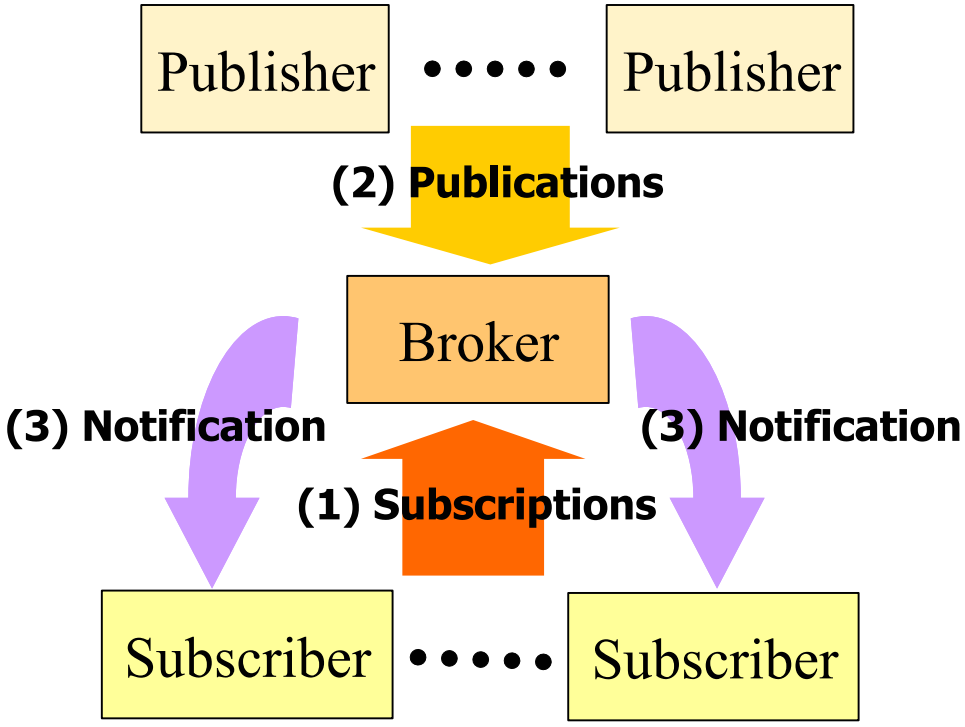

Figure 2.1: Publish/Subscribe message sequence

nation application. In this example, the publisher would be the stock exchange, such as the Toronto Stock Exchange, and the consumer could be a stock broker who is interested in tracking the latest prices of certain stocks. A subscriber S expresses his interest in these stocks by sending a *subscription* message s to the broker. The broker stores these subscriptions in an index T as a set of $(subscription, subscriber)$ tuples. The publisher P communicates the latest stock updates by sending a *publication* message p to the broker. Upon receipt of a publication p, the broker searches index T for the set of subscribers whose subscriptions *match* p, and notifies these subscribers of a matching publication (usually by simply forwarding p to the subscribers). This sequence of events is illustrated in Figure 2.1.

The publish/subscribe model is in some ways the dual of the traditional database model. In a database, data is persisted in a database and queries retrieve this data, whereas in the publish/

subscribe model, the queries (subscriptions) are stored at the broker, and the data (publications) are pushed to the subscribers. A subscription can be thought of as a long running query, so it may be helpful to think of publish/subscribe model as being similar to database triggers [11]. It is important to note that subscriptions only match future publications. Publications enter the system, are sent to interested subscribers, and are then lost. This is an important difference from the database model. Another point to note is that data only travels from publisher to subscriber. Of course, in a given application, bidirectional communication between two components is possible by having the components act as both publisher and subscriber.

2.1.2 Semantics

While there have been several publish/subscribe semantics in the literature [12, 13, 14], the one described here is one of the more expressive ones, and is often used.

A publication p is a set of $\{name, value\}$ predicates. The $name$ is a string that uniquely identifies some unit of data, and $value$ is a string or numeric data type that specifies the value of that data item. There cannot be two predicates in a publication with the same $name$. Using the stock market example, the $name$ could be the stock ticker symbol, and $value$ the current price of that stock. The publication $\{(stock, IBM), (price, 82.18)\}$ specifies that IBM's current stock price is \$82.18.

A subscription s is a conjunction of predicates, where a predicate consists of a tuple of $\{name, operator, value\}$. As with a publication, the $name$ is a string that refers to some unit of data and the $value$ can be a string or numeric data type. The $operator$ and $value$ are used to specify the values in a publication that the subscriber is interested in. The $operator$ can be one of $\{=, <, >, \leq, \geq\}$. Unlike publications, multiple predicates in a subscription can have the same $name$. For example, the subscription $\{(stock, =, IBM), (price, \leq, 80.00), (price, \geq, 70.00)\}$, specifies that the subscriber is interested in being notified of all publications in which IBM's stock price is between \$70.00 and \$80.00.

A subscription predicate (n, o, v) *matches* a publication predicate (n', v') iff $n = n'$ and v'

falls in the range specified by the operator o and value v. A subscription s *matches* a publication p iff every predicate in s matches some predicate in p. For example, the subscription $\{(stock, = , IBM), (price, <, 80.00)\}$ matches the publication $\{(stock, IBM), (price, 78.14)\}$. However it does not match publication $\{(stock, IBM), (price, 82.18)\}$ nor does it match publication $\{(stock, JNJ), (price, 14.38)\}$.

2.1.3 Centralized and distributed algorithms

Early publish/subscribe systems were centralized system [15, 16, 17]; there was only one broker in the system that received all publications and subscriptions in the system. This is the scenario shown in Figure 2.1. Research in these systems were focused on developing algorithms that can quickly and efficiently match a publication against millions of subscriptions. However, in systems with potentially millions of subscriptions and with subscribers dispersed geographically, a centralized broker can still become a processing, memory, and network bottleneck. Newer systems were developed that employed a distributed set of brokers [2, 7, 8, 9]. The research here was focused on distributed matching and multicasting. The former refers to storing subscriptions in a subset of the nodes in the system such that matching can be distributed among the nodes. The latter refers to a technique of disseminating publications to interested subscribers such that total network traffic is minimized.

2.1.4 Benefits

The publish/subscribe model offers many benefits. First the model has a very simple interface. As shown in Figure 2.1, the only messages involved are publications, and subscriptions, and the only operations are publish, subscribe, and notify.

Another important benefit is the *decoupling* of publisher and subscribers. This decoupling exists along several dimensions as described below.

- **Address decoupling** Publishers and subscribers do not know the addresses of one other.

Instead publications are routed to subscribers based on the content of the publications and the interests of the subscribers. For this reason, publish/subscribe is also often referred to as *content-based routing*. Another advantage of address decoupling is the anonymity it offers to the publishers and subscribers. Only the brokers know the identity and interest of a subscriber, and the data published by a given publisher.

- **Platform decoupling** All entities in the publish/subscribe communicate using network messages, and as such can run on heterogeneous platforms. Therefore a powerful multiprocessor server with a dedicated network connection can seamlessly communicate to a resource constrained PDA with a wireless link.

- **Space decoupling** As mentioned above, publishers and subscribers can exist in geographically distant areas of the globe.

- **Time decoupling** Some publish/subscribe algorithms [18, 19] allow publishers and subscribers to not be connected to the network simultaneously, allowing subscribers that experience network disconnection to retrieve missed publications upon reconnection.

- **Representation (semantic) decoupling** Some publish/subscribe systems such as [13] can mediate between publications and subscriptions whose predicates have different meaning. For example, in a dating service application, it is possible to match publications that specify a user's age in *years*, and subscriptions that request users *born before* a certain date.

Publish/Subscribe also provides benefits regarding network bandwidth use. Notably, data is *pushed* to subscribers as it becomes available. This is more efficient than having subscribers periodically poll for new data. A system with millions of subscribers polling a broker can easily overwhelm a broker. Data push also delivers data quicker to subscribers than polling. Furthermore, distributed publish/subscribe systems use efficient multicast techniques to minimize the messages used by subscribers. An example of the benefits of multicast is shown in Figure 2.2

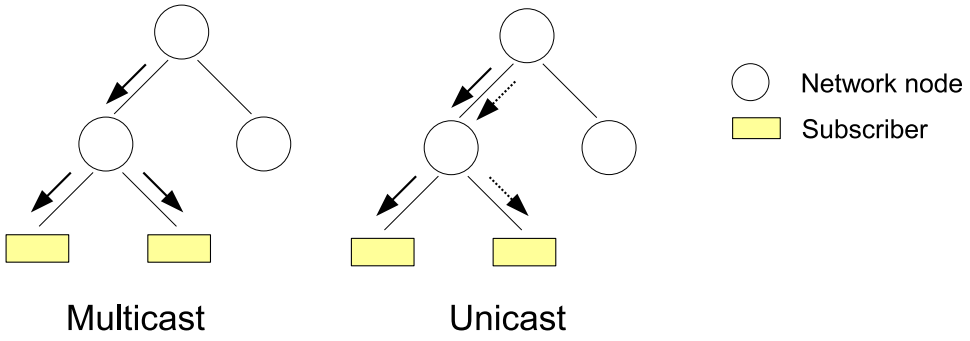

Figure 2.2: Comparison of multicast and unicast propagation

in which the publication only travels a total of three hops to reach both subscribers. *Unicasting* the publication, that is, sending the publication individually from the publisher to each subscriber, would require four hops. In a large system with millions of subscribers, multicasting is much more bandwidth efficient than unicasting.

2.2 Peer-to-peer

A popular architecture used by Internet applications is the client/server model. This model is characterized by a server that provides some service to a client. Communication is initiated by the client using a well-known server address. The resource provided by the server includes content, storage, and computation. For example, in World Wide Web applications, the server provides content to the client in the from of Web pages. The client/sever model is popular in Internet applications but does suffer some limitations. For instance, even powerful servers can hit scalability limits. Furthermore, the server represents a single point for failure for the service provided by the server. These concerns are usually addressed with clusters of servers, but this introduces complexity; issues such as replica consistency, failure detection, automatic failover, and load balancing need to be considered. Another limitation of the client/server model arises from the asymmetry between the client and server. In most such applications, the server has little unused resources whereas the clients, in aggregate, have ample unused resources. For

example, while a Web server can often reach resource limits, the user with a desktop computer often has unused bandwidth, memory, storage and processing power.

Peer-to-peer networks address some of these limitations of the client/server model. P2P networks are characterized by the direct sharing of resources among the peers in the network. These resources include storage, memory and processing power. The key to P2P networks is to make use of the unused resources at the edge nodes in the network—as opposed to the core servers—to benefit all users.

In P2P architectures, nodes take on the role of clients, servers and routers; there are no nodes that are dedicated to serving other as in client/server architectures. P2P nodes are autonomous; in the client/server model the servers are under some administrative authority while nodes in a P2P architecture are not under any administrative control.

Nodes in a P2P architecture conspire to create neighbour relationships among themselves. These neighbour relationships form an application level network topology that does not necessarily correspond to the physical network topology. This P2P network topology is referred to as an *overlay network*, a *routing substrate* or an *overlay routing substrate*. We use these terms interchangeably.

P2P networks are also dynamic, with nodes entering and leaving the network frequently, while in the client/server model, the servers usually enjoys long uptimes. Another implication of all nodes playing an equal role is that any node can initiate communication with any other node. Also, nodes in a P2P network usually have widely varying capabilities. This includes wide differences in network bandwidth, processing power, memory and storage.

P2P networks promise several benefits. As mentioned above, the main advantage is the use of otherwise unused resources at the edge nodes of the network. The requirement that all users in the network contribute their resources to the network leads to an interesting property of *organic scaling* [20]: the aggregate resources in the network grows naturally with the application utilization. The distribution of work among many nodes in the network means that, if designed properly, P2P applications have geographically distributed replicas and do not suffer

from a single point of failure. Finally, P2P protocols are designed to self-organize, having no administrative requirement. There is no need to deploy additional resources to satisfy demand (see also organic scalability), and the protocols often have built-in fault-tolerance, replication and load balancing features.

It should be clear from the discussion that P2P does not attempt to replace all client/server applications. There are many applications for which each model is more appropriate. For example, the client/server model is probably more appropriate for applications that require some administrative control, or those that do not require very large scale.

P2P protocols can be loosely classified as unstructured and structured protocols. The following sections will discuss these protocol classes in more detail.

2.2.1 Unstructured P2P networks

The first generation of P2P networks such as Napster [21] Gnutella [22], and the Fasttrack network [23] are called unstructured networks. These networks have been used for large scale file sharing applications. Common among the protocols is that some user makes files on their computer available for download to other users. These other users search for files of interest and then download these files *directly* from the sharer's computer. The protocols differ mainly on the mechanism used to search for files shared by users.

As will be seen in Section 2.2.2, a key limitation of the unstructured networks is that they provide no performance guarantees. Below, the key features of a sampling of unstructured P2P protocols are discussed.

Napster

Napster uses a centralized index server. File sharers upload a list of files they wish to share to this central server, and users query this server for files of interest. The server replies to these queries with the addresses of the peers in the network with matching files, and the user then downloads these files directly from the sharers. Napster is characterized by a centralized

(client/server) search, but P2P file transfer.

The advantage of a centralized search server is that correct results can be ensured. However, this server can become a scalability bottleneck, becomes a single point of failure, and is susceptible to denial of service (both by malicious users, and lawsuits and legislation).

Gnutella

Gnutella lies on the other end of the spectrum of Napster, and has no centralized nodes. The network topology is built in an ad-hoc fashion with no structure imposed; each node connects to, and becomes the neighbour of, an existing Gnutella node, whose address it discovers through some out-of-band mechanism such as a Web page of known Gnutella nodes. The search protocol is completely decentralized. To search for a file, a user asks her neighbours which in turn ask their neighbours, and so on. A time-to-live field in the query stops the query after a certain number of hops. A node with matching files replies to the requester, who then downloads their file directly from this node.

The distributed search does not suffer from a single point of failure but can no longer ensure that all matching files will be found. Also, the search protocol is not scalable as a single query can result in many messages.

Fasttrack

The Fasttrack network uses a search protocol that is a hybrid of the centralized Napster protocol and the completely decentralized Gnutella. In the Fasttrack protocol, super-peers act as local search hubs. Each super-peer plays a similar role as Napster's centralized server, but only indexes the files in a small portion of the network. Super-peers are automatically chosen by the system based on a node's resources and availability. Users upload information about the files they are sharing to a local super-peer, and super-peers periodically exchange the file lists to propagate file information throughout the network.

store(key, value)	Store $(key, value)$ at the $peer$ responsible for key
lookup(key)	Retrieve the $value$ associated with key (from the appropriate $peer$).

Table 2.1: DHT interface

2.2.2 Structured P2P networks

The second generation of P2P networks, based on a distributed hash table (DHT), are called structured P2P networks. DHTs differ from unstructured P2P networks by providing theoretical performance guarantees. These guarantees include the balance of key storage among the peers in the DHT, the expected routing state a peer needs to maintain, and the number of hops required to lookup a key in the DHT. DHTs are self-organizing, load balanced and fault-tolerant. They provide statistical guarantees on bandwidth usage and node state requirements.

A DHT is a distributed version of a hash table. A DHT stores a (key,value) pair at a node in the network that is deterministically computed from the key. This allows another node to retrieve the value based solely on the key. Each peer in the network stores a subset of the (key, value) pairs in the system. The core operation of a DHT protocol is to map a key to a node and efficiently route messages to this node.

A DHT implementation is an application layer (OSI layer 7) protocol that provides an Application Programming Interface (API) for other applications. The DHT interface consists of the insert() and lookup() functions described in Table 2.1.

A primary motivation of early DHTs was distributed file storage, where the key might be the filename, and the value the contents of the file. However, both key and value may be an arbitrary string of bytes. A surprising number of diverse distributed applications can be built on top of a DHT interface including data streaming [24], co-operative messaging [25], storage systems [26, 27], name servers [28], file sharing [29], databases [20], and Internet telephony [30].

To facilitate the exposition, we will use the Pastry [6] DHT implementation when discussing DHT specifies. Other DHTs have similar interfaces but differ in the implementations and performance. In the discussion below, we assume a DHT network of N nodes and K keys.

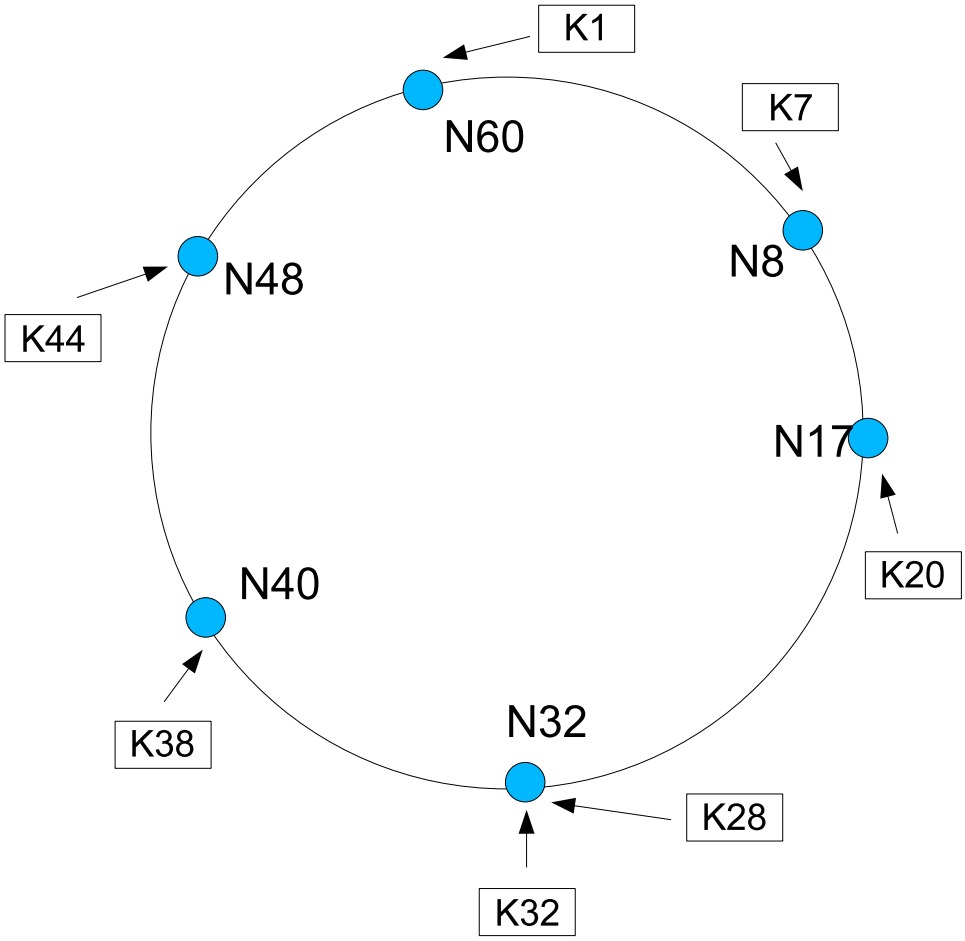

Figure 2.3: Node and key assignment in an identifier space with range [0,61]

Identifier circle

In the (key, value) pair stored in the DHT, the $value$ is a sequence of bytes and the key is a 128-bit number. Each peer in the DHT network is addressed by a 128-bit *node identifier*. The key and nodeid are a sequence of 2^b digits (where b is a parameter that is typically equal to 4) and belong to a circular 128-bit identifier space, also referred to as an identifier circle. The keys and nodeids are generated by the SHA-1 [31] cryptographic hash to ensure an even distribution of keys and nodeids around the identifier circle. The use of the hash function to generate nodeids also ensure that, with high probability, nearby nodes in the identifier circle are distant geographically and network-wise.

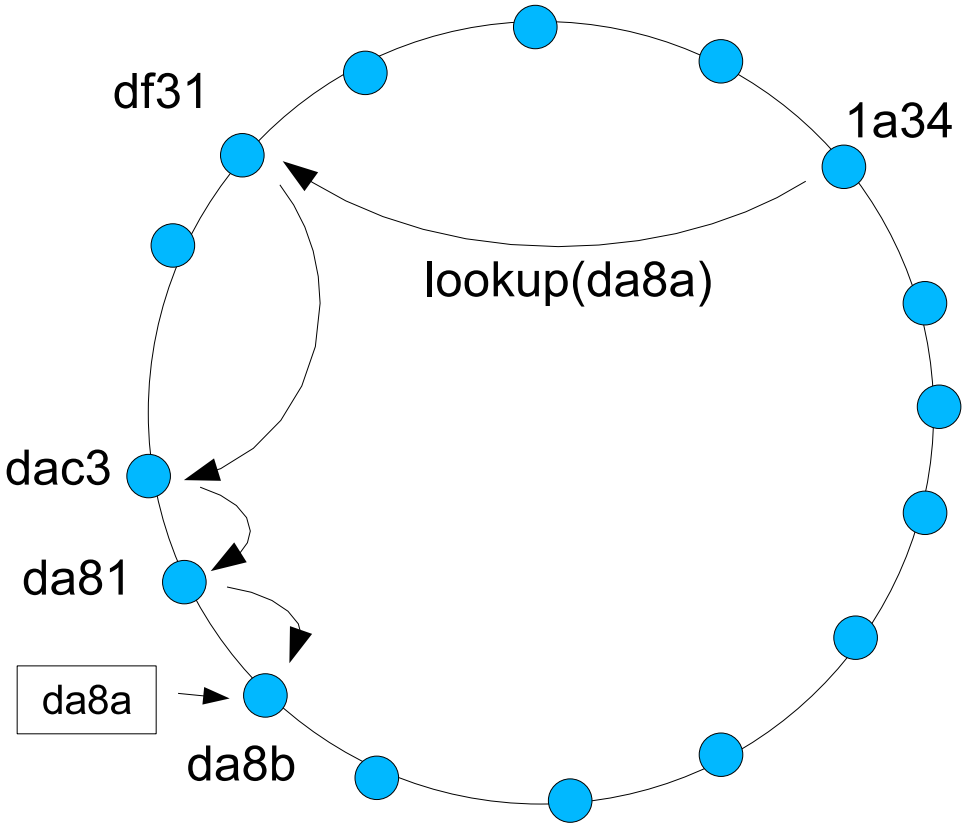

Figure 2.4: Prefix routing

Key assignment

The mapping of keys to nodes is a basic function of a DHT. As illustrated in Figure 2.3 Pastry maps keys to the node with the numerically closest nodeid in the identifier circle. Note that since the cryptographic hash function distributes keys and nodeids evenly around the identifier circle, no node stores a disproportionate share of the (key, value) pairs in the system. In fact, in a network with N nodes and K keys, each node stores K/N keys with high probability.

Routing

Pastry uses a prefix routing algorithm in which each hop of a message is sent to a node that matches the destination nodeid by at least one more digit (b bits). If no such node is known, then the message is forwarded to a node that is numerically closer to the destination nodeid

than the current node's id. As will be described below, Pastry attempts to maintain enough routing state at each node to perform prefix routing. An example of the message traversal performed to achieve a key lookup is shown in Figure 2.4. This routing algorithm is able to route a message to the destination in $\lceil \log_{2^b} N \rceil$ hops. Other DHT implementation use various other routing algorithms to achieve their performance guarantees. It is important to note that a hop here refers to a message sent from one node to another in the DHT overlay network. This hop may correspond to more than one physical network hop.

Fault-tolerance

Pastry guarantees eventual delivery of a message unless $\lfloor |L|/2 \rfloor$ nodes with adjacent nodeids fail concurrently. (As described below, $|L|$ is the leaf set size, and is usually set to 16 or 32.) Note that the routing guarantee of $O(\log N)$ hops is not guaranteed when failures occur.

Node state

As described in [6], Pastry nodes maintain a *routing table,* a *neighbourhood set* and a *leaf set*.

The routing table, R, stores various nodeids in a table with $\lceil \log_{2^b} N \rceil$ rows and $2^b - 1$ columns. The entry in the ith row and jth column has the first i digits in common with the current node's id, but has a jth digit of j. Each entry in R also stores the IP address of the node with the corresponding nodeid. Several nodeids meet the constraints of an an entry in R. Among these nodeids, Pastry chooses the nodeid of a "nearby" node. Typically the number of physical hops is used to determine the proximity between nodes. The value of b can be used to tradeoff routing table size (about $\lceil \log_{2^b} N \rceil * (2^b - 1)$ entries) and routing hops ($\lceil \log_{2^b} N \rceil$).

The neighbourhood set M stores the nodeids and IP addresses of the $|M|$ closest nodes (according to the proximity metric) to the current node. $|M|$ is typically $2 * 2^b$. While the neighbourhood set is not required for correct routing, it is used to maintain locality properties in order to optimize routing.

The leaf set L stores the nodeids and IP address of the $|L|/2$ numerically closest nodes with

a nodeid larger than the current node in the identifier circle, and the $|L|/2$ smaller such nodes. $|L|$ is typically 2^b. During routing, if the destination id falls within the leaf set, the message is sent directly to the appropriate leaf node, bypassing prefix routing as described above.

Node adaptation

Pastry nodes automatically update their state as nodes arrive and depart. A node entering the network must know the IP address of an existing node in the overlay network; the state of this node is used to bootstrap the entering node's state. In Pastry, to maintain good performance, it is desirable for the known node to be close (according to the proximity metric) to the entering node. This is necessary to ensure that a node's routing state initialized with nearby nodes. An entering node requires $O(\log_{2^b} N)$ messages to populate its state.

Node departures or failures can cause entries in the routing table to become unreachable. In this case, a node finds a replacement entry by querying other nodes in its routing table.

This concludes the introduction to the DHTs in general and the Pastry DHT in particular. Below is a brief overview of some other DHT implementations.

2.2.3 DHT implementations

Chord

Chord [5] is similar to Pastry in the use of a circular identifier circle. A minor difference is that Chord stores a key at the first *successor* node in the identifier circle. Each Chord node maintains a *finger table* of m entries, where m is the number of digits in a node identifier. The ith finger points to the first node that is at least 2^{i-1} distance away from the current node in the identifier circle. Each routing hop follows the most distant finger that does not overshoot the destination. This results in successive routing hops reducing the distance to the destination by at least half in the identifier circle.

The finger table allows only one node in the network to meet the criteria for each finger,

unlike the routing table entries in Pastry. Therefore, the Chord protocol cannot favour nearby nodes (according to some proximity metric).

Chord makes a probabilistic guarantee of routing messages in $O(\log N)$ hops with an $O(\log N)$ node routing state. The node join protocol requires $O(\log^2 N)$ messages.

Pastry

As described in 2.2.2, Pastry nodes are organized in a circular identifier space as in Chord, but use prefix routing, in which the prefix of each hop's identifier matches the destination by at least one extra digit. This allows a choice of nodes for each hop. Also, Pastry considers locality—a measure of the physical network distance between nodes—when deciding the best node for each hop. In a network of N overlay nodes, Pastry can theoretically route a message to a destination in $O(\log_{2^b} N)$ hops with $O(2^b \log_{2^b} N)$ routing state, where b is used to trade off between routing hops and state.

CAN

Unlike Chord and Pastry, CAN [32] organizes nodes in a "d-dimensional Cartesian coordinate space on a d-torus." Essentially, this is a d-dimensional space with the range of each dimension wrapping around (like Chord and Pastry's identifier circle). Each node owns a zone in the space, and each key maps to a point in the space and stored at the node that owns that point. A node stores information about its neighbours in the CAN space, and a message is greedily routed towards the neighbour that gets the message closer to the destination.

CAN has $O(2d)$ routing table size and $O(\frac{d}{4}N^{1/d})$ routing path length. Compared to Chord and Pastry's $O(\log N)$ bound for both these metrics, CAN has a better table size, but worse routing path. The constant routing table size is unique to CAN. In addition, CAN provides many parameters that can be used to tune the system for a known workload.

Tapestry

Tapestry [33] is similar to Pastry in the use of an identifier circle and prefix routing, and guarantees the same $O(logN)$ routing hops and state. It adds some additional interfaces to allow storage of duplicate (key, value) pairs across the network for caching purposes. Queries for a key will automatically find the closest copy.

P-Grid

P-Grid [34] organizes nodes in a binary search tree. The nodeid of a node is derived from its position in the tree: in the path from the root to a node, a traversal down the left (right) subtree appends the digit 0 (1) to the nodeid. Keys are stored at the node whose nodeid shares the longest prefix with the key. The prominent difference of P-Grid from other DHT implementations is the tree is automatically reorganized in response to the key distribution; more nodes are placed at portions of the tree with many keys. In this way, P-Grid focuses on the load balancing problem of DHTs.

SkipNet

Most DHT networks do not allow any control over the placement or routing of data. SkipNet [35] allows such control while still providing the performance guarantees of other DHTs.

SkipNet adapts the skip list data structure to a distributed context. Unlike DHTs, each SkipNet node has a user specified name ID, and a system determined numeric ID. In an N node network, with high probability, routing in either ID space can be performed in $O(\log N)$ hops using only one routing table of size $O(\log N)$. The SkipNet scheme to control the storage of data is flexible enough to place data at one specific node, at any node in a domain, or at any node in the network.

While data placement control in SkipNet is alluring, it can introduce complexities that P2P networks are supposed to avoid. In particular, an administrative authority is required to provision the name IDs of nodes in a domain. Another limitation is the support of only one

user specified name ID per node. This restricts the ability to perform range queries to only one dimension, and also confines a node to only one domain.

2.3 Multidimensional indexing

As will be described in Chapter 4, the proposed design in this thesis makes use of a distributed multidimensional indexing structure. This section briefly describes some centralized multidimensional indexing algorithms, and borrows from a survey paper on this topic [36]. This section is not meant to be an exhaustive or even necessarily indicative of the state of the art in the field; instead focus is given to algorithms with techniques that are used in this thesis. The overview is separated into algorithms that index points in a multidimensional space, and those that index regions in this space.

2.3.1 Indexing points

A point indexing structure needs to support exact value and range queries. In a multidimensional space, range queries are represented as a region in the space.

The B-Tree indexes points in a one-dimensional space. A B-Tree is a balanced tree in which each node represents an interval in the space, and the children of a node represent mutually exclusive subsets of the parent's interval.

The k-d-Tree indexes points in a multidimensional space. It is a binary search tree in which each node represents an iso-oriented splitting hyperplane in the space. The splitting hyperplanes serve to subdivide the spaces into recursively smaller subspaces, and must pass through a point to be indexed. The hyperplanes cycle through each dimension in the space. For example, in a three dimensional space, the planes split the x, y, and z axes in order. The k-d-Tree is an unbalanced tree and is sensitive to the order of insertion.

The BD-Tree is a binary tree that subdivides a d-dimensional space into interval-shaped regions that are encoded by a bit string. The bit strings are known as DZ-expressions, Peano

codes, ST_MortonNumbers or z-codes. An alternating series of iso-oriented splitting hyperplanes each divide a region in half along the splitting dimension. Leaf nodes in the tree store the points within that region. Consider a region R with a z-code Z. Suppose that last splitting hyperplane that created R is parallel to the nth dimension (where $0 \leq n < d$). Region R is then split by a hyperplane parallel to the $m = n + 1$ (mod d) dimension. This creates two subregions R' and R''. If R' corresponds the lower half of the mth dimension then, its z-code is $Z0$, that is, 0 appended to the z-code of R. Likewise, the z-code of R'' is $Z1$. The root node has a null z-code. Z-codes are used in our proposed algorithm in Chapter 4.

Space filling curves are used to give a total order to points in multidimensional spaces, thereby mapping a set of multidimensional points into a set of one-dimensional points. This allows structures such as the B-Tree that can only index one-dimensional points to index multidimensional points. A space filling curve splits the space into (arbitrarily small) cells and traces a curve along these cells imposing a total order of these cells.

2.3.2 Indexing regions

Region indexing structures are used to find all regions that overlap or are enclosed by a given region, or to find all regions that enclose a given point.

One technique for indexing regions is to transform a region such that it can be indexed by a point indexing structure. For example, a region can be represented by its centroid, or its corners. Another transformation method is to approximate a region by the set of interval-sized cells that enclose the region. The cells can be represented by a space filling curve or z-code encoding of the cells. The encodings are then stored in a point indexing structure.

An R-Tree is a balanced tree in which each node corresponds to the minimum bounding box (MBB) of a d-dimensional subspace. The MBB of an interior node is the MBB of the MBB of its descendants, while the MBB of a leaf node is the MBB of the regions stored at the leaf. Notice that the MBB of nodes can overlap. A region is only stored at one leaf node, so even point queries may traverse multiple paths down the tree. Improved variants of the R-Tree

include the R*-Tree and the X-Tree. The R+-Tree makes a significant change from the other R-Tree variants by requiring non-overlapping MBB of nodes. This results in regions being stored at multiple leafs (so insertion can require traversing multiple paths in the tree) but point queries can be answered in one traversal.

Chapter 3

Related work

As this thesis builds a distributed publish/subscribe system, this chapter first presents an overview of the canonical distributed publish/subscribe algorithm. This is necessary because some of the concepts present in this canonical algorithm are used in the ideas proposed in this thesis. There has also been some research into building publish/subscribe systems over a DHT substrate. In order to provide some context for this thesis, we present an overview of some of these systems and touch on the limitations and differences of these implementations with this thesis. We also point out some research projects that have attempted to build a database over a DHT substrate. While publish/subscribe and databases are different problems, some of the techniques used to implement these systems over a DHT are similar.

3.1 Distributed publish/subscribe

There is much distributed publish/subscribe research in networks that require a dedicated broker network [2, 7, 8, 9]. Here, we focus on the distributed matching and multicast aspects of these algorithms as opposed to the matching algorithms at a node.

Consider the hierarchical broker network in Figure 3.1. In this network, subscriptions from subscribers are sent towards the root broker. Every broker along the path of a subscription stores the subscription and the address of the node the subscription was received from.

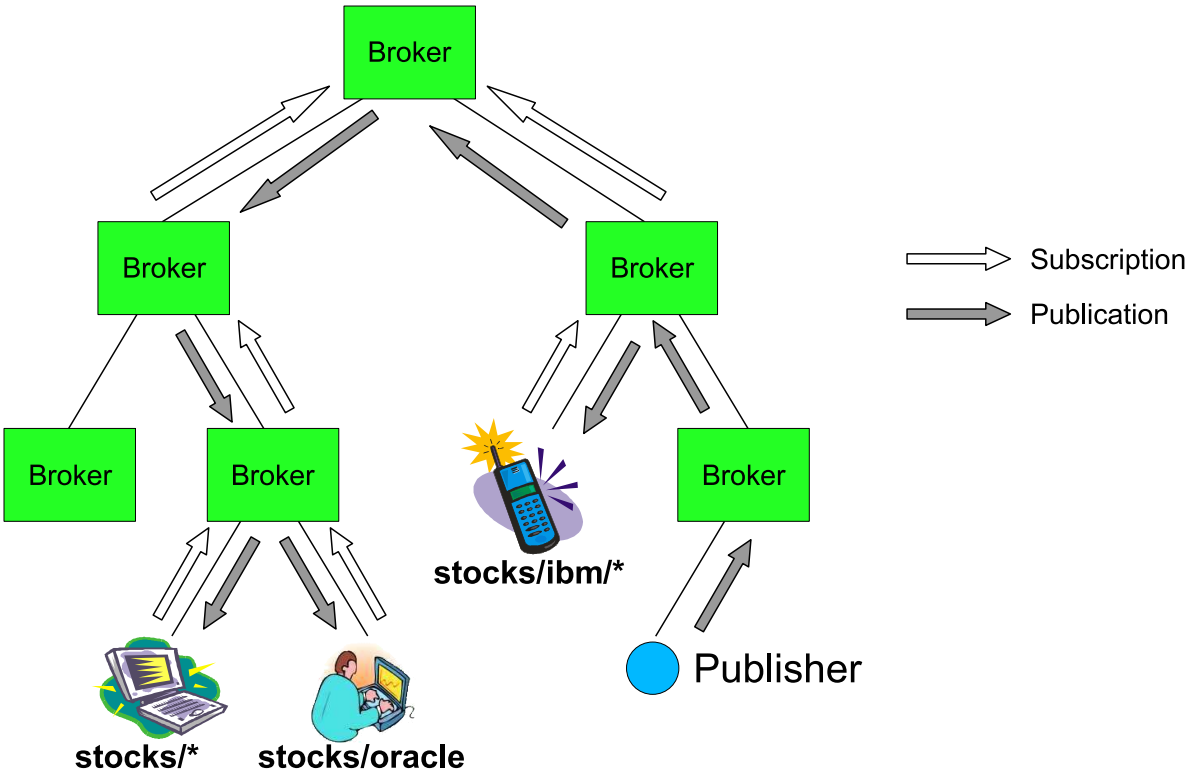

Figure 3.1: Subscription and publication propagation in a hierarchical broker network

Subscription *covering* is used to quench subscription propagation. A subscription s_1 covers subscription s_2 if the set of publications p_1 that match s_1 is a superset of the set of publications p_2 that match s_2. A broker that has previously seen and forwarded s_1 towards the root does not need to forward s_2; the broker simply stores s_2. This covering optimization reduces network traffic and the size of the subscription tables in the upstream brokers.

Publications from publishers are also sent towards the root broker. Each broker that receives the publication also checks for subscriptions in its subscription table that match the publication, and sends a copy of the publication downstream towards those matching nodes. Notice that publications follow the reverse path of the subscriptions; the subscriptions build the *multicast* tree over which the publications are sent. This algorithm has two key properties: the matching of publications to subscriptions is distributed among several brokers, and publications are *multicast*. Multicasting ensures that the same publication is not sent over the same

link twice.

The algorithm described above suffers from a root node that can become overloaded. An optimization that uses *advertisement* messages addresses this problem. A publisher sends an advertisement that specifies the set of the publications that it intends to publish. For example, a publication may send an advertisement that indicates that it will only publish IBM stock quotes. An advertisement is sent towards the root node. Subscriptions then flow up towards the root and down the reverse path of matching advertisements. A subscription s matches an advertisement a if the set of publications p_a specified by a intersects the set of publications p_s that match s. Note that the reverse path of the subscriptions has now built a tree from each publisher to the subscribers potentially interested in the publications published by the publisher. Publications can now follow this multicast tree without unnecessarily traversing to the root broker.

More generalized topologies are also possible with slight modifications to the propagation of the messages. For example, the flooding of advertisements can be used to build an acyclic network graph. Subscriptions and publication now propagate as described above.

3.2 Publish/Subscribe and DHTs

In this section, we discuss publish/subscribe systems that are implemented over a DHT substrate.

Scribe

Scribe [37] is a *channel-based* publish/subscribe system built over the Pastry DHT. In channel-based publish/subscribe subscribers belong to a named channel and publications are sent to all subscribers in a channel. This is not as expressive as the content-based publish/subscribe which allows subscribers to specify finer-grained interest.

Scribe treats a channel name c as a key in the DHT. Recall that the DHT assigns this key to a peer r in the network, where r is called the channel root peer. A subscription s is sent

towards r. As in traditional publish/subscribe algorithms, peers in the path of s store s and the address of the peer that sent s in order to build a multicast tree from the channel root r to the subscribers. Publications are also sent to the channel root, and then follow the multicast tree to the subscribers in the channel.

Hermes

Hermes [38] is a content-based publish/subscribe system built over the Pastry DHT. It essentially assigns a channel to each publication and subscription and the matching algorithm degenerates to that of Scribe. Note that Hermes offers content-based publish/subscribe delivery semantics but the multicast trees and publication propagation are similar to Scribe.

Terpstra

This algorithm [39] builds a content-based publish/subscribe system over the Chord DHT. The algorithm creates a separate multicast tree rooted at each broker. Each tree has logarithmic bounds on its depth, and multiple trees help distribute network traffic among the nodes, and remove bottlenecks and single points of failure. The multicast trees are built by essentially flooding subscriptions, with flooding quenched by covering and merging the subscriptions. The flooding of subscriptions is drastic. The claim that flooding will be heavily attenuated by subscription covering and merging is unconvincing.

Tam/Azimi/Jacobsen

Tam et al. [40] attempt to map a content-based publish/subscribe system to a channel-based one as in Scribe. The algorithm requires a globally known schema that specifies the attribute names, types, and values in the system. Additionally, several indices, each consisting of strategically chosen attributes, must be specified. Each publication and subscription is mapped to several index digests, one for each index. An index digest is a topic name comprised of the concatenation of the name, type, and value of the attributes in the corresponding index. In this

way, content-based publish/subscribe semantics can be achieved from a topic-based one.

The use of indices is unsatisfactory. As shown by the authors, the system performance is very sensitive to the specification of these indices, yet the indices need to be manually specified. Furthermore, manually specified, globally known indices are contrary to the P2P philosophy of minimal administration.

Meghdoot

Meghdoot [41] is a content-based publish/subscribe system built over the CAN DHT. Meghdoot requires a global static schema of the publish/subscribe attributes in the system. For a system with k attributes, Meghdoot constructs a CAN space of dimension $2k$. Subscriptions are mapped to a point in the CAN space and stored the the responsible node. To perform matching, publications traverse all regions with possible matching subscriptions.

We feel that this is the wrong approach: publish/subscribe workloads typically have many more publications than subscriptions, and it does not make sense to optimize subscription state (a subscription is only stored at one peer in Meghdoot) at the expense of publication matching load and delay (publications are routed to multiple peers).

3.3 Databases and DHTs

There have been attempts to build a relational database on top of a DHT P2P network. These attempts are motivated by similar arguments as for developing a publish/subscribe system over a DHT, such as infrastructureless scalability.

However, techniques from these database solutions are not readily applicable to publish/subscribe. The database problem of finding all data that match a specified query is in many ways the dual of the publish/subscribe problem of finding all “queries” (subscriptions) that match a given “data” (publication).

PIER

Peer-to-Peer Information Exchange and Retrieval [20] is a database query engine built over a DHT interface. PIER relaxes traditional relation database semantics such as guaranteed consistency and atomic transactions in order to achieve a massively distributed database.

Range Queries for Grid

A technique to perform efficient database range queries on data stored in a DHT is developed by Andrzejak and Xu [42]. The technique involves assigning intervals of the range being indexed to nodes in the network. A Space Filling Curve, such as the Hilbert curve, is used to globally order the nodes in a d-dimensional CAN DHT such that neighbouring nodes in the CAN have neighbouring intervals. To process queries, requests are routed, in a controlled or brute-force manner, to those nodes whose interval intersects the query range.

PePeR

PePeR [43] assigns ranges to different nodes in the network. Each node stores records that fall within the range indexed by a node. Strategic links are maintained among the nodes to facilitate traversing to a node with the desired range. A potentially weakness of PePeR is that only one attribute can be indexed by this structure.

Chapter 4

Proposed design

There are two main parts of a distributed publish/subscribe algorithm: matching publications with subscriptions and multicasting publications to interested subscribers. This chapter focuses on the former problem in the context of a P2P network.

Distributed matching requires that publications and subscriptions meet at some node in the system. In distributed publish/subscribe systems such as Siena [2], there is an assigned root node at which all publications and subscriptions will eventually meet. (The root node can be a global root as in JEDI [8], or one per publisher as in Siena.) Having such a root node is not appropriate in P2P networks, since the root node must be both powerful and reliable (high availability). It needs to be powerful because, even with the use of advertisements, many publications travel to the root in order to reach subscribers on the other subtree of the root. In addition, the root is a single point of failure.

The topic-based publish/subscribe problem has been addressed in P2P networks without the use of static trees as in Siena [37, 44, 45]. However, the techniques used in topic-based P2P publish/subscribe can't be trivially extended to content-based publish/subscribe in P2P networks. The problems become evident when subscriptions with range predicates are used.

We will describe three distributed publish/subscribe matching algorithms in this chapter: *attribute chains, multidimensional matching,* and a hybrid approach. The remainder of this

report will only consider the hybrid algorithm due to its superior properties that will be outlined in the discussion below.

4.1 Choice of P2P substrate

In Section 1.1, we motivated the benefits of building a publish/subscribe system over a P2P network, but we have not explained why we choose to use a DHT instead of using another (unstructured) P2P substrate or building our own P2P protocol customized and tuned to our publish/subscribe application.

First, the decision to build our software on top of a P2P routing substrate instead of tying a P2P protocol to our application leads to a more modular design. A structured P2P substrate (as opposed to an unstructured one) is used since it has analytically proven and experimentally demonstrated performance guarantees. These guarantees include bounded message cost and state. A DHT substrate is used because it has a simple interface with many implementations. We can choose from a large number of implementations and there is active research on improving these implementations. Furthermore, there are several existing publish/subscribe implementations built over a DHT substrate, so a comparison to these related applications is conceivable.

Despite these reasons, a DHT imposes some constraints that make it challenging to build a publish/subscribe system on top of it. In particular, a hash table is not well suited for performing range queries. It is typically necessary to “walk” the range to find all matching entries in the hash table. This problem is exacerbated when the data items are continuous (floating point) values. Finally, range queries in a distributed system introduce the issues of data placement and query routing.

4.2 Terminology

Key notation and terminology from other parts of this thesis are summarized in this section.

P2P substrate

An underlying DHT routing substrate is assumed. The DHT exposes a hash function $h(key)$ that maps a key (a sequence of bytes) to a $nodeid$. The DHT also has a $send(msg, key)$ function that sends a message to the peer in the network that is "responsible" for the key. Let function $resp(nodeid)$ map a nodeid to the responsible peer.

The notation $p(x) = resp(h(x))$ is used to refer to the peer that a string x is mapped to by the DHT.

An "attribute root" peer is used to denote the peer that is "responsible" for an attribute (in a publication or subscription). The attribute root for an attribute a_i is $p(a_i)$.

Publish/Subscribe semantics

A subscription I (interest) is a conjunction of predicates, where each predicate is a triple of $\{attribute, operator, value\}$. The $attribute$ is a string such as "stock" or "price" that names the predicate. The $operator$ and $value$ denote the constraints of the subscriber's interest. In this report, we restrict the value to be a string or number (integer, floating point, or any other type whose values can be totally ordered). Only equality constraints are allowed for string types, while number types can have operators in the set $\{=, >, <, \geq, \leq\}$.

A publication E (event) is a set of $\{attribute, value\}$ pairs, whose fields have the same restrictions as the corresponding ones in a subscription. Notice that publications have an implicit equality operator on the value.

A $\{a, o, v\}$ predicate covers a $\{a', v'\}$ pair, iff $a = a'$ and the v' falls in the range of values specified by the predicate. A publication E *matches* a subscription I iff every predicate $\{a_i, o_i, v_i\}$ in I covers some $\{a'_j, v_j\}$ pair in E.

4.3 Attribute chains

This section develops the first attempt to develop a publish/subscribe system over a DHT.

Let the peer $p(a_i)$ be called the "attribute root" for attribute a_i, where $p(a_i)$ is the peer to which the underlying DHT overlay maps the (string) name associated with attribute a_i.

Consider a subscriber S with subscription (interest) I and a publisher P with matching publication (event) E. As a concrete example, $I = \{a_1 = \text{“}ibm\text{”}, a_2 > 40\}$, and $E = \{a_1 = \text{“}ibm\text{”}, a_2 = 100, a_4 = \text{“}nyse\text{”}\}$.

The algorithm works as follows.

1. To subscribe to subscription I, subscriber S randomly (for want of better information[1]) chooses an attribute in I. Suppose attribute a_1 is chosen. S sends I towards $p(a_1)$. The underlying overlay network is used to perform this routing.

2. Nodes along the path of I from S to $p(a_i)$ simply forward the subscription towards the attribute root. In particular these nodes do not store any state about I or where it came from.

3. Upon receipt of a subscription I, an attribute root $p(a_i)$ records the subscription along with the id of the subscriber in a table.

4. To publish event E, publisher P sends copies of E to the attribute root of every attribute in E, that is to $p(a_i) \forall a_i \in E$. (An event E sent to $p(a_i)$ should flag attribute a_i so the attribute root knows which attribute to match.)

5. Upon receipt of an event E at an attribute root $p(a_i)$, the attribute root matches subscriptions in its table only on attribute a_i, and sends E to all those subscribers. Note that subscribers may receive false positive events and must perform local matching.

[1] Statistics collected over time on the attributes could be used to make this decision

6. Since matching is only done on one attribute in a subscription, the subscriber will receive many false positives. Over time, the subscriber can examine these false positives to determine the next most selective attribute to filter on. Suppose subscriber S, that is currently subscribed to $p(a_1)$, chooses to next filter on attribute a_2. S sends subscription I to $p(a_2)$ along with a stack of the remaining attributes to filter; in this case this stack consists of attribute a_1. $p(a_2)$ then forwards I to $p(a_1)$. Matching events now flow through a chain from the publisher to $p(a_1)$ to $p(a_2)$ to the subscriber. S also needs to unsubscribe to $p(a_1)$ so as not to receive duplicate events from $p(a_1)$ and $p(a_2)$.

This algorithm ensures that events meet matching subscriptions at exactly one place in the network, namely at an attribute root. To see why, consider any matching publication E and subscription I. According to publish/subscribe matching semantics, the set of attribute $a_i \in E$ must be a superset of $a_j \in I$. So, by subscribing to a single a_j and publishing to every a_i, we ensure that a publication finds every matching subscription (since a_i must be a superset of a_j) and that it is matched only once (since we only subscribe to one a_j). Note that sending subscriptions to every a_j and publications to one a_i will not guarantee matching.

Only the propagation of subscriptions was described above. Unsubscriptions are handled equivalently. Namely, unsubscriptions are propagated in the same way as the corresponding subscription, but removed from the corresponding attribute root's subscription table instead of inserting it there.

Here are some notes on the advantages and disadvantages of this method.

- There is no need for a global schema. A publisher can make up new attributes, and subscribers can subscribe to anything; an ad-hoc publish/subscribe network can arise. This fits well with the P2P philosophy of having no administrative authority define a schema.

- This algorithm distributes computation by having each attribute root only match one attribute at a time. However, it is questionable whether the price we pay in bandwidth by

forwarding to multiple attribute roots is worth it.

- An attribute root needs to know the id of every subscriber to that attribute which can become large. We deal with this problem with the "attribute range delegation" and "covering" variants below.

- After matching at an attribute root, the event is unicast to every matching subscriber. Multicast event propagation is addressed with the "covering" variant below.

- The path from publisher to attribute root to subscriber can be long even if the publisher and subscriber are close. This problem also applies to the paths between attribute roots. One solution to this is to choose a new attribute root that is closer to the publisher and subscriber than the original attribute root, and send messages to the new attribute root instead of the original. This pointer from the original to new attribute root can be cached by publishers and subscribers. However, choosing a new attribute root is not an easy problem in itself.

- It is wasteful for publishers to send events to the attribute root of every attribute in the event. This is especially so if events have many attributes. However, this waste is slightly diminished with more subscribers, as it becomes more likely that publications to all these attribute roots will follow a path to a subscriber.

- Attribute roots for popular attributes can become overwhelmed with events. This is addressed by the Attribute-Value Chain algorithm described below.

- When a subscriber S subscribed to $p(a_1)$ then chooses to filter by $p(a_2)$, the protocol needs to ensure that the subscription from $p(a_2)$ reaches $p(a_1)$ before the unsubscription from S to $p(a_1)$. Also, duplicate events from $p(a_1)$ and $p(a_2)$ must be detected and dropped. Missed publications can be addressed by delaying the unsubscription to $p(a_1)$ for some time after the subscription to $p(a_2)$, and duplicate publications can be detected by checking against a cache of recently received publications.

4.3.1 Attribute range delegation

An attribute root can easily become overloaded with subscriptions consuming both memory and matching computation time. In order to address this, an attribute root for an attribute can delegate portions of the attribute range to other peers.

To illustrate by example, consider an attribute root peer A that is responsible for an attribute foo whose domain is integers in the range $[1, 100]$. If at some point, the number of subscriptions managed by A exceeds some threshold, it may choose to delegate a portion of these subscriptions, say those with queries contained in the range $[1, 50]$, to some peer B. Likewise, B may further delegate subscriptions, creating a delegation tree.

Subscriptions and unsubscriptions that arrive at the attribute root A with a foo attribute constraint that intersects the range $[1, 50]$ immediately get forwarded to B without further processing. Likewise, events that arrive at A and whose foo value might be matched by subscriptions at B are forwarded to B. Note that depending on how subscriptions are delegated, events might have to be forwarded to multiple delegate peers including the current peer.

We note that this delegation does not reduce the number of incoming events at an attribute root. In fact, the total bandwidth increases as events and subscriptions are forwarded to one or more delegates.

An efficient implementation of the above algorithm needs to consider several issues, including those below.

- The threshold number of subscriptions at a peer before it starts to delegate some to other peers. A too low threshold will cause excessive delegation while a large threshold will lead to overloaded peers.

- The subscriptions delegation policy. A good decision here would minimize the need to forward events to multiple children. One should try to keep popular subscriptions (that match many events) close to the root to reduce bandwidth.

- The choice of delegate peers. A delegate peer should probably be close to the parent

network-wise. Also the peer must be powerful enough (bandwidth-wise) to handle the stream of incoming events.

- The delegate revocation policy. It is not clear if a parent peer should repossess subscriptions from a child when the parent has enough resources to do so, or if a child should return subscriptions to the parent when a low water mark of subscriptions is reached at the child?

4.3.2 Covering

The basic Attribute Chain algorithm described above suffers from the severe flaw of the need for unicast event dissemination. To achieve multicast dissemination, we require peers along the path from a subscriber to an attribute root to perform covering and hence also maintain state. The basic algorithm must be modified as follows.

2. Every peer N along the path of subscription I from subscriber S to attribute root $p(a_i)$ remembers I and the peer it received I from. This information is used to build the multicast tree rooted at $p(a_i)$. If $N \neq p(a_i)$ has no existing subscription that covers I on attribute a_i, then N forwards I up towards $p(a_i)$.

3. Upon receipt of a subscription I, an attribute root $p(a_i)$ records the subscription along with the id of the peer it received the subscription from (instead of the subscriber's id).

5. Upon receipt of an event E at a peer (including the attribute root), the peer sends E to those child peers in the multicast tree whose subscription matches E.

While we now gain multicast event dissemination, the system is now more susceptible to faults. In addition to the attribute roots, the state of every peer from subscriber to attribute root needs to be replicated.

4.3.3 Attribute-value root

The attribute root can become overwhelmed with many events. Constructing attribute trees by delegating subscriptions to other peers doesn't help since events still go to the attribute root.

To address this issue, we can use attribute-value roots instead of attribute roots. Subscriptions and events are sent towards the root of an attribute-value. The basic algorithm is modified as follows.

1. To subscribe to subscription I, subscriber S randomly chooses an attribute that has an exact query in I; the exact query can be for either string or numeric types. Suppose attribute a_1 is chosen. S sends I towards $p(a_1 = ibm)$. The underlying overlay network is used to perform this routing.

4. To publish event E, publisher P sends copies of E to the attribute-value root of every attribute in E, that is to $p(a_i = v_i) \forall a_i \in E$. (An event E sent to $p(a_i = v_i)$ should flag attribute a_i so the attribute root knows which attribute to match.)

This approach creates more specialized roots than the basic Attribute Chain algorithm. As such these roots will tend to store fewer subscriptions, (and hence have smaller attribute trees) and will receive fewer events. However, popular attribute-values can still overwhelm a root. Also, this algorithm requires that subscriptions and events have at least one exact query attribute in common.

4.4 Distributed multidimensional matching (DMM)

The attribute chains algorithm matches one attribute at a time. This is inefficient when subscriptions have many attributes. We now develop an a distributed data structure that can match multiple attributes simultaneously.

This algorithm maps the publish/subscribe matching problem to a multidimensional indexing problem. Techniques from existing solutions to the multidimensional indexing problem

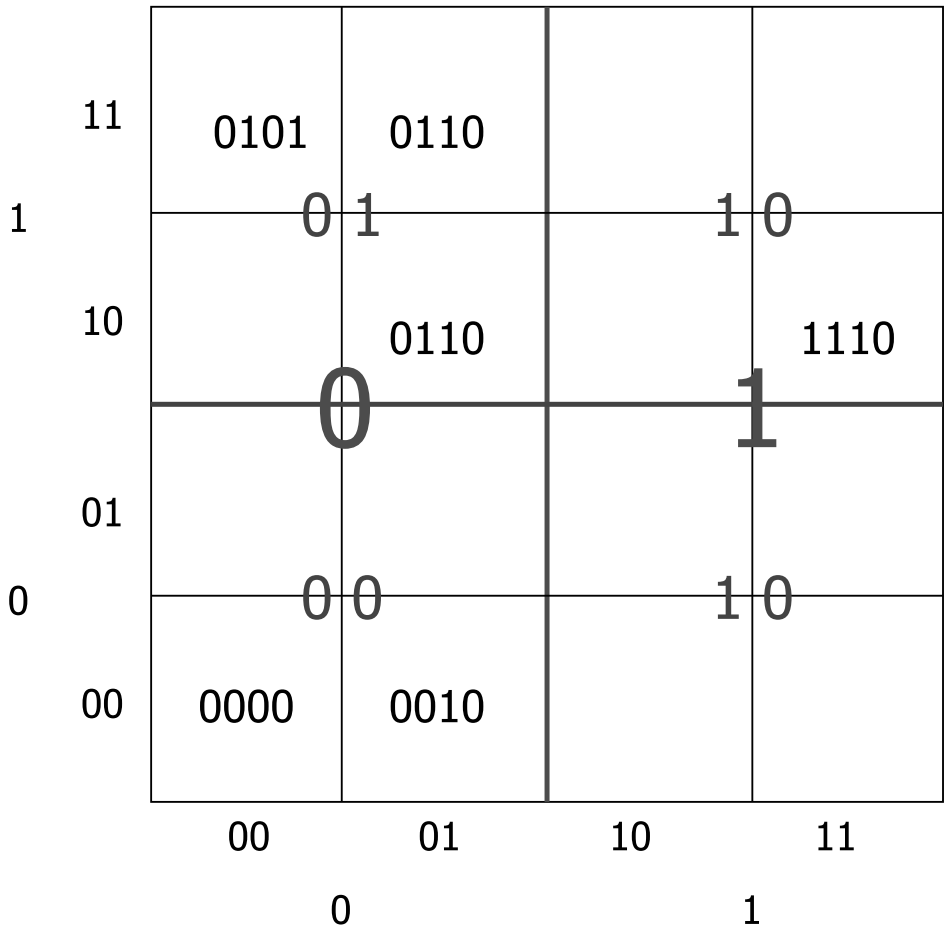

Figure 4.1: Z-code

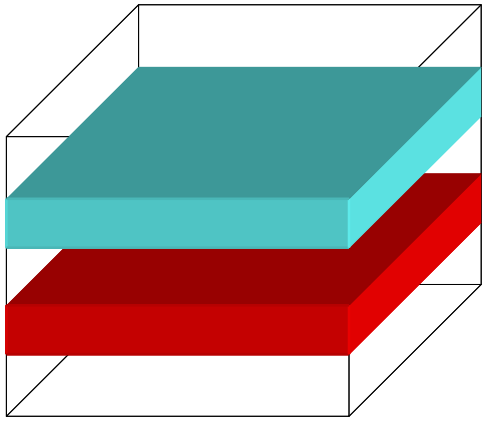

Figure 4.2: Spatial extent of a region

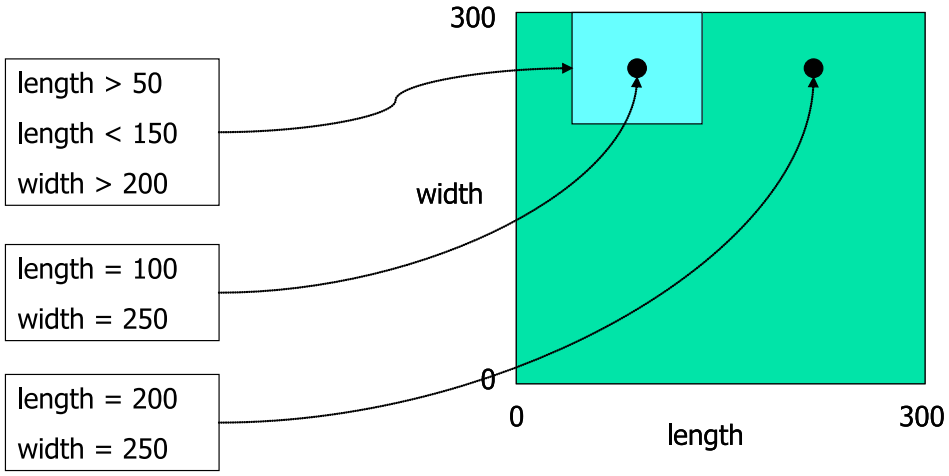

Figure 4.3: Mapping subscriptions and publications to regions

are used to develop a distributed indexing structure. Unlike attribute chains, DMM can match multiple attributes simultaneously.

4.4.1 Assumptions

This algorithm is based on the following assumptions.

- Attribute types must have a known, fixed size domain. For example, a “weight” attribute of type integer must be specified as having values in the range [0,300]. Note that values can have arbitrary precision but must have fixed bounds, so a floating point number can have an arbitrarily large number of digits but cannot have an infinitely large value. In almost all practical applications, data types have fixed bounds (such as a 32 bit integer) so this constraint has minimal practical restrictions.

- There is a known granularity for each attribute value. For example, a “length” attribute might have a granularity of 0.001m. As will be explained later, the granularity is used as a terminating condition in the recursive indexing algorithm. Note that the actual values can be of finer granularity, but the values are rounded (temporarily) to the finest granularity for indexing purposes. Again, in most practical applications we expect an obvious finest

granularity choice. For example, the finest granularity in a 32 bit integer attribute will be one unit.

- String attributes must have a maximum number of characters. There may be variable length string attributes in an event, but these attributes are not matched by DMM (they can be filtered while the event is being multicast as described in Section 5.1). A maximum string length is required because strings, as stated above, we require that attributes have a known a fixed size domain. For example, an attribute with maximum string length of four has a domain whose lowest value string is "a" and highest value "zzzz".

- There is a known global order of the attributes in the system. The global order can simply be the alphabetical ordering of all attribute names.

In addition to the above constraints, we temporarily require that the set of all attribute names and domain is globally known. This is a serious restriction in a decentralized P2P context, but is included to simply the discussion of the algorithm. We remove this constraint in Section 4.5.

The DMM algorithm allows publications to support inequality constraints as in subscriptions. For example, it is possible for the "height" attribute in a publication to have a value > 3. In addition, an attribute's value in a subscription can now be bound to another attribute's value in the subscription. For example, a subscription can express interest for publications with $height > 3$ and $length < 2 * height$.

4.4.2 Mapping publish/subscribe to multidimensional indexing

The mapping from the publish/subscribe domain to a spatial domain for multidimensional indexing is as follows:

- A d dimensional space is created, where d is the number of unique attributes (name and data type) in the publish/subscribe domain.

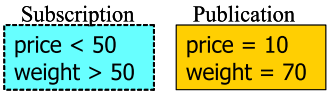

Figure 4.4: Publish/ Subscribe domain

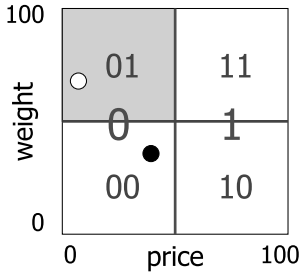

Figure 4.5: Spatial domain

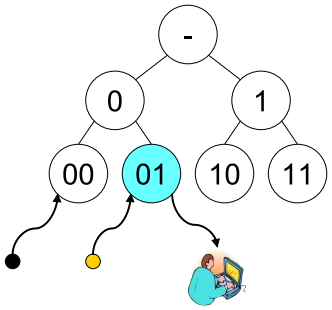

Figure 4.6: Network domain

- Every attribute a_i in the publish/subscribe domain maps to a dimension d_i in the spatial domain.

- An attribute a_i's range in the publish/subscribe domain corresponds to the range of d_i in the spatial domain.

In this model subscriptions and publications using traditional publish/subscribe semantics, map to regions and points, respectively, in the spatial domain. The example in Figure 4.3 shows a space with two integer attributes, *width* and *length* with values in the range [0,300]. A subscription is mapped to a region in the space, and two publications are mapped to points in the space.

We use a multidimensional space where each attribute is assigned to a dimension and the range of a dimension corresponds to the domain of the attribute. As discussed earlier, for string types this requires bounded strings length so that string values can be totally ordered. A subscription is treated as a region in this space, and an event as either a point or a region. For simplicity, in the rest of this discussion we assume events are points rather than regions.

A d-dimensional space S is managed by a binary search tree that represents a recursive subdivision of the universe into subspaces (regions) by means of $(d-1)$-dimensional hyperplanes. The hyperplanes are iso-oriented and their direction alternates among the d possibilities. For $d = 3$, for example, splitting hyperplanes are alternately perpendicular to the x-, y-, and z-axes. Each splitting hyperplane divides a region in half. Each region r has a corresponding

Figure 4.7: Determining the z-code of an integer attribute

Algorithm *IntegerAttributeToZCode*($lval, uval, lbnd, ubnd$)
($*$ Return the z-code of an integer with value [lval,uval] and bounds [lbnd,ubnd] $*$)
1. $l \leftarrow lbnd$
2. $u \leftarrow ubnd$
3. $zc \leftarrow$ **nil**
4. $lastIter \leftarrow$ **false**
5. **repeat**
6. $\quad m \leftarrow \frac{l+u}{2}$
7. $\quad$ **if** $lval \leq m \wedge uval \leq m$
8. $\quad\quad$ **then** $u = \lfloor m \rfloor$
9. $\quad\quad\quad zc \leftarrow zc.Append(0)$
10. $\quad\quad$ **else if** $lval > m \wedge uval > m$
11. $\quad\quad\quad$ **then** $l \leftarrow \lceil m \rceil$
12. $\quad\quad\quad\quad zc \leftarrow zc.Append(1)$
13. $\quad\quad\quad$ **else stop** ($*$ Doesn't fit in either half, so stop. $*$)
14. $\quad lastIter \leftarrow l == u$
15. **until** $lastIter$
16. **return** zc

node $n(r)$ in the search tree.

Each region is addressed by a bit string, called a z-code, and is associated with one node in the tree. Consider a $d = 2$ space where we alternate between vertical and horizontal splitting hyperplanes (lines in this space). A region r in this space is assigned a z-code as follows. If r lies to the left of the first subdividing hyperplane, the first bit of the corresponding z- value is 0; otherwise it is 1. In the next step, we subdivide the subspace containing r by a horizontal line. If r lies below that line, the second bit of the z-code is 0; otherwise it is 1. As this decomposition progresses, we obtain one bit per splitting line. Bits at odd positions refer to vertical lines and bits at even positions to horizontal lines. Figure 4.1 shows the z-codes in a two-dimensional space. An interesting property resulting from such bit interleaving is that if a z-code z_1 is the prefix of another z-code z_2, then z_1 encloses (or covers) z_2. Figure 4.7 presents the algorithm to convert an integer to a z-code, and Figure 4.8 shows the algorithm to determine the z-code of a set of attributes that make up a space.

A subscription (object) s is stored at all the leaf nodes $n(r_i)$ in the search tree such that r_i intersects s. Thus the insertion or deletion of a subscription requires the traversal of multiple paths from the root to leafs of the tree. An event (point) e will find matching subscriptions by

Figure 4.8: Determining the z-code of a set of attributes

Algorithm *AttributesToZCode*($attrs$)
($*$ Return the z-code of the specified attributes $*$)
1. ($*$ Determine the minimum z-code of all attributes. $*$)
2. $minlen \leftarrow MinZCodeLength(attrs)$
3.
4. ($*$ Weave zcodes into one. $*$)
5. $zc \leftarrow$ **nil**
6. **for** $i \leftarrow 0$ **to** $minlen - 1$
7. **do for** $attr \in attrs$
8. **do** $z \leftarrow ZCodeOf(attr)$
9. $bit \leftarrow GetBit(z, i)$
10. $zc \leftarrow zc.Append(bit)$
11. **return** zc

traversing a single path from the root to a leaf. The matching subscriptions are stored at the leaf node.

The splitting/merging of regions is done dynamically. If the number of subscriptions stored at a node $n(r)$ reaches some threshold, then region r is split into r' and r'', and new nodes $n(r')$ and $n(r'')$ are created. The z-code of the new region r' (r'') is the z-code of r with bit 0 (1) appended.

The described structure uses a *clipping-based scheme* (such as R+-trees and extended k-d-trees) where we do not allow overlaps between bucket regions. Furthermore, we partition the space in a systematic way (similar to a grid file), using *z-codes* to address regions (such as in a GBD-tree). We will see that this allows peers to access nodes in the structure without traversing through the root. The trade-off for this is having inserts and deletes traverse multiple paths and stored at multiple nodes in the tree, and having an unbalanced tree.

The alternative of using *overlapping regions* (such as R-trees, R*-trees, GBD-trees) results in more efficient inserts and deletes. But we lose out by having a more complex partitioning scheme that requires queries to traverse from the root. Structures with overlapping regions also require queries to traverse multiple paths in the structure. Since we expect queries (publications) to occur more often than inserts (subscriptions), this is not an acceptable tradeoff.

4.4.3 Mapping multidimensional indexing to a DHT

This section discusses the mapping from the spatial domain to the network domain. In other words, we attempt to distribute the indexing structure described above.

Each region r with z-value z has a corresponding node $n(r)$ in the tree. The information of each node is stored at the peer $p(r)$ in the network. The peer $p(r)$ is the one where the underlying DHT would store key z. Note that multiple nodes can map to the same peer. It is important to note that given the z-value of a region r, peers can independently find $p(r)$.

We start out with a single root peer $p(S)$ for the entire space. In order for both publishers and subscribers to find this root, $p(S)$ can be the hash of the attributes in the system (which information is known to all peers).

Subscriptions are sent to the root peer $p(S)$ and flow down to the appropriate leaf nodes. As an optimization, a subscription s can be sent to the peer $p(r)$ where region r is the smallest region that encloses s, and then allowed to traverse up or down to leaf nodes. This reduces the load on the root peer. Note that if the search tree has not been recursively subdivided enough that $n(r)$ is in the tree, then $p(r)$ will need to propagate the subscription to its first ancestor node that does exist in the tree. This is further discussed in the event propagation below.

An event e too can be sent to $p(S)$ and traverse down to find matching subscriptions, but we can do better. Assuming we have a finest granularity of regions, we can find the smallest region r that encloses e, and send e to $p(r)$. Assuming a fine granularity of r, it is very likely that $n(r)$ doesn't exist in the binary search tree. So, $p(r)$ must now forward e to its first ancestor $p(r')$ in the tree, where r' encloses r, $n(r')$ is in the tree, and there is no $n(r'')$ in the tree such that r'' encloses r.

One way for $p(r)$ to find $p(r')$ is by recursively forwarding to parent peers until we reach $p(r')$. This will take $O(\log |S|)$ overlay hops, where $|S|$ is the maximum number of regions in space S for the given granularity of regions. An optimization would start with large traversals of the tree and make smaller hops as we get closer to the destination node. This can be accomplished by forwarding to the ancestor node whose depth is half that of the current node. Doing

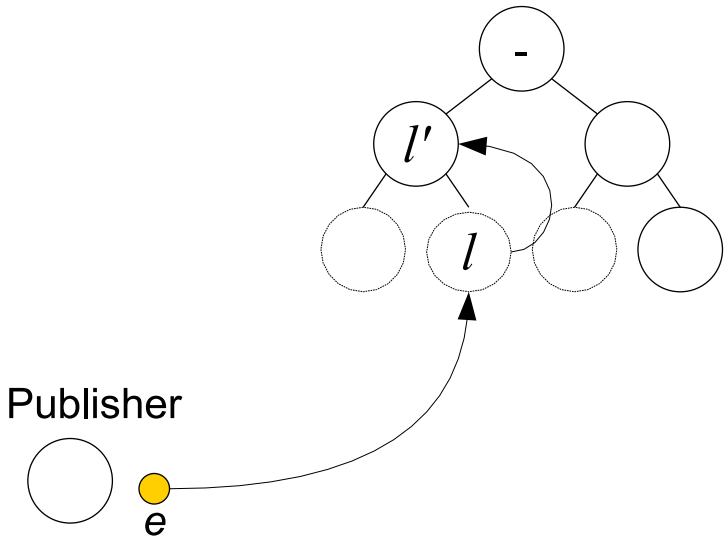

Figure 4.9: Publication traversal

this might overshoot to an ancestor of $p(r')$ requiring us to traverse down the tree again.

This traversal of a publication is illustrated in Figure 4.9, which shows a space with four existing nodes. The publication e is sent from the publisher towards a (non-existent) leaf node l in the tree. As this node does not yet exist in the tree, the publication e if forwarded up the tree until an existing leaf node l' is reached. Any subscriptions that match e will be found at node l'.

An example of a complete mapping from the publish/subscribe to spatial to network domain is shown in Figures 4.4, 4.5, 4.6, respectively.

4.4.4 Algorithm

We now present the pseudo-code to handle publication and subscription transmissions. Table 4.1 shows the details of these messages.

The propagation of a subscription goes through two state. First, the subscription goes through the *finding tree* state in which it travels towards the DMM tree. In this state, every peer along the subscription path stores an entry in its subscription table; the reverse path of these

MSG	Any message below
m.sender	Peer that sent the message
SUB	Subscription message
s.sbr	Subscriber
s.spc	The subscription space (i.e. the attributes)
s.zc	The current z-code
s.stt	Subscription state
s.GetKey(n)	Get the DHT key for this sub with an n-bit z-code
PUB	Publication message
p.pbr	Publisher
p.spc	The publication space (i.e. the attributes)
p.zc	The current z-code
p.stt	Subscription state
p.GetKey(n)	Get the DHT key for this sub with an n-bit z-code
p.chain	Sequence of peers that have seen this publication

Table 4.1: Messages

subscriptions is used to build the multicast tree. Once the subscription has found a node in the tree, it then goes into the *finding leaf* state. In this state, the subscription travels up or down the tree searching for an existing leaf node; the subscription is not stored in the subscription table in this state until it reaches an existing leaf node. The pseudo-code for this procedure is shown in Algorithm *ReceiveSubscription*.

Algorithm *ReceiveSubscription*($sub, next$)
($*$ Store and foward the subscription $*$)

```
1.  TreeTableEntry tree = GetTree(SpaceOf(sub))
2.  iAmDest = next == null
3.  iAmRoot = NextHop(sub.GetKey(0)) == null
4.  iBelongToTree = tree ≠ null ∧ tree.GetMyZCode(sub.zc) ≠ null
5.  iKnowAboutTree = tree ≠ null ∧ tree.NumKnownZCodes > 0
6.  childrenExist = tree ≠ null ∧ tree.ChildrenExistForThatMatch(sub.zc, sub.spc.zc)
7.
8.  if iAmRoot ∧ ¬iBelongToTree ∧ sub.zc.Length == 0
9.    then (* Make sure tree exists if I am the root and this sub is meant for the root. *)
10.         tree = CreateTree(sub.spc, ZCode(""))
11.         iBelongToTree = true
12.
13. if sub.stt == FindingTree
14.   then (* Sub hasn't reached the DMM tree yet. *)
15.        if iBelongToTree ∧ iAmDest
16.          then (* I am the desired node in the tree. *)
17.               if childrenExist
18.                 then (* I have children. *)
19.                      (* Go down the tree to find a leaf node. *)
20.                      sub.stt = FindingLeaf
21.                      if sub.zc == sub.spc.zc
```

```
22.                         then (∗ Forward to both children. ∗)
23.                             origzc = sub.zc
24.                             sub.zc = LeftChildOf(origzc)
25.                             SendSubscription(sub)
26.                             sub.zc = RightChildOf(origzc)
27.                             SendSubscription(sub)
28.                         else (∗ Forward to one child. ∗)
29.                             IncrementZCode(sub)
30.                             SendSubscription(sub)
31.                 else (∗ I am the leaf. ∗)
32.                     (∗ Store sub here. ∗)
33.                     sub.stt = FindingLeaf
34.                     subTable.Store(sub)
35.
36.         if ¬iBelongToTree ∧ iAmDest
37.           then (∗ I am a tree node but don't belong to the tree yet. ∗)
38.                 (∗ Go up the tree to find a living leaf node. ∗)
39.                 sub.stt = FindingLeaf
40.                 DecrementZCode(sub)
41.                 SendSubscription(sub)
42.           else (∗ I am an intermediate node. Store sub here. ∗)
43.                 hasCovering = subTable.HasCovering(sub)
44.                 subTable.Store(sub)
45.                 sub.sbr = MyAddress
46.                 if hasCovering
47.                   then (∗ Stop propagating sub. ∗)
48.                       next = null
49.
50. if sub.stt == FindingLeaf
51.   then (∗ Only destination nodes process this type of message. ∗)
52.         if iAmDest
53.           then if ¬iBelongToTree
54.                   then (∗ I am a tree node but don't belong to the tree yet. ∗)
55.                       (∗ Go up the tree to find a leaf node. ∗)
56.                       DecrementZCode(sub)
57.                       SendSubscription(sub)
58.                   else (∗ I belong to the tree ∗)
59.                       if childrenExist
60.                         then (∗ I have children. ∗)
61.                             (∗ Go down the tree to find a leaf node. ∗)
62.                             if sub.zc == sub.spc.zc
63.                               then (∗ Forward to both children. ∗)
64.                                   origzc = sub.zc
65.                                   sub.zc = LeftChildOf(origzc)
66.                                   SendSubscription(sub)
67.                                   sub.zc = RightChildOf(origzc)
68.                                   SendSubscription(sub)
69.                               else (∗ Forward to one child. ∗)
70.                                   IncrementZCode(sub)
71.                                   SendSubscription(sub)
72.                         else (∗ I am the leaf. Store sub here and stop. ∗)
73.                             subTable.Store(sub)
74.
75. return next;
```

Publication propagation is similar to that of subscriptions but has three states: *finding tree*, *finding leaf*, and *multicasting*. Every hop of a publication in the finding tree state looks for subscriptions in the subscription table that matches the publication. If one or more matches exist, a copy of the publication is made and sent to matching subscribers. This publication copy goes into the multicasting state, while the original continues in the finding leaf state. As with subscriptions, once a publication reaches a node in the DMM tree, it goes into the finding leaf state to search for an existing leaf. No multicasting is done in the finding leaf state until the publication reaches an existing leaf node. The pseudo-code for this procedure is shown in Algorithm *ReceivePublication*.

Algorithm *ReceivePublication*($pub, next$)
(∗ Store and foward the publication ∗)
1. TreeTableEntry tree = GetTree(SpaceOf(sub))
2. iAmDest = next == null
3. iAmRoot = NextHop(pub.GetKey(0)) == null
4. iBelongToTree = tree $\neq$ null $\wedge$ tree.GetMyZCode(pub.zc) $\neq$ null
5. iKnowAboutTree = tree $\neq$ null $\wedge$ tree.NumKnownZCodes ¿ 0
6. childrenExist = tree $\neq$ null $\wedge$ tree.ChildrenExistForThatMatch(pub.zc, pub.spc.zc)
7. bool seenBefore = pub.chain.Contains(MyAddress)
8.
9. **if** iAmRoot $\wedge$ $\neg$iBelongToTree $\wedge$ pub.zc.Length == 0
10. **then** qcomMake sure tree exists if I am the root and this pub is meant for the root.
11. tree = CreateTree(pub.spc, ZCode(””))
12. iBelongToTree = true
13.
14. **if** pub.stt == FindingTree
15. **then** (∗ Sub hasn't reached the DMM tree yet. ∗)
16. (∗ Multicast to any children with matching subscription. ∗)
17. **if** seenBefore
18. **then** (∗ Stop if I've seen this before ∗)
19. **stop**
20.
21. pub.chain.Append(MyAddress)
22. ForwardToMatchingChildren(pub)
23. pub.pbr = MyAddress (∗ act like the publisher from now on. ∗)
24.
25. **if** iBelongToTree $\wedge$ iAmDest
26. **then** (∗ I am the desired node in the tree. ∗)
27. **if** childrenExist
28. **then** (∗ I have children. ∗)
29. (∗ Go down the tree to find a leaf node. ∗)
30. pub.stt = FindingLeaf
31. **if** pub.zc == pub.spc.zc
32. **then** (∗ Forward to both children. ∗)

```
33.                                 origzc = pub.zc
34.                                 pub.zc = LeftChildOf(origzc)
35.                                 SendPublication(pub)
36.                                 pub.zc = RightChildOf(origzc)
37.                                 SendPublication(pub)
38.                             else (∗ Forward to one child. ∗)
39.                                 IncrementZCode(pub)
40.                                 SendPublication(pub)
41.                     else (∗ I am the leaf. ∗)
42.                         (∗ Nothing else to do. ∗)
43.
44.         if ¬iBelongToTree ∧ iAmDest
45.           then (∗ I am a tree node but don't belong to the tree yet. ∗)
46.                (∗ Go up the tree to find a living leaf node. ∗)
47.                pub.stt = FindingLeaf
48.                DecrementZCode(pub)
49.                SendPublication(pub)
50.
51.         if ¬iAmDest
52.           then (∗ Nothing special to do. ∗)
53.
54. if pub.stt == FindingLeaf
55.   then (∗ Only destination nodes process this type of message. ∗)
56.         if iAmDest
57.           then if ¬iBelongToTree
58.                   then (∗ I am a tree node but don't belong to the tree yet. ∗)
59.                        (∗ Go up the tree to find a leaf node. ∗)
60.                        DecrementZCode(pub)
61.                        SendPublication(pub)
62.                   else (∗ I belong to the tree ∗)
63.                        if childrenExist
64.                          then (∗ I have children. ∗)
65.                               (∗ Go down the tree to find a leaf node. ∗)
66.                               if pub.zc == pub.spc.zc
67.                                 then (∗ Forward to both children. ∗)
68.                                      origzc = pub.zc
69.                                      pub.zc = LeftChildOf(origzc)
70.                                      SendPublication(pub)
71.                                      pub.zc = RightChildOf(origzc)
72.                                      SendPublication(pub)
73.                                 else (∗ Forward to one child. ∗)
74.                                      IncrementZCode(pub)
75.                                      SendPublication(pub)
76.                          else (∗ I am the leaf ∗)
77.                               if ¬seenBefore
78.                                 then pub.chain.Append(MyAddress)
79.                                      ForwardToMatchingChildren(pub)
80.                                      pub.pbr = MyAddress (∗ act like the publisher. ∗)
81.
82. if pub.stt == Multicasting
83.   then if iAmDest ∧ ¬seenBefore
84.           then pub.pbr = MyAddress (∗ act like the publisher. ∗)
85.                pub.chain.Append(MyAddress)
86.                ForwardToMatchingChildren(pub)
```

4.4.5 Discussion

By mapping publish/subscribe matching to multidimensional indexing, the matching algorithm in DMM attempts to match all attributes in a publication simultaneously.

The DMM structure described above is a relatively poor data structure when compared to other centralized multidimensional indexing structures, in terms of an unbalanced tree and duplicate storage of subscriptions at multiple nodes in the tree. However it has some desirable properties when used for distributed publish/subscribe matching. For example, it is undesirable that object (subscription) insertion requires the traversal of multiple paths from the root and storage at multiple leaves, made worse by the possibility that the tree can become very unbalanced. However, DMM allows queries (publications) to traverse up the tree and not create a hotspot at the root node. (Root hotspots are not a problem in centralized indexing structures where upper nodes in a tree can be cached in main memory and access to them is essentially free. This is not the case in a distributed system.) Also, the bottom-up traversal of queries do not suffer from an unbalanced tree. This is an acceptable tradeoff since it is generally accepted that publish/subscribe applications typically have a much higher rate of publication than subscription.

The network distance between a parent p and child c in the tree can be large. This is a consequence of the DHT hash function that can map similar z-codes to distant peers. Another possible solution to this problem is for peer c to store a pointer to a peer c' that is closer to p. Thus, when p traverses down the tree to access c it will be forwarded to c' instead. This pointer can be cached at p for future accesses to c.

A serious problem with the DMM is the use of one high-dimensional space that requires knowing all attributes in the system. Semantically, this is not acceptable in a P2P environment where global knowledge is difficult to distribute. It also restricts the publish/subscribe application from adding new attributes over time. From a technical perspective, the performance of indexing structures tends to degenerate rapidly as the number of dimensions grows very large. Intuitively, a reason for this is that as the number of dimensions in the space grows, the dis-

parity between the dimensionality of the objects stored in the space and the dimension of the space itself increases. For example, consider a one dimensional object that is stored in a three dimensional space. The spatial extent of this object in the space will make a large "slice" in the space as shown in Figure 4.2. It is difficult to index such large "slices" efficiently, since most indexing structures rely on being able to recursively subdivide the space into regions and store objects in as few regions as possible. This general problem is known as the *curse of dimensionality*. We try to address this problem with the hybrid algorithm in Section 4.5.

The use of a multidimensional space makes the CAN DHT seem like a natural substrate to use. Recall that unlike Chord and Pastry's identifier circle, CAN organizes peers in a multidimensional space. However, as will be described in Section 4.5, we do not want to restrict the system to a fixed set of global attributes. Thus CAN provides no obvious inherent advantage over the other substrates.

4.5 DMM with attribute roots (DMM-AR)

This algorithm tries to achieve the superior matching capabilities of the distributed multidimensional access structure without needing a global schema and without the problems associated with high dimensional spaces.

In this approach, we create a multidimensional space S for every combination of attributes in the system. Note that these spaces are created dynamically as needed when the system sees new attributes, so we don't need to pre-specify all the attributes in the system. For instance, a space consisting of attributes a_1 and a_2 is constructed only when a subscription with exactly those attributes is seen. In addition, as will be described below, the use of soft state ensures that these spaces are automatically garbage collected when they are no longer needed. In each S with $d > 1$ we choose an attribute a_x to be the "primary" attribute of S. A node $n(r)$ of S is mapped to a peer by using the underlying DHT to hash the concatenation of the names of the attributes of S and the z-value of r. The root node of S has a null z-value.

Consider a subscription s with attributes a_1 and a_2. We construct our multidimensional structure with a $d = 2$ space S for these attributes, with say a_1, as the primary attribute. Subscription s is stored in this structure as described earlier. The node in space S that ends up storing subscription s then sends s to the one-dimensional space S_1 consisting of attribute a_1. This ends up creating a subscription chain from the subscriber to some node in structure S to some node in structure S_1.

Events are sent to the one-dimensional space for each attribute in the event. Consider event e with $a_1 = v_1$ and $a_3 = v_3$. The publisher calculates the smallest region r_1 in the space S_1 that encloses v_1 (based on the pre-specified granularity), and sends e to $p(r_1)$. This will result in e being sent to the node in space S_1 that contains subscriptions whose a_1 attribute matches e. This node in turns sends e to the node in space S that contains subscriptions whose a_1 and a_2 attributes match e. This node then finally multicasts e to the known subscribers. The publisher, also sends e to some $p(r_2)$, and e might be forwarded to some other subscribers from here.

4.5.1 Discussion

We have addressed high dimensionality by creating multiple smaller dimensional spaces. However, we pay the price for this by requiring an extra level of distributed matching though a one-dimensional space. As well, there is now a large publication fanout at the publisher, since it needs to send one copy of the publication for each attribute in the publication. Also, the multicast trees will not be as good as with one large universal multidimensional space; since subscriptions are sent to many different spaces, the subscription paths are not as likely to meet.

Chapter 5

Other design issues

Chapter 4 described a distributed publish/subscribe matching algorithm. We now complete the design by discussing other concerns in a publish/subscribe system such as the need to multicast publications to subscribers, and fault-tolerance issues. We only discuss these issues for the DMM or DMM-AR algorithm.

5.1 Multicast

We borrow multicast techniques from traditional content-based publish/subscribe systems such as Siena [2].

Consider a subscription s from subscriber p_0 that is stored at k peers p_{h_j} in the DMM tree, where $0 \leq j < k$. The path of s from p_0 to p_{h_j} is $p_{0_j}, \ldots, p_{i_j}, \ldots, p_{h_j}$. Note that according to the DMM algorithm *ReceiveSubscription*, $p_{i_j} = p_{i'_{j'}}$ for every $0 \leq i, i' < h$ and $0 \leq j, j' < k$. That is, s follows a single path until the hop before p_{h_j} at which point the path fans out to each p_{h_j}. So, without loss of ambiguity, p_i is used to refer to any p_{i_j}.

To achieve multicast, every peer p_{i_j} where $0 < i \leq h$ and $0 \leq j < k$ stores (s, p_{i-1_j}) in a subscription table T_S to remember the peer that sent it the subscription. These tables build a multicast tree from a DMM tree node to all subscribers that have sent an event to the tree. When an event e is received at a peer p_{i_j}, it forwards e to all peers p such that (s, p) is in T_S

and event e matches subscription s.

5.2 Fault-tolerance

Resilience to network failures is important in dynamic P2P networks. Network failures include communication failures, such as dropped packets, and crash failures due to a peers either voluntarily or involuntarily disconnecting from the network. Communication failures can be addressed for the most part by the use of acknowledgement messages or by creating logical reliable connections between peers. The latter can be accomplished by, for example, building TCP connections between peers. In this section we are more concerned with the "failures" due to peers entering and leaving the network. Node arrivals are treated as failures in this context as they can lead to temporary incorrect routing tables and imperfect routing.

In a distributed publish/subscribe system, the only state that needs to be maintained are the subscription tables. In the DMM algorithms, this includes both the subscriptions in the DMM tree which are use for matching publications with subscriptions, and subscriptions used to build a multicast tree which are used to direct publications towards subscribers.

The standard technique to maintain state in the face of peer failures is to replicate the state among several nodes. Some important design decisions include the choice of replicas, the failure detection mechanism, and the failure recovery algorithm. We leverage the DHT node identifier circle in our fault tolerance algorithm.

Replica selection

We employ a simple replica selection algorithm: the replicas of a peer are chosen to be its r successor and predecessor peers in the identifier circle, where r is the number of replicas per peer. For example, in Figure 5.1, with $r = 2$, q's replicas are p and r. Recall that nearby nodes in the identifier circle are likely to be geographically dispersed. This leads to the desirable property that a replica is likely to survive localized network failures, such as a network parti-

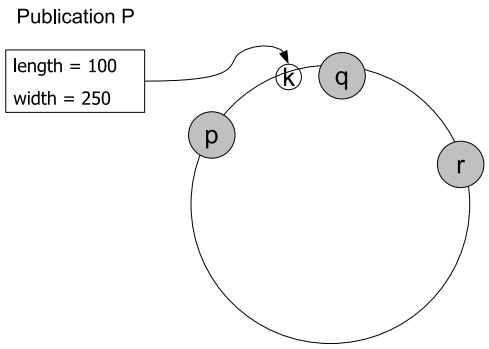

Figure 5.1: Replicas

tion. However, the tradeoff for this is that messages between between the primary peer and its replicas in order to keep synchronize state need to traverse large network distances, leading to increased latency and message load on the network.

Failure detection

The failure of a peer can be detected in two ways. First, if a replica does not receive a periodic beacon from the primary, the replica can assume that the primary has failed, and can take over for the primary. (It is also possible that the primary has not failed but that the replica has lost network connectivity. We will see below that the takeover algorithm ensures that the replica does not incorrectly take over for the primary in such a case.) This technique requires a tradeoff between the beaconing frequency and speed of failure detection. However, there is a cheaper way to detect a failure. Consider peer q in Figure 5.1. Suppose q is the node in the DMM tree that a publication P would normally be sent to. That is, the hash of the attribute names in P along with the z-code of P result in a node id k that is closest to q. Now, if q fails, P would get sent to p, the next closest node to k. By comparing the distance between k and the node ids of p and q, peer p can determine that P should have been sent to peer q. Therefore, the receipt of publication P at peer p tells p peer q has failed, and furthermore that p is the replica that should take over for q. Notice that in this algorithm, the failure of a peer is detected in "real-time" as a consequence of a message arriving at an unexpected peer.

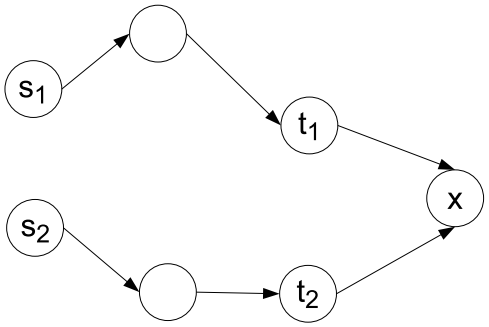

Figure 5.2: Subscription soft state

It is important to note that this failure detection algorithm can also handle peer arrivals. Consider a new peer a whose node id falls between that of p and q. A publication or subscription that would originally be sent to q that is now received by a will cause a to trigger a takeover for some of q's subscriptions. As will be described below, peer a will only takeover those subscriptions that belong to a, leaving q with the subscriptions that should belong to it.

The failure detection process above requires that there is at least two replicas—one successor and one predecessor in the identifier circle. This will ensure that any message originally intended for the primary will be sent to one of the replicas.

Replication data

It was noted earlier that only subscription table state needs to be replicated. However, even this is not necessary when we realize that the subscription table at a peer can be rebuilt from that of other peers, which in turn can eventually be rebuilt from the subscriptions at the original subscribers. For example consider subscriptions from subscribers s_1 and s_2 to the space root peer x in Figure 5.2. If peer x fails, its replica can recreate x's state from the subscriptions at peers t_1 and t_2, that is from x's children in the multicast tree (not its children in the DMM tree). Therefore, only the addresses of the peers from which x received its subscriptions need to be replicated. This information is cheaper to store and transmit than the full subscriptions themselves. Also, due to the route convergence properties of most DHT implementations, subscriptions from all over the network tend to arrive at a peer through a small set of neighbours.

Therefore the children of a peer do not change nearly as often as the subscriptions stored at the peer, so it is cheaper to keep replicas synchronized with the primary peer.

Failure recovery

When a replica p detects that a primary peer q has failed, it needs to recover the subscription state at q. It does this by requesting all of q's children to resend their subscriptions. Note that it is not necessary so that all of q's children will send their subscriptions to p. Some subscriptions may be sent to a peer r that is closer to the hash of the z-code and attribute names of a subscription. This automatically ensures that subscriptions are migrated to the correct peers after a failure. Storing the actual subscriptions at a replica would have complicated the process of deciding which replicas should takeover for which subscriptions.

In summary, the fault tolerance algorithm described above is a light-weight replication scheme that detects failures based on the presence of an unexpected message receipt. We call this active failure detection, as opposed to the passive failure detection used by the absence of periodic beacons from the primary peer.

In our implementation we achieve fault tolerance using both the active failure detection and recovery of the replica scheme described above and the passive failure recovery of periodically beaconing subscriptions. Since the active scheme only recovers to a failure in response to a unexpected message receipt, the passive beaconing approach ensures that subscription state is rebuilt even during prolonged periods of inactivity at a peer. Of course, the use of the replicas allows the beaconing period to be less frequent and still achieve the fast failure recovery.

5.3 DMM tree cache

A publication sent to DMM tree in Figure 5.3 must traverse from a leaf node l up until it reaches an existing node l'. While this ensures that the root node is not unnecessarily overloaded with messages, the multiple hops that the publication travels can be expensive, especially since each

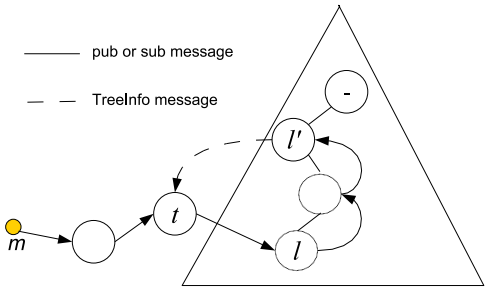

Figure 5.3: TreeCache optimization

traversal from a node in the tree to its parent can take multiple network hops. These issues become more important if the tree is very large and/or there are many publications.

We introduce an optimization that retains the benefits of searching from the leaf up, but alleviates the repeated bottom up traversal. The TreeCache optimization caches information about the DMM tree at various peers in the network. Recall that a publication or subscription traversal can be split into two phases. In the first phase (the finding tree phase), the message travels toward the DMM tree until it reaches a node in the tree. In the second phase (the finding leaf), the message traverses up or down the DMM tree until it finds an existing leaf.

In the TreeCache optimization, a publication or subscription message m stores the final hop t in the finding tree state. When m finally reaches its destination leaf node l' in the DMM tree, the peer corresponding to node l' will send a TreeInfo message to peer t notifying it of the existence of node l'. This information consists of the characteristics of the DMM space (the attribute names and bounds), and the z-code of node l'. Any future publications or subscriptions that traverse through peer t can be directly sent to node l'. In our algorithm, the entries in the tree cache are expired after some time. Thus peers only cache information about nodes to which they have recently sent a publication or subscription.

Stale tree caches do not affect the correctness of the algorithm. If a publication is sent to a node l that no longer exists in the tree, the regular publication handling algorithm will ensure that the publication finds a leaf node.

The experiments in Chapter 6 show that this optimization is effective and has little message

cost.

5.4 Unaddressed issues

There are some issues that are not addressed in this thesis. Some of these problems are difficult to solve and might not have even been addressed in tradition publish/subscribe systems. Other problems are issues that are secondary to the core publish/subscribe task of delivering publications to interested subscribers, and as such are out of the scope of this thesis

Reliability

We distinguish between fault-tolerance and reliability. Fault-tolerance refers to the ability for the system to recover from a fault such that correct operation eventually continues; notably, correct delivery of publications are not guaranteed while recovering from the fault. By reliability on the other we mean the the ability of some peer in the network to know if a given publication has been delivered to all interested subscribers in the system. For instance, in some publish/subscribe algorithms, the publisher can determine if all interested subscribers have received a publication; this allows the publisher to continue to attempt sending the publication until it reaches all interested subscribers.

The proposed algorithm does not provide reliability; no one peer has knowledge of whether all interested subscribers have received a publication. Existing algorithms to achieve this type of reliability require the publisher to know have global knowledge of all interested subscribers. This is feasible in small networks but not in large scale P2P networks. It is also not clear if most applications require such strong guarantees at the expense of the required overhead.

Locality

Traversing up or down the DMM tree is expensive because adjacent nodes in the tree correspond to potentially geographically distant peers. This is a consequence of the DHT hash

function that randomly maps peers onto the identifier circle.

A solution to this is to use a hash function that maps geographically nearby peers to adjacent nodes in the DMM tree. However, this compromises the fault tolerance arguments for randomly distributing nodes around the identifier circle. It is not clear what the impact of such a change on the performance of the underlying routing substrate would be.

Another solution is also possible. Consider a DMM tree with a root node r and a child c. If we wish the child to be geographically close to r, we can choose such a c' and leave a pointer at c to c'. This pointer can be cached at other peers such that a message intended for c is sent to c' instead. We do not pursue this technique because of the complexity in choosing nearby peers as new nodes are added to the DMM tree, and maintaining the pointers to these peers. Moreover, the TreeCache optimization largely addresses the costs of large traversals of the DMM tree.

Encryption, malicious behaviour, and privacy

Publications and subscriptions are sent in clear text. This is an issue that is difficult to solve in a distributed publish/subscribe system where messages are routed based on content; encrypted content makes it impossible to use the current routing techniques. Current solutions to this problem require building elaborate webs of trust that is not appropriate to autonomous P2P networks.

There are also no security guarantees against malicious behaviour by peers. For example, a peer can modify publications, or ignore subscriptions. Again, this is a difficult problem to solve in a decentralized P2P network in which there is no centralized control of the peers.

Strictly speaking, there is no privacy provisions in this system in that the subscriptions and publications propagating through the network are visible to the peers. However, peers do enjoy some anonymity; it is impossible to know if a subscription originated at a peer or whether that peer is simply forwarding the subscription. Likewise, it is not possible to distinguish between a peer forwarding a publication and begin the publisher of that publication.

Chapter 6

Evaluation

In this chapter, we evaluate the DMM and DMM-AR algorithms described in Chapters 4 and 5. The evaluation is based on a simulation testbed that simulates a physical network, the Pastry DHT algorithm, and our publish/subscribe algorithms. We present the metrics we are interested in measuring, the workload parameters of the experiments, and a discussion of the experimental results.

6.1 Testbed

The experiments are run on SimPastry, a Pastry simulator developed at Microsoft Research, U.K. SimPastry is a discrete event simulator implemented in C# that simulates a network generated from the Georgia Tech Internetwork Topology Model (GT-ITM) generator [46]. Latencies between nodes are simulated. SimPastry includes an implementation of the Pastry protocol [6]. This simulator and Pastry implementation have been used in published research on the Pastry protocol [47, 48]. Note that only network costs are simulated in these experiments. In particular, local computation is ignored under the assumption that network latencies will dominate the computation delays.

An implementation of the DMM and DMM-AR algorithms has been implemented on top of SimPastry. The implementation includes the matching algorithms described in Chapter 4

and the multicast and fault-tolerance features outlined in Chapter 5.

6.2 Metrics

The main metrics are the load on the system, measured in terms of message and storage load on a peer, and the quality of service seen by the user, measured by the latency of delivering publications and the percentage of successfully delivered publications.

6.3 Methodology and Parameters

Topology:

The simulations where performed on a transit-stub network generated by the GT-ITM topology generator. A transit-stub network is supposed to model the Internet topology. Routers are grouped into either transit or stub domains. Transit domains correspond to the "core," backbone or interior routers in the Internet, whereas stub domains make up the edge routers. As in the Internet, there are many connections between routers within a domain, and few connections between routers in different domains.

Network characteristics:

In the simulations below, each router in a transit domain is connected to an average of 10 stub domains, with each stub domain having an average of 10 routers. Intra-domain connections have an average latency of 50ms, while inter-domain connection latencies average between 100ms and 500ms. In total there are 5050 transit and stub routers. Each peer randomly connects to one of the stub routers with a LAN connection that has a 1ms latency. Unless otherwise stated, there are 5000 peers, 100 of which are publishers, and the remaining peers are subscribers with the subscriptions evenly distributed among these subscribers. It is assumed that 0.1% of the messages transmitted in the system are randomly lost with a uniform random

distribution.

Algorithm parameters:

Unless otherwise specified, the DMM-AR algorithm with subscription covering is used in the simulations. Subscriptions are beaconed every 30s, and expire after 60s. Tree cache information is beaconed every 15s and expires after 30s.

Publish/Subscribe workload:

Each simulation run consists of a subscription phase, a stabilization phase, a publication phase, and a second stabilization phase. During the subscription phase, subscription messages spaced 10ms apart are sent for each subscription in the system. The publication phase starts 40s after the final subscription is sent. In this phase, each publisher begins to publish at some random time in the first 10s of the phase, and then continues to periodically publish every 10s over a 100s period. This results in a total of 1100 publications published in the system over the simulation run. After the last publication, the simulation continues to run for another 40s to allow the system to stabilize.

Unless otherwise stated, each publication and subscription is generated as follows. There are 10 attributes in the system with each attribute having a unique name. Each attribute is an integer in the range [1,256] having finest granularity of 1; thus, each attribute has a maximum z-code of 8 bits. A subscription consists of a random choice of 1 to 5 of the 10 attributes. For each attribute, the lower and upper values are chosen randomly within the range [1,256]. A publication consists of a random choice of 1 to 10 of the 10 attributes. For each attribute, the lower and upper values are the same, and chosen randomly within the range [1,256].

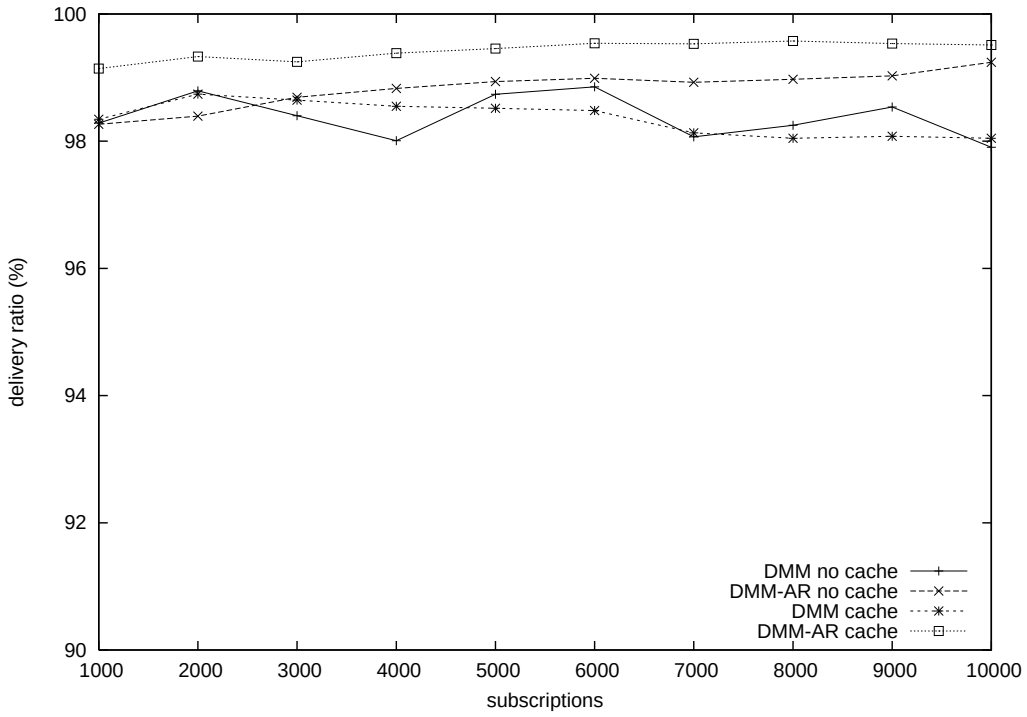

Figure 6.1: Delivery ratio (subscription scalability)

6.4 Subscription scalability

In this experiment the number of subscriptions in the system is varied, and various metrics are studied.

6.4.1 Delivery rate

The most important metric is the delivery rate, which measures the percentage of publications successfully delivered to subscribers. Figure 6.1 shows that the DMM-AR algorithm with the TreeCache information delivers virtually all the publications. Some loss is inevitable due to message losses in the system. We do not show results with acknowledgements to make the difference in delivery rates among the algorithms more evident. The use of the TreeCache optimization improves the delivery rate in the DMM-AR algorithm. This is because the optimization reduces the number of hops a publication or subscription travels and thus reduces the likelihood of a message loss. Furthermore, the DMM algorithm both with and without the

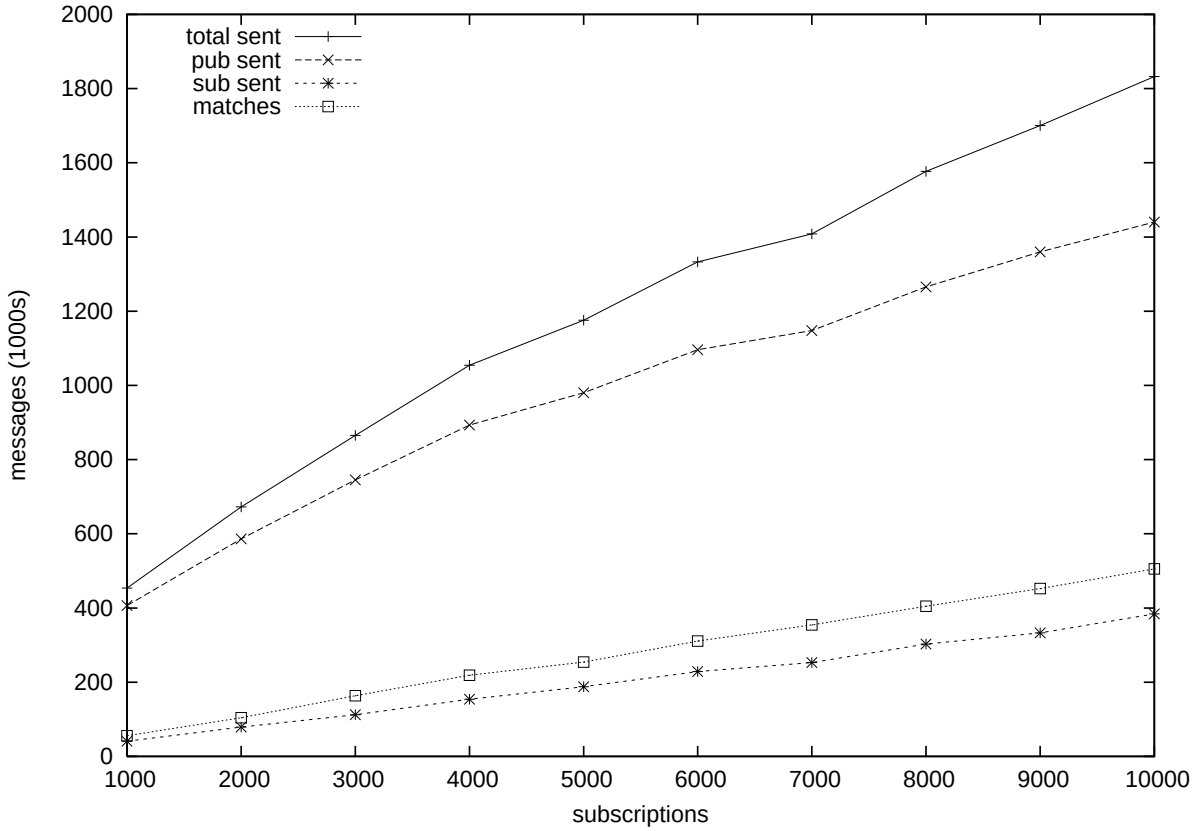

Figure 6.2: Message cost for DMM-AR with TreeCache (subscription scalability)

TreeCache optimization has a worse delivery rate than DMM-AR. This is because publications travel longer paths in DMM compared to DMM-AR (see Section 6.4.3) and hence are more likely to experience a message loss.

6.4.2 Message cost

Figure 6.2 shows the number of messages in the system for the DMM-AR algorithm with the TreeCache optimization. First note that the total number of messages grows sub-linearly with the number of subscriptions in the system—a ten fold increase in subscriptions from 1000 to 10000 results in a less than five fold increase in messages—so the algorithm is scalable with respect to message cost. The figure also plots the total number of publications and subscriptions. We see that these two types of messages make up the bulk of the messages in the system.

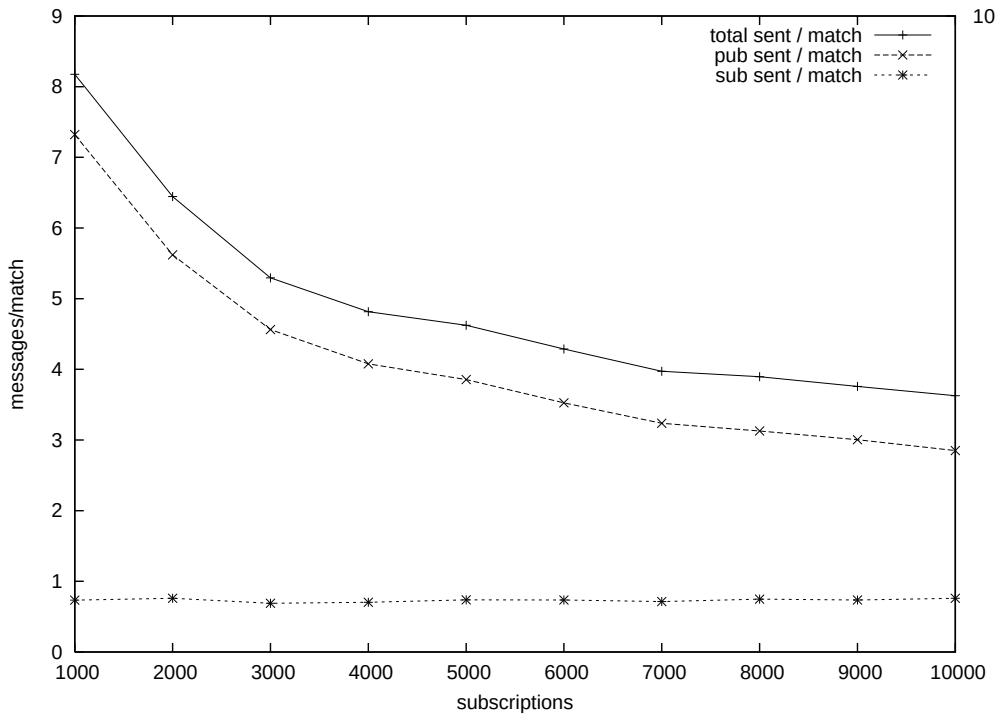

Figure 6.3: Normalized message cost for DMM-AR with TreeCache (subscription scalability)

In particular, the relative number of TreeCache messages is almost non-existent. We also see that publications dominate the messages in the system. Notably, the periodic beaconing of subscriptions does not cause an excessive message load compared to the publications.

To emphasize the sub-linear increase in message cost, Figure 6.3 plots the message cost normalized by the number of expected matches. We see that there is a incremental cost of delivering a publication to an additional subscription diminishes with the number of subscriptions (more accurately, the number of matching subscriptions) in the system. This confirms that the DMM-AR algorithm still provides the advantages of multicast publication delivery.

Figure 6.4 shows the total message counts for the DMM and DMM-AR algorithms with and without the TreeCache optimization. The DMM-AR algorithm has a much lower message cost than DMM. We see that TreeCache helps to reduce the message cost in the DMM-AR algorithm. Also, the DMM algorithm suffers from a larger message cost than the DMM-AR algorithm. This again is due to the longer paths that publications take in the DMM algorithm.

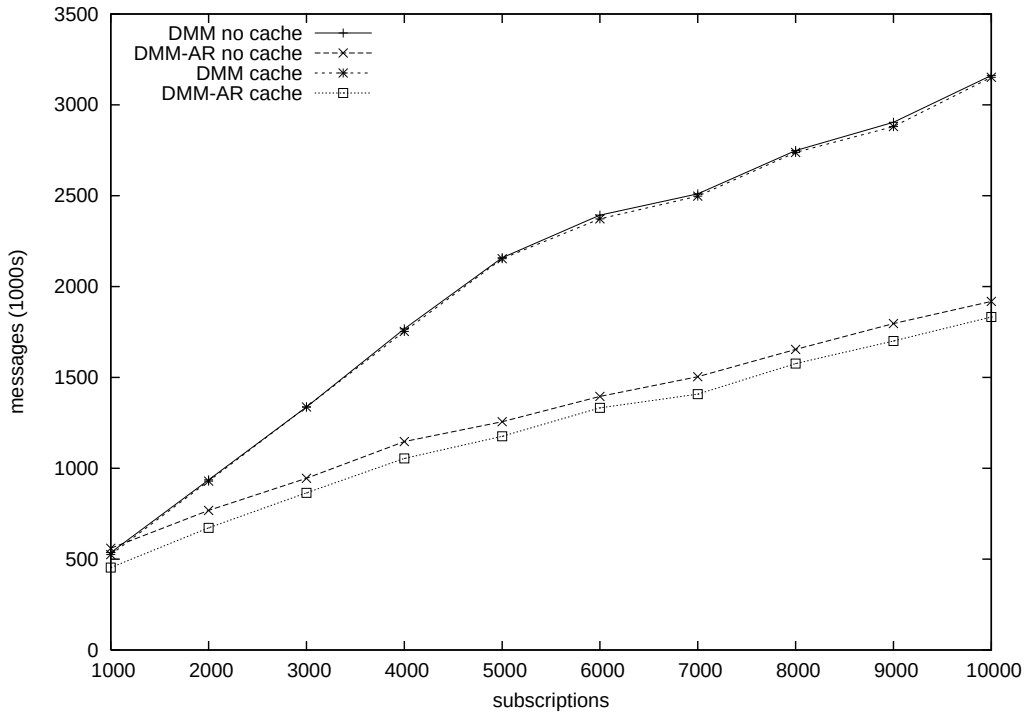

Figure 6.4: Message cost (subscription scalability)

The TreeCache optimization does not improve this cost because, as explained in explained in Section 6.4.3, the effectiveness of the cache is diminished when information about the relatively larger DMM tree needs to be cached.

It is important to note that the message costs above count the physical hops traversed, not the overlay hops, and hence improvements in the DHT will improve this metric. Also, our experiments show that these message costs are largely determined by the publication and subscription workloads, i.e., the size of the DMM tree and the cost of traversing the tree to find matches. In particular, the costs do not increase with the size of the network. In this way, the algorithm is quite scalable with respect to message cost.

6.4.3 Publication delivery latency

First we measure the latency of publication delivery. As a given publication is likely to be delivered to multiple subscribers, the minimum, average, and maximum delivery latencies are

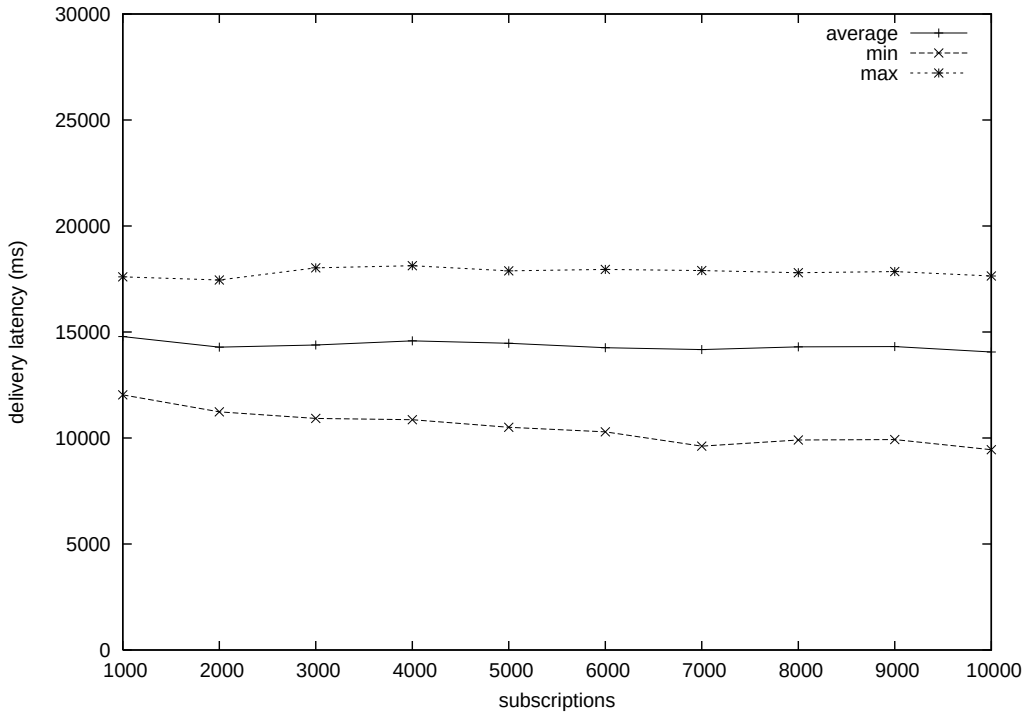

Figure 6.5: Delay for DMM-AR without TreeCache (subscription scalability)

plotted.

Figure 6.5 shows the latency with the DMM-AR algorithm. We see that the average publication takes about 15s to be delivered. The large delay is primarily due to the need for publications to traverse multiple hops from the bottom of the tree towards the root. This hypothesis is verified by Figure 6.6 which shows the TreeCache optimization. Here, the average publication latency has been drastically reduced to about 6s.

Figures 6.7 and 6.8 investigate the effect of the TreeCache optimization further. The figures plot the delivery latency of each individual publication. We see that after about 20s of the simulation, the tree information is sufficiently cached to quickly and drastically reduce the delivery latency from about 15s to about 3s.

It is also instructive to see how DMM performs relative to DMM-AR. Recall that the DMM algorithm constructs one large global space (and thus requires a global schema) and does not suffer from the additional step of attribute roots. Figure 6.9 shows the latency without the

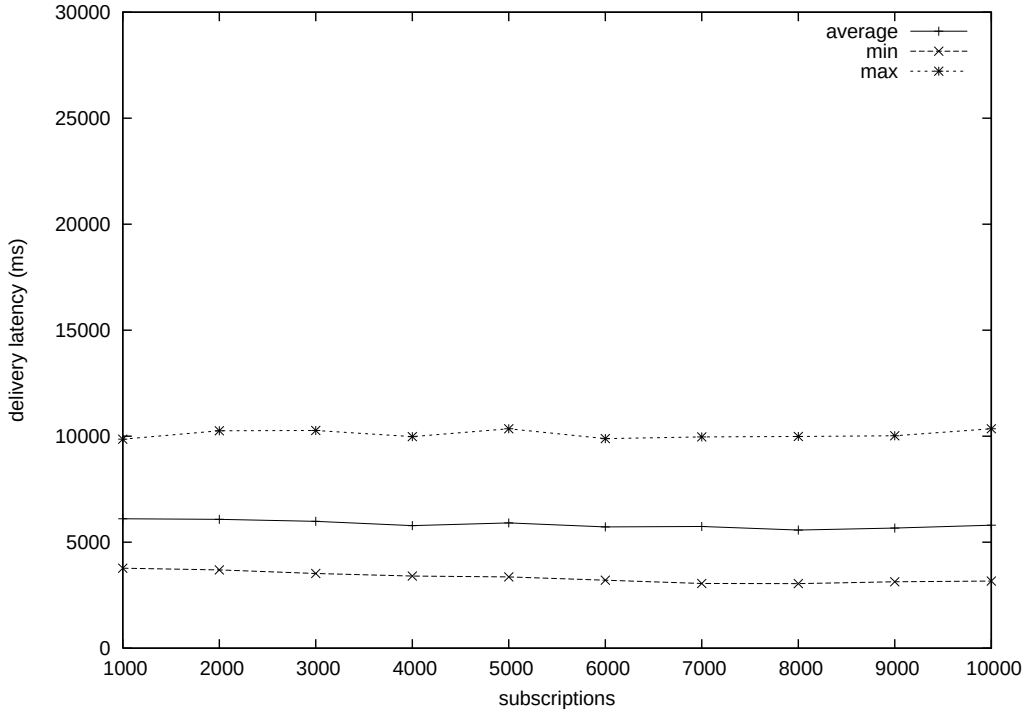

Figure 6.6: Delay for DMM-AR with TreeCache (subscription scalability)

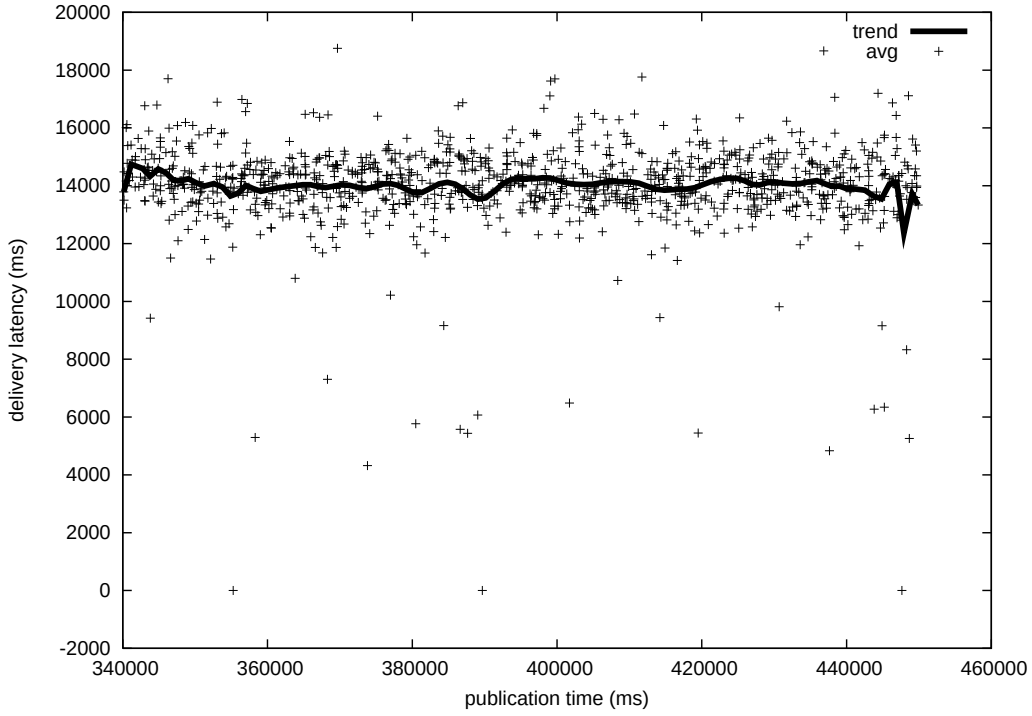

Figure 6.7: Delay over time for DMM-AR without TreeCache (10000 subscriptions)

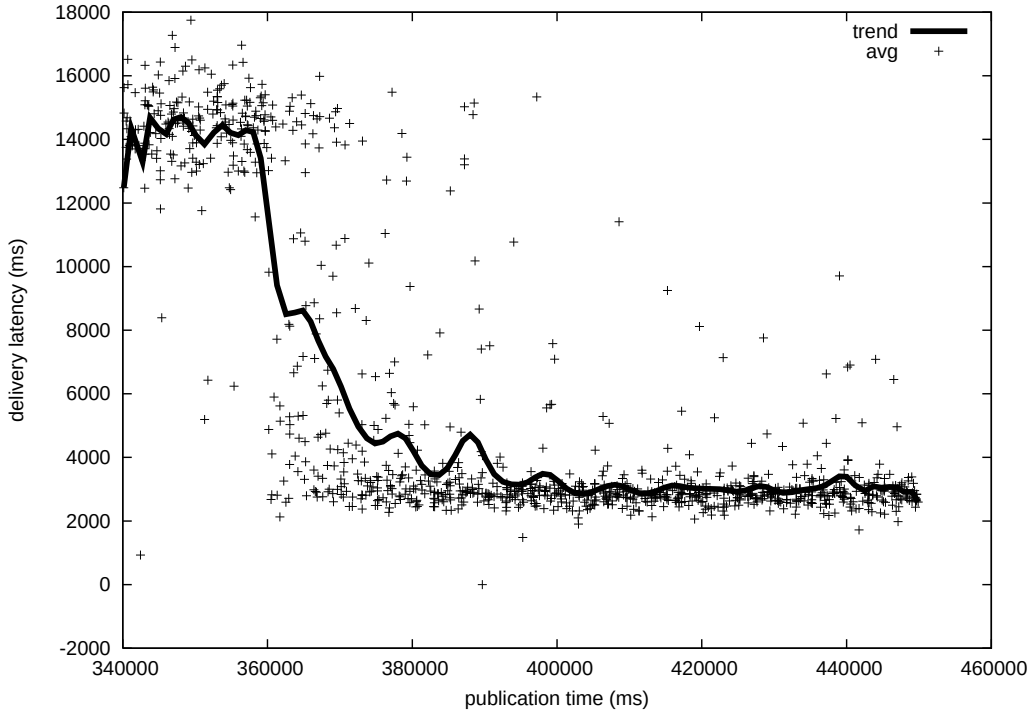

Figure 6.8: Delay over time for DMM-AR with TreeCache (10000 subscriptions)

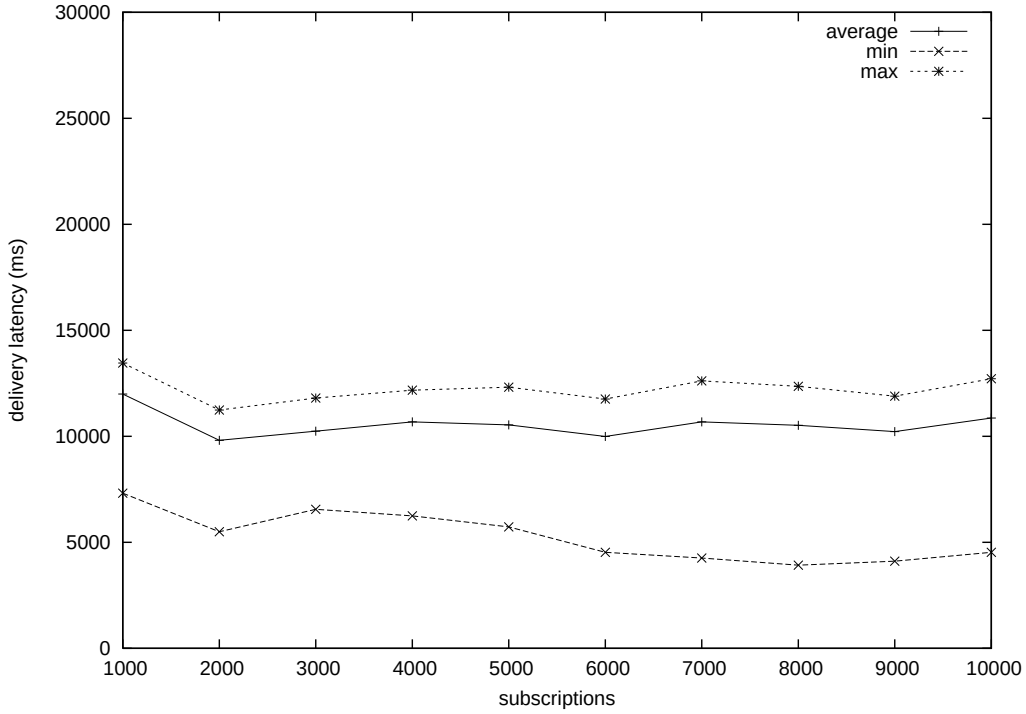

Figure 6.9: Delay for DMM without TreeCache (subscription scalability)

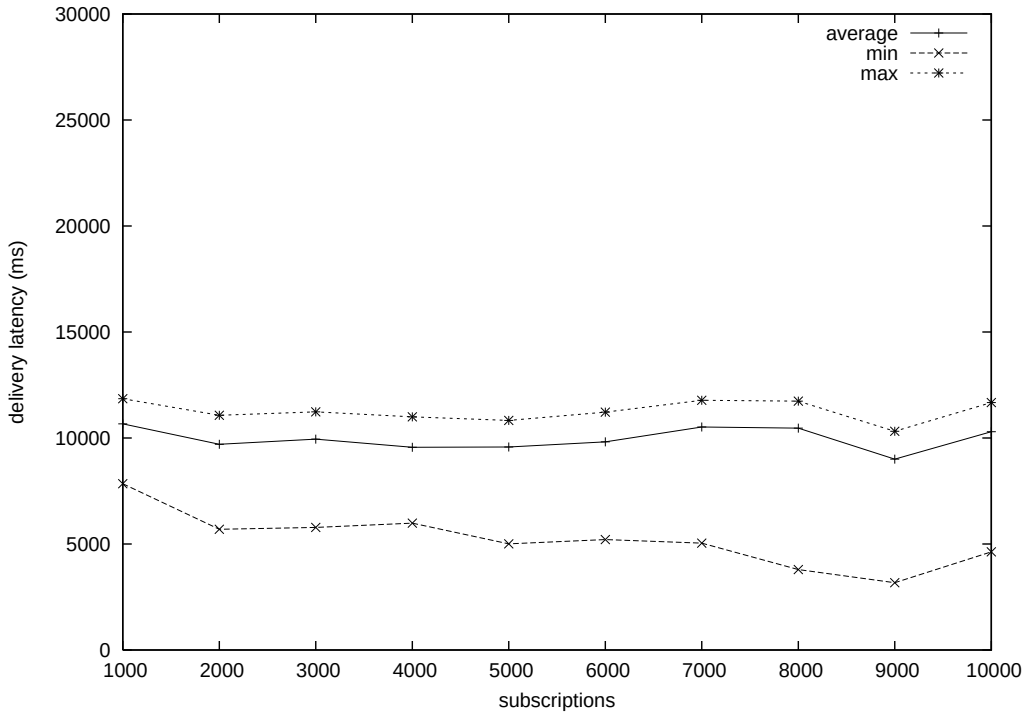

Figure 6.10: Delay for DMM with TreeCache (subscription scalability)

TreeCache optimization. Notice that the average latency with the DMM algorithm is about 10s which is lower than that in DMM-AR; this is due to the absence of attribute roots, which reduces the number of hops a publication travels.

Interestingly, the use of the TreeCache optimization in the DMM algorithm has little effect; in Figure 6.10 the average publication delivery latency remains around 10s. This is because the DMM algorithm sends publications towards a tree that represents a high-dimensional space. With the workload used here (10 attributes with an 8-bit z-code per dimension, leading to 80 bits in the z-code of a publication), the full tree contains 2^{80} leaf nodes. A given publication is sent towards one of these leaf nodes. Since the tree information is only cached at the hop preceding the intended destination leaf, the large number of leaves means that the tree cache is dispersed over many nodes and is not used frequently—that is, there are few cache hits

Compare this to the DMM-AR case where publications are sent to an attribute root whose full tree only has 2^8 leaves. The DMM-AR algorithm interestingly also benefits from the fan-

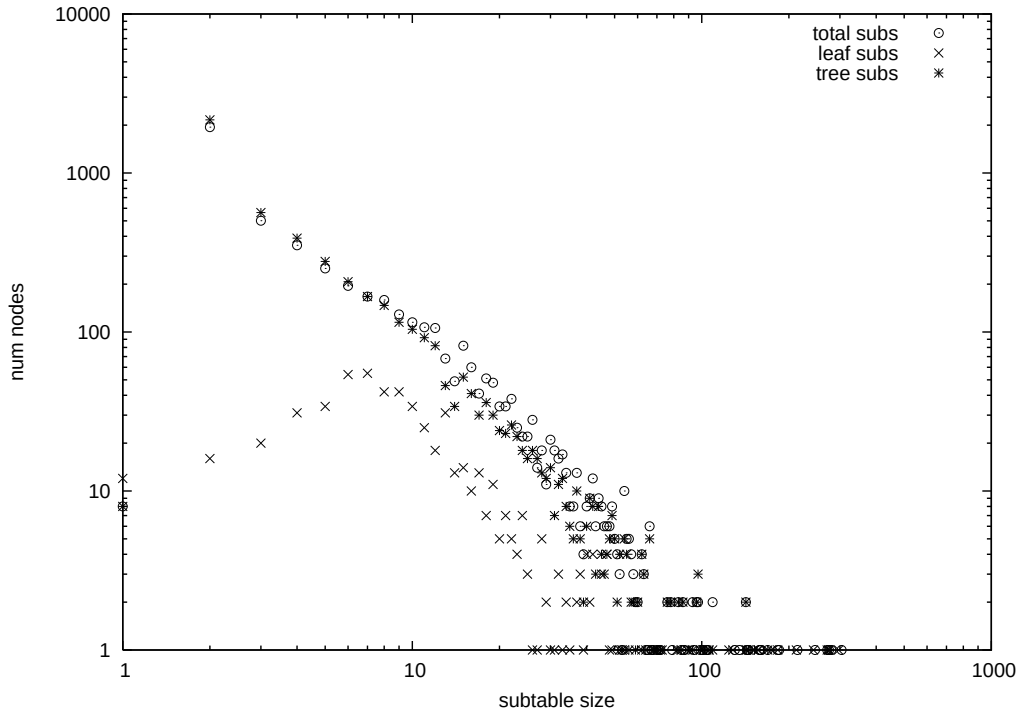

Figure 6.11: Subscription state for DMM-AR with TreeCache (10000 subscriptions)

out of a publication to multiple attribute roots. This has the effect of increasing the publications in the system and thus more opportunities for the tree information cache to be updated.

6.4.4 Subscription state

Figure 6.11 illustrates the distribution of the subscription state among the peers in the network. A peer's state is taken to be the number of subscriptions stored at that peer. The figure shows the number of peers that have a particular subscription table size. A point on the graph with an x-value of 10 and y-value of 100 means that there are 100 peers that store 10 subscriptions. Three sets of plots are shown: the total number of subscriptions, the number of subscriptions in the leaf state and the number of subscription in the tree state. Recall that subscriptions in the leaf state are stored in the DMM (or DMM-AR) tree for matching publications with subscriptions, while those in the tree state are stored to maintain multicast paths and are used to multicast publications. The figure shows that most peers have very little total state, while

only a few peers have a large state. This effect is more pronounced for the tree state subscription compared to the leaf state subscriptions.

6.5 Subscription dimensionality

In this experiment the number of dimensions in a subscription is varied, to measure the effects of more "complex" subscriptions. Each subscription has a fixed number of attributes from 1 to 10. A subscription with more attributes is more selective, and hence fewer publications will match it. This will, among other things, decrease the message load on the system. These secondary effects are removed by constructing a workload that has that the same number of matches regardless of the number of attributes in the subscription. This is done by randomly choosing the lower and upper bounds of the first attribute in each subscription as usual. The additional attributes have lower and upper bounds that cover the entire range. Hence, only the first attribute acts to discriminate among publications; the other attributes will always match. In addition, publications always have 10 attributes all whose values are chosen randomly. This workload results in an equal number of matches per subscription regardless of the the number of attributes per subscription.

Figure 6.12 shows that the TreeCache optimization increases the delivery rate, and Figure 6.13 shows that the message cost is reduced by this optimization. We also see that increasing the number of dimensions of the subscriptions has negligible impact on the message cost. This means that the matching algorithm is not affected by the complexity of the subscriptions, measured here by the number of attributes per subscription.

Figure 6.14 shows as expected that the publication latency does not vary greatly with increasing subscription dimension. So, the complexity of subscriptions has minimal effect on the quality of service (as measured by delivery latency) experienced by a subscriber. Also, as before, the TreeCache optimization helps to greatly reduce the delivery latency of publications.

The speed with which the TreeCache optimization reduces publication delivery latency, and

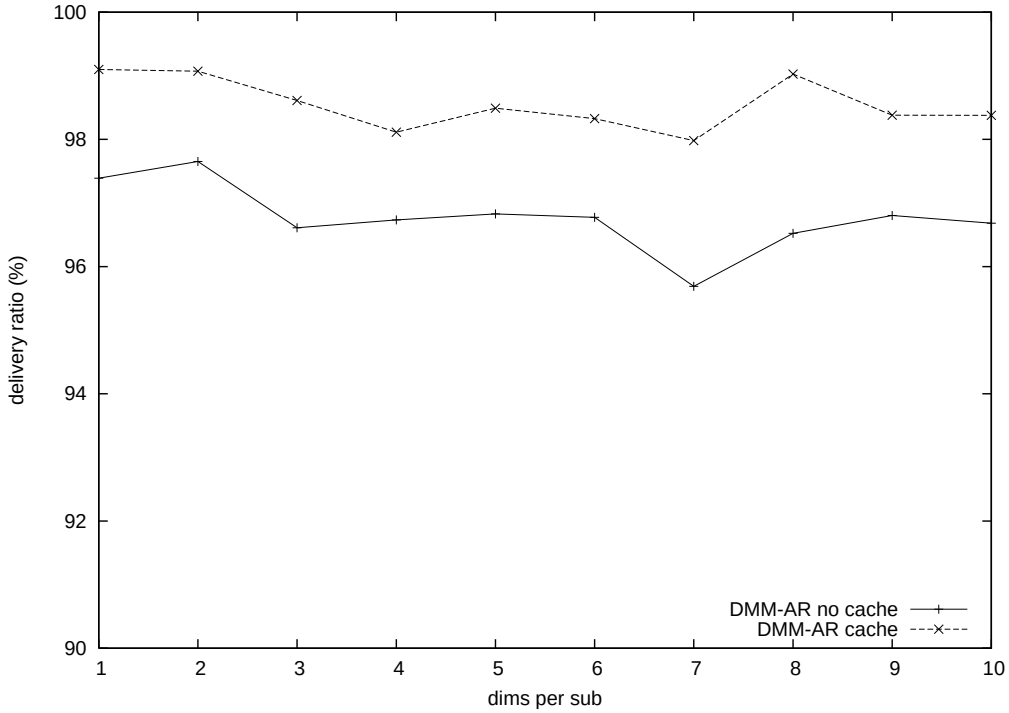

Figure 6.12: Delivery ratio (subscription dimensionality)

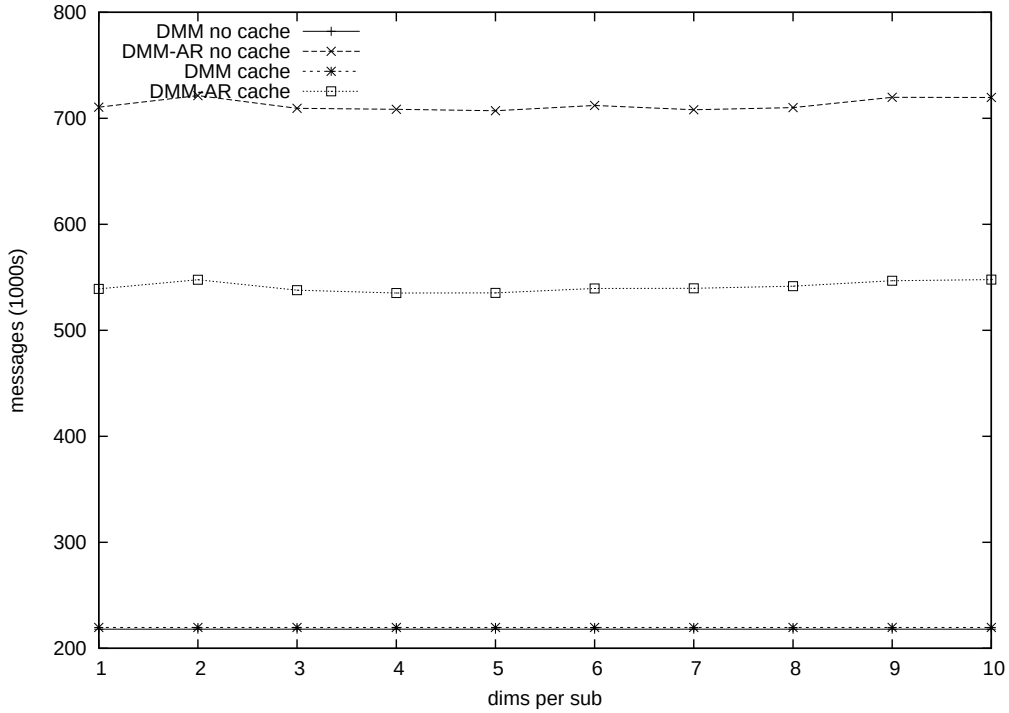

Figure 6.13: Message cost (subscription dimensionality)

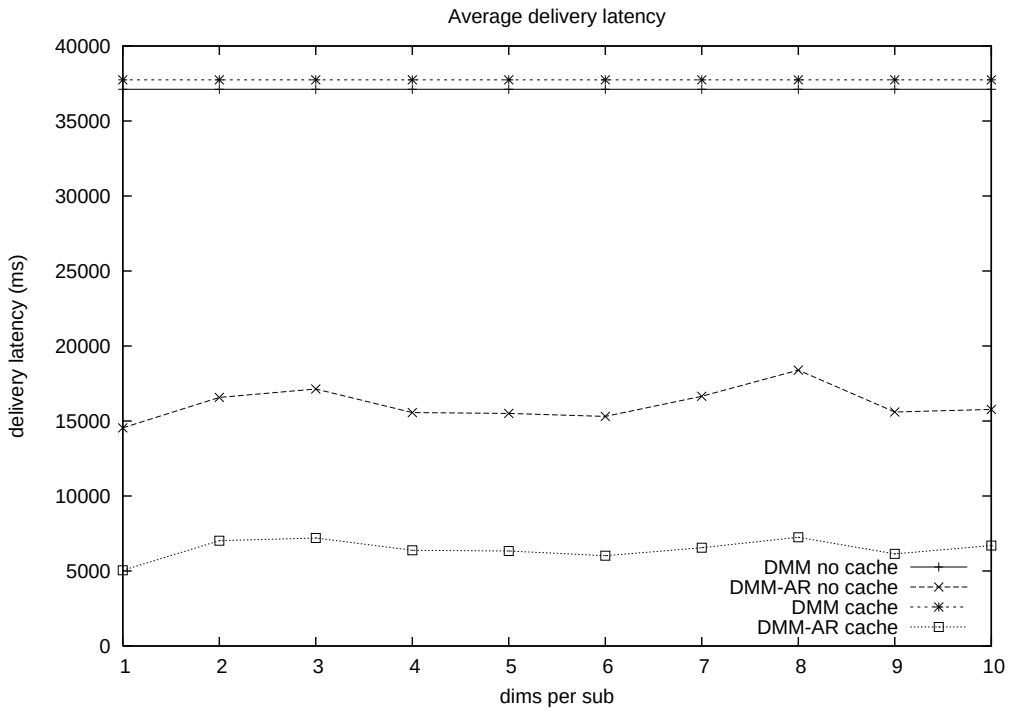

Figure 6.14: Delay (subscription dimensionality)

the distribution of the subscription state is similar to the results in the previous section.

6.6 Fault-tolerance

In this experiment, we study the resilience of the algorithm to faults in the network. The workload consists of 1000 subscriptions, and an aggregate publication rate of 100 per minute. At the 400s mark, 1000 of the 5000 nodes in the network simultaneously crash, that it they no longer send or receive any messages.

Figure 6.15 plots the delivery ratio of the publications in the system over time. A Bezier curve is used to smooth the curve and show the trend. The delivery ratio is a ratio of the number of actual deliveries of a publication to the number of live interested subscribers during the time of publication. Therefore, failure to deliver to an interested subscriber that crashes immediately after publication reduces the delivery ratio, while an interested subscriber that crashes before publication does not affect the delivery ratio.

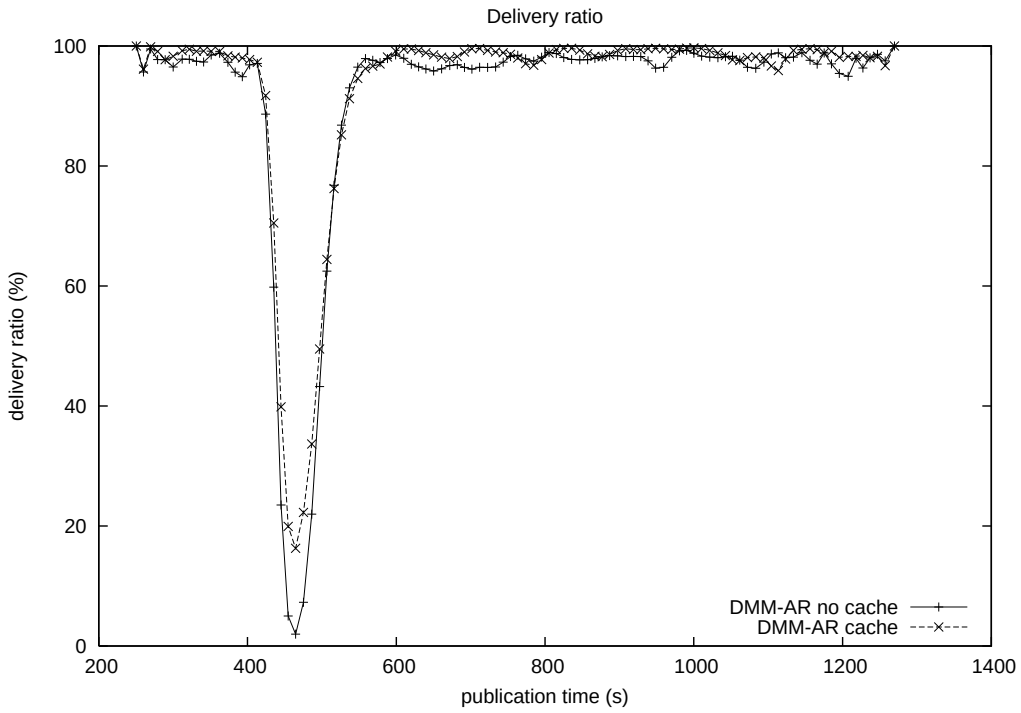

Figure 6.15: Delivery ratio (fault-tolerance)

The results show that the correct delivery of publications resumes within about 100s of the nodes crashing. (While in the figure it seems as though this recovery is closer to 200s, this is an artifact of the Bezier curve approximation lagging the actual data.) The recovery time is largely accounted for by the delay in the underlying DHT to reorganize around the faults. Note that despite the unreasonably large number of network failures—one fifth of the peers simultaneously crash—the network is still able to recover relatively quickly.

Chapter 7

Conclusions

This chapter summarizes the work presented in this thesis and proposes directions for future research.

7.1 Summary

Early publish/subscribe research focused on developing more efficient matching algorithms. These were followed by research into distributed matching algorithms. However, these distributed algorithms, require the provisioning of the distributed broker infrastructure. In this work, we address this problem by using a P2P network that does not require any dedicated infrastructure. Just as the publish/subscribe paradigm attempts to reduce the effort and expertise required to develop distributed applications, a publish/subscribe middleware based on a P2P substrate lowers the cost of deploying distributed applications.

We use a P2P substrate based on a distributed hash table. While we use Pastry as the DHT substrate, our matching algorithm can work with any overlay that supports a DHT interface. A DHT offers scalable probabilistic guarantees on routing and storage performance as well as self organization and a simple hash table interface. While DHTs can offer infrastructureless scalability, it is not trivial to implement a distributed content-based publish/subscribe system on a DHT interface. This thesis developed such an algorithm. The algorithm addresses limitations

with other algorithms such as requiring a globally known static set of attributes. The algorithm is based on mapping the content-based publish/subscribe matching problem into a multidimensional indexing problem. We build a multidimensional tree index over a DHT interface, and use this to achieve publish/subscribe matching.

The algorithm balances load by delegating subscriptions from overloaded peers and by employing a bottom up tree search technique in order to avoid the root overload problem present in most multidimensional tree indexing algorithms. While the bottom up tree search relieves load at the root node, it can cause long search paths. We address this by caching some data on the nature of the multidimensional index. Our results show that this technique substantially reduces the search path. Experiments also demonstrate that the performance of the publish/subscribe multidimensional indexing technique is not affected by the "complexity" of the subscriptions. Namely, the number of attributes in a subscription does not affect the delivery ratio, message cost or delivery latency. The algorithm also scales well in terms of message cost as the number of subscriptions in the system increases. In fact, due to the multicast message dissemination, the cost of delivering a publication to an interested subscriber decreases as the subscriber population grows. Two fault-tolerance mechanisms—an active failure detector, and periodic state refreshing—allow the algorithm to quickly recover from even a serious crash of the network.

7.2 Future work

There are many directions in which this research can be continued. We would like to support more complex data types in the publication and subscription language such as XML and XPath. The evaluation would also benefit from a more realistic workload. There are no standard test benches or workloads in the publish/subscribe community, nor do we have access to real-world publish/subscribe traces. Another class of problems is real-time issues. Some applications require timely delivery of data.

Another rich area for future work is addressing security issues, including authentication and confidentiality. Authentication refers to the ability for a publisher to control the set of subscribers to which a publication can be delivered. However, solving this problem may require knowing the addresses of publishers and subscribers, which violates the publish/subscribe model. With confidentiality, we would like to keep the publications and subscriptions private. This is difficult because publish/subscribe content-based routing requires access to the content of the messages. These issues have not been adequately addressed in the literature. We note that security concerns are both more pressing and more difficult in a decentralized P2P architecture where even the brokers may be malicious.

Bibliography

[1] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *SIGMOD Conference*, 2001.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, pp. 332–383, Aug. 2001.

[3] IBM, "IBM serves on demand solutions at the Australian Open." http://www-8.ibm.com/e-business/au/australianopen/pdf/ australian_open_case_study.pdf.

[4] Akamai. http://www.akamai.com/en/html/about/company_information.html.

[5] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

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

[7] 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 *19th IEEE International Conference on Distributed Computing Systems (ICDCS)'99*, 1999.

[8] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, vol. 27, pp. 827–850, sep 2001.

[9] L. Fiege, F. C. Grtner, S. B. Handurukande, , and A. Zeidler, "Dealing with uncertainty in mobile publish/subscribe middleware," in *Proceedings of Middleware 03 - Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC)*, 2003.

[10] P. Sutton, R. Arkins, and B. Segall, "Supporting disconnectedness - transparent information delivery for mobile and invisible computing," in *CCGrid 2001 IEEE International Symposium on Cluster Computing and the Grid*, 2001.

[11] D. McCarthy and U. Dayal, "The architecture of an active database management system," in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pp. 215–224, ACM Press, 1989.

[12] H. Liu and H.-A. Jacobsen, "A-ToPSS - a publish/subscribe system supporting approximate matching," in *Very Large Databases (VLDB'02)*, (University of Toronto), August 2002.

[13] M. Petrovic, I. Burcea, and H.-A. Jacobsen, "S-ToPSS - a semantic publish/subscribe system," in *Very Large Databases (VLDB'03)*, (Berlin, Germany), September 2003.

[14] H. K. Y. Leung, "Subject space: A state-persistent model for publish/subscribe systems," in *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, p. 7, IBM Press, 2002.

[15] F. Fabret, H.-A. Jacobesen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2001.

[16] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system," in *Symposium on Principles of Distributed Computing*, pp. 53–61, 1999.

[17] G. Ashayer, H. Leung, and H.-A. Jacobsen, "Predicate matching and subscription matching in publish/subscribe systems," in *DEBS'02 Workshop at ICDCS'02 (DEBS'02)*, (Vienna, Austria), 2002.

[18] I. Burcea, H.-A. Jacobsen, E. de Lara, V. Muthusamy, and M. Petrovic, "Disconnected operation in publish/subscribe middleware," in *Proceedings of the 5^{th} International Conference on Mobile Data Management (MDM'2004)*, (Berkeley, CA, USA), pp. 39–50, IEEE Computer Society, 2004.

[19] S. Bhola, Y. Zhao, and J. Auerbach, "Scalably supporting durable subscriptions in a publish/subscribe system," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003)*, pp. 57–66, June 2003.

[20] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica, "Querying the internet with PIER," in *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sept. 2003.

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

[22] "The Gnutella protocol specification v0.4." http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[23] Kazaa. http://www.kazaa.com.

[24] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. I. T. Rowstron, and A. Singh, "Splitstream: High-bandwidth content distribution in cooperative environments," in *IPTPS*, pp. 292–303, 2003.

[25] A. Mislove, C. Reis, A. Post, P. Willmann, P. D. schel, D. Wallach, X. Bonnaire, P. Sens, J.-M. Busca, and L. Arantes, "Post: A secure, resilient, cooperative messaging system," in *Ninth IEEE Workshop on Hot Topics in Operating Systems (HotOS-IX)* (IEEE, ed.), (Kaui (USA)), IEEE Society Press, June 2003.

[26] A. I. T. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *SOSP*, pp. 188–201, 2001.

[27] F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP*, pp. 202–215, 2001.

[28] V. Ramasubramanian and E. G. Sirer, "The design and implementation of a next generation name service for the Internet," in *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 331–342, ACM Press, 2004.

[29] T.-W. Ngan, D. S. Wallach, and P. Druschel, "Enforcing fair sharing of peer-to-peer resources," in *IPTPS*, pp. 149–159, 2003.

[30] K. Singh and H. Schulzrinne, "Peer-to-peer Internet telephony using SIP," in *New York Metro Area Networking Workshop (NYMAN)*, (New York, NY), September 2004.

[31] National Institute for Standards and Technology, "Secure Hash Standard (SHS)," Technical Report FIPS 180-1, U.S. Department of Commerce, 1995.

[32] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 161–172, ACM Press, 2001.

[33] 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*, vol. 22, pp. 41–53, Jan. 2004.

[34] K. Aberer, "P-Grid: A self-organizing access structure for P2P information systems," in *Proceedings of the 9th International Conference on Cooperative Information Systems*, pp. 179–194, Springer-Verlag, 2001.

[35] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "Skipnet: A scalable overlay network with practical locality properties," in *Proc. USITS Conf.*, (Seattle, WA), March 2003.

[36] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, 1998.

[37] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, oct 2002.

[38] P. R. Pietzuch and J. Bacon, "Peer-to-peer overlay broker networks in an event-based middleware," in *Proceedings of the 2nd international workshop on Distributed event-based systems*, pp. 1–8, ACM Press, 2003.

[39] W. 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 the 2nd international workshop on Distributed event-based systems*, pp. 1–8, ACM Press, 2003.

[40] D. Tam, R. Azimi, and H.-A. Jacobsen, "Building content-based publish/subscribe systems with distributed hash tables," in *International Workshop on Databases, Information Systems, and P2P Computing (DBISP2P) (co-located with VLDB 2003)*, vol. 2788 of *Lecture Notes in Computer Science*, (Berlin, Germany), pp. 138–152, September 2003.

[41] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: Content-based publish/subscribe over P2P networks," in *Middleware*, pp. 254–273, 2004.

[42] A. Andrzejak and Z. Xu, "Scalable, efficient range queries for grid information services," in *Proceedings of the Second International Conference on Peer-to-Peer Computing*, p. 33, IEEE Computer Society, 2002.

[43] A. Daskos, S. Ghandeharizadeh, and X. An, "PePeR: A distributed range addressing space for peer-to-peer systems," in *International Workshop on Databases, Information Systems, and P2P Computing (DBISP2P) (co-located with VLDB 2003)*, vol. 2788 of *Lecture Notes in Computer Science*, (Berlin, Germany), pp. 200–218, September 2003.

[44] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pp. 14–29, Springer-Verlag, 2001.

[45] 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 the 11th international workshop on Network and operating systems support for digital audio and video*, pp. 11–20, ACM Press, 2001.

[46] K. L. Calvert, M. B. Doar, and E. W. Zegura, "Modeling Internet topology," *IEEE Communications Magazine*, vol. 35, pp. 160–163, June 1997.

[47] A.-M. K. M. Castro, P. Druschel and A. Rowstron, "Scribe: A large-scale and decentralised application-level multicast infrastructure," *IEEE Journal in Selected Areas in Communication (JSAC)*, vol. 20, October 2002.

[48] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Proximity neighbor selection in tree-based structured peer-to-peer overlays," Technical Report MSR-TR-2003-52, Microsoft Research, 2003.