A Content-based Publish/Subscribe Framework over Structured Peer-to-Peer Networks

by

Wei Li

B.Sc., Beijing Normal University, 2001
M.Eng., Beijing Normal University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

(Vancouver)

June, 2008

Abstract

The Publish/Subscribe model has become a prevalent paradigm for building distributed notification services by decoupling the publishers and the subscribers from each other. Content-based publish/subscribe allows for highly expressive descriptions of subscriptions and thus is more appropriate for content dissemination when a finer level of granularity is necessary. However, scalability has become an issue due to the expensive matching and delivering inherent in content-based events. In this thesis we propose a novel content-based publish/subscribe framework built over a DHT-based P2P network in order to provide scalable content delivery mechanisms. Based on efficient subscription installation, event publishing and event delivery techniques, our system can provide a scalable platform to support multiple different pub/-sub schemas. There are three key features in our design: (1) A logic space mapping and a distributed 2d-tree that maintains this space over DHT; (2) Novel random probing searching schemes allowing for subscription installation and event publication; (3) An efficient application layer multicast algorithm for message delivery with low bandwidth consumption.

Table of Contents

List of Tables

List of Figures

List of Programs

Acknowledgements

I would like to gratefully and sincerely thank my supervisor, Dr. Son Vuong, for his inspiration, guidance and encouragement. Without his support, my current achievement would have been impossible.

To my second reader, Dr. George Tsiknis, I am grateful to him for his invaluable and insightful comments that have helped me improve this thesis.

I also would like to thank all my colleagues in NIC lab, especially Juan Li, Mohammed Alam, Billy Cheung, Stanley Chiu, Ricky Cheng and Minghao Lu for their constructive discussions and comments, to all those who have made NIC lab a wonderful workplace. In particular I am indebted to Billy for proofreading and polishing my thesis.

Last, but not least, I thank my family for their consistent support, love and patience.

Wei Li

The University of British Columbia

June 2008

Dedication

To my father, who loved and supported me throughout his whole life.

Chapter 1

Introduction

Publish/Subscribe systems provide users with the ability to express their interest in a subscription and subsequently receive notifications for of any event, generated by a publisher, that matches their interest [11]. A subscription can either be described by a topic name (topic-based model), or by a set of attributes and values (content-based model). A Publish/Subscribe service usually evinces two basic properties: (1) Anonymity: Publishers and subscribers are independent of each other. A publisher does not need to be aware of any subscribers to publish an event, and vice versa. (2) Asynchronization. The publication and notification of events do not conform to the main flow of control of the publishers and subscribers. Thus, the communication between publishers and subscribers are asynchronous.

1.1 Motivation

Publish/Subscribe systems arise in many domains including personal (e.g., news alert, online deal-hunting and online bidding), financial (e.g. trading stock and commodity in real time), etc. For example, a bargain-hunter may want to buy a new laptop online, but find it too expensive. He decides to wait until the price drops to a price that he can afford. With a publish/subscribe

service, by simply submitting a subscription with his requirements, he is able to get a notification whenever there is a deal meets his requirements. This frees him from having to keep refreshing the websites. A stock trader could also benefit from a publish/subscribe system by watching stock price through notifications for selling or buying.

Current Publish/Subscribe designs are either centralized or distributed. Centralized solutions store all the subscriptions on a central server. When an event is published, a DBMS is used to match the subscriptions with the event [14] [8]. However, these systems lack scalability as the number of events and subscriptions increases. Although there are some special data structures[4] proposed to solve this problem, they try to improve scalability by limiting the expressiveness of subscriptions. In addition, a single point of failure can devastate the entire service.

To overcome the scalability problem of centralized model, the academic community has paid great attention to solutions based on distributed model such as SIENA [6], Gryphon[3] and MEDYM[5]. All these designs rely on pre-deployed overlay networks formed by a set of independent, communicating servers. These servers act as brokers for subscriptions and events. The principle behind these systems involve allowing only a subset of the nodes in the system to store each subscription and a subset of nodes are visited by each event so that these events are forwarded only to nodes that lie on an overlay path leading to interested subscribers. The natural architecture of the brokers' overlay for this kind of solution is usually an acyclic tree or graph. Events are delivered through a multicast tree formed by these brokers. Obviously the performance of these solutions is significantly influenced

by the topology of the overlay network. In addition, these brokers' overlay networks are inherently static, managed by administrators and expensive to deploy.

Peer-to-Peer overlay networks have emerged as a promising alternative solution for internet-scale distributed applications due to their flexibility and scalability. As completely decentralized peer-to-peer networks composed of a set of nodes forming a structured graph, Distributed Hash Tables (DHTs)[22], [18], [1], [19], [29] have additional advantages including high searching efficiency, low overhead and fault-tolerance. Therefore, we choose DHT to be the P2P substrate of our content-based publish/subscribe system. But DHTs are designed only for exact key matching. By mapping an object to a hash key in the virtual key space, DHT hides all the semantics of the object, which makes it a significant hurdle to implement a content-based publish/subscribe system on top of DHT because users' interests can contain various semantic information. Therefore, a content-based publish/-subscribe system over DHT needs to address four core issues: (1) How to reconstruct the semantics of users' interests over DHT. (2) How to install subscriptions to the network and manage them. (3) How to efficiently find all the subscriptions that match an incoming event and deliver the event to the corresponding subscribers. and (4) How to take into consideration the fact that a practical peer-to-peer network is formed by heterogeneous peers.

1.2 Thesis Contributions

In this thesis, we propose a novel content-based publish/subscribe system built on top of DHT. The main contributions of this thesis are:

- We propose a distributed 2d-tree structure to reconstruct the semantics behind subscriptions and events, inspired by the kd-tree data structure[4].
- We design a novel subscription installation algorithm to install all the subscriptions in the overlay.
- We develop an event publishing and delivery algorithm to publish events and search peers for event matching. To reduce the bandwidth consumption, we also propose an application layer multicast algorithm.
- We propose a load balancing scheme to improve our system's performance.
- We evaluate our proposed system with FreePastry[26] through an extensive simulation on a large Internet-like network model.

1.3 Thesis Organization

This thesis consists of five chapters. Chapter 2 introduces the background of P2P and DHT networks, Publish/Subscribe basis and a review of related work. In Chapter 3, we present our system design in detail, focusing on our space mapping technique, then our subscription installation, event publishing and delivering algorithms, as well as a load balancing scheme to ensure

system scalability. We evaluate our design in Chapter 4 and in Chapter 5 examine our results and what conclusions can be drawn from them as well as look into future areas where we can explore.

Chapter 2

Chapter Two: Background and Related Work

This chapter first provides background information about Peer-to-Peer (P2P) technology with an emphasis on Pastry [20] network. Then we introduce two fundamental concepts in a content-based Publish/Subscribe diagram. The last section is a review of related work.

2.1 P2P Computing

With the prevalence of the Internet, computing and communication environments have become significantly more complex and chaotic, evolving into something beyond what a classical distributed systems were ever intended to support. Since the debut of the first P2P product Napster [16] in late nineties, Peer-to-Peer overlays have attracted plenty of interest from both academia and industry because they provide a good substrate for creating Internet-scale data sharing and content distribution applications.

What is peer-to-peer? A quick look into many literature available reveals that various different definitions are being used. For example, a widely accepted definition by Shirky[21] says "peer-to-peer is a class of applications

that take advantage of resources-storage, cycles, content, human presence—available at the edges of the internet". But this definition excludes applications which rely upon some centralized servers for their operation, such as the aforementioned famous p2p application: Napster. Though it's not the intention of this thesis to provide a universally accepted definition of P2P, from our perspective there are three common characteristics present in any P2P system:

- Nodes are able to be interconnected to share resources such as content, storage, computing capacities, etc.
- Unlike traditional Client/Server model, nodes in a p2p network are able to function as both clients and servers.
- A p2p network is self-organizing and resilient to environment dynamics. This means that the system is able to adapt itself to topology changes as nodes enter or leave the network.

In terms of overlay structure, p2p networks can be divided into two different categories, unstructured and structured P2P.

2.1.1 Unstructured P2P

In an unstructured p2p network, there is usually no specific node placement structure on the overlay topology. Nodes are randomly placed in either a flat or hierarchical graph. As a result, unstructured p2p systems traditionally suffer from the problem of objects in the network being difficult to locate. Searching mechanisms, ranging from early proposed flooding model of the

network with propagating queries to more recent and sophisticated random walks and routing indices, have not completely overcome this problem. A main drawback of unstructured p2p networks therefore is that they lack scalability.

2.1.2 Structured P2P and Distributed Hash Table

Structured P2P networks have emerged mainly in an attempt to address the scalability issues that unstructured systems face with. In contrast to unstructured P2P overlays, in structured P2P systems, through a tightly controlled network topology, peers are logically placed at deterministic locations to form and maintain a special structure. The most prominent class of approaches to such structured P2P systems are Distributed Hash Tables (DHT), in which each node holds a part of the hash table and an object is stored on this node if this object's identifier falls within the range it is responsible for. DHT-based systems have an identifier space where nodeIds and objectIds (*key*) are uniformly generated. Given a key, *put(key, object)* stores the object to the node corresponding to the key. To retrieve an object with a given identifier *key*, *get(key)* is called. Both of these two operations need an overlay routing algorithm to send the request to the peer responsible for the key.

Pastry

Among all the variants of DHT-based overlay schemes such as P-Grid[1], CAN[18], Chord[22], Pastry[19], Taspestry[29], Kademilia[15], Pastry is one of the most well-known. Since our system is initially designed and evaluated

on Pastry, we now provide some background information on it.

In Pastry, each peer or content object is assigned a 128-bit identifier to be a nodeId or objectId (key[1]). A nodeId is used to position the node in a circular Id space, which ranges from 0 to 2^{128}-1. Scattered on the Id space, each node is responsible for a fraction of the Id space (with each key corresponding to an object, this acts exatcly like a hash table. Thus, this type of methodology is called Distributed Hash Table). When an object's Id falls into a specific Id range, this object is stored on the node corresponding to this Id range. Particularly, Pastry stores an object to a node whose nodeId is the numerically closest to the objectId.

Each node maintains a routing table and a leaf set. A routing table, as shown in Fig 2.1, is designed with $Log_B N$ (B=2^b) rows, where each row holds B-1 number of entries. The B-1 number of entries at row n of the routing table each refer to a peer who shares the current peer's NodeId in the first n digits, but whose $(n+1)^{th}$ digit has one of B-1 possible values other than the $(n+1)^{th}$ digit in the current peer's NodeId. Each entry in the routing table contains the IP address of peers whose NodeIds have the appropriate prefix, and it is chosen according to a close proximity metric.

Now we give a simple example of how Pastry uses its routing table to perform a lookup function. On the pastry network illustrated in Fig 2.2, we suppose node *0x65a1fc* wants to find the object with key *0xde74f7* which is supposed to be stored on node *0xde7400*. A lookup message with the object key is sent out from node *0x65a1fc* first. Then the sender looks up his local routing table to decide the next hop for this message. In this case, the next

[1]We use objectId and key interchangeably throughout this thesis

0x0x	0x1x	0x2x	0x3x	0x4x	...	0xDx	0xEx	0xFx
0x30x	0x31x	0x32x	...	0x37x	0x38x	...	0xEx	0xFx
0x370x	0x371x	0x372x	...	0x37Ax	...	0x37Dx	0x37Ex	0x37Fx
0x37A0x	0x37A1x	0x37A2x	...	0x37ABx	0x37ACx	0x37ADx	0x37Ex	0x37Fx

Figure 2.1: Routing Table of a Pastry node with NodeId 37A0x with b=4

hop is *0xd0320b* which shares the first digit with the object Id. The rest of this routing process acts in a same way on each intermediate node. As we can see from Fig 2.2, the lookup message gets closer and closer to the target Id, and finally gets to the destination node *0xde7400*. The actual object stored by this key can then be retrieved by the requesting node. This routing process takes no more than $Log_B N$ hops.

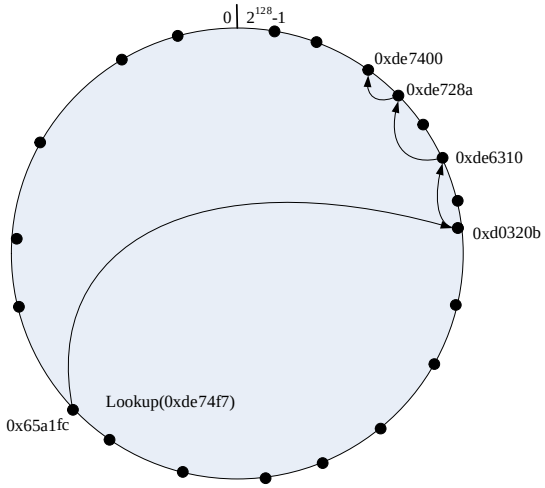

Figure 2.2: Looking up an object in Pastry overlay network

To maintain the overlay, each node has a leaf set L which consists of

$|L|/2$ peers with numerically closest larger nodeIds and $|L|/2$ peers with numerically smaller NodeIds. A typical value of $|L|/2$ is B. Even with concurrent peer failures, a message is guaranteed to be delivered unless $|L|/2$ peers with adjacent NodeIds fail simultaneously. To reduce the risk of a simultaneous failure of nodes that are geographically close to each other, which is possible to happen in reality, a uniform hash function such as SHA-1 is chosen to distribute nodes and objects randomly on the Id space.

Because node failures may cause data loss in the whole network, a data preserving technique through replication has been developed specifically for Pastry, called PAST[10].

2.2 Publish/Subscribe Background

2.2.1 Elements of a Publish/Subscribe System

A generic Pub/Sub system (also known as Event Service or Notification Service) is composed of a set of nodes distributed over a communication network. Clients of the systems are classified according to their role as publishers, which act as producers of information, and subscribers, which act as consumers of information. Instead of communicating directly among themselves, subscribers and publishers are decoupled: interaction of subscribers and publishers relies on the intermediate nodes of the pub/sub system. This decoupling is a desirable characteristic for a distributed communication system because applications can be made more independent from the communication issues, avoiding having to deal with aspects such as synchronization or the need for publishers to address their subscribers directly.

Operationally, the interaction between client nodes and the pub/sub system takes place through a set of basic operations that can be executed by clients on the pub/sub system and vice versa (Fig 2.3). A publisher injects a piece of information e (i.e. an event) to the pub/sub system by executing the *publish(e)* operation. On the subscribers' side, interest in specific events is expressed through subscriptions. A subscription, s, is a filter over a part of the event content (or the whole of it), expressed through a set of constraints. A subscriber can install and remove a subscription s from the pub/sub system by executing *subscribe(s)* and *unsubscribe(s)* operations respectively.

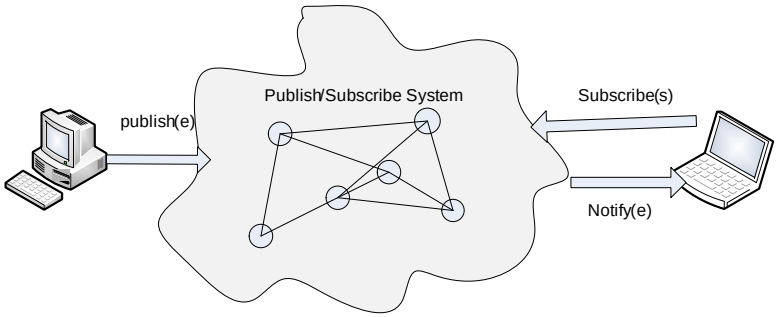

Figure 2.3: Elements of a Pub/Sub System

We say an event e matches a subscription s if it satisfies all the declared constraints of subscription s. The task of verifying whether an event e matches a subscription s is called matching.

2.2.2 Content-based Publish/Subscribe Model

Various ways of expressing interests have led to distinct variants of the pub/sub paradigm. The pub/sub models that have been widely adopted

are generally categorized according to their expressive power: topic-based model and content-based model.

In a topic-based model, a subscriber describes his interest by using only a topic name and will be notified of all events related to that topic. A topic-based model is also sometimes called the channel-based model. Due to the coarse grain expressiveness, a multicast algorithm is usually adopted to disseminate events to numerous interested subscribers.

The content-based model, on the other hand, is much more expressive. In this model, subscriptions can be described using a set of attributes, each of which has a value specified by the subscriber. Our system, based on a popular model proposed by Fabret et al.[12] in 2001, adopts a schema described as: $\mathbf{S}= \{A_1, A_2, A_3, \ldots, A_n\}$, where each element of **S** corresponds to an attribute. Each attribute has a name, type and domain, and can be specified by a tuple {**Name**: **Type**, **Min**, **Max**}. The attributes are identified by their unique name, which can be a unique namespace followed by the actual attribute name. The type could be integer, float, string, etc. The values **Min** and **Max** define the range of domain values of the given attributes.

A subscription is a conjunction of predicates over one or more attributes. Each predicate specifies a constant value (using $=$) or a range (using $<$, $>$, $\leq$, $\geq$) for an attribute. If a subscriber needs to specify multiple predicates over the same attribute, we can model such a subscription as a combination of multiple subscriptions, each of which specifies one continuous range over the attribute. For simplicity of presentation, henceforth, we assume each subscription specifies a continuous range over one attribute. An example of

a subscription is $s = \{(A_1 < v_1) \wedge (v_2 < A_2 < v_3)\}$. An event is a set of equalities over the attributes $\in \mathbf{S}$, which can be expressed as $e = \{A_1 = c_1, A_2 = c_2, A_3 = c_3, \ldots, A_n{=}c_n\}$.

An event e matches a subscription s if each predicate of s is satisfied by the value of the corresponding attribute specified by the event e. A subscription s might not contain every attribute of the schema, but it is a match to an event as long as the event satisfies the predicates that the subscriber has specified. The main functionality of a pub/sub system is to store the subscriptions and given an event, find all the subscriptions matching the event and deliver the event to the subscribers.

2.3 Related Work

In this section, we review previous work related to our research. We start from the non-P2P approaches to pub/sub sytems including central server based and distributed broker sever based approaches. Then we discuss some work which has been done on P2P networks.

2.3.1 Non-P2P Approaches

Centralized approaches such as[14] [8] rely on a central server to store subscriptions and match events with subscriptions. DBMS or some other special data structure[12] are usually utilized. As we mentioned in Chapter 1, these approachese suffer a serious scalability problem as the number of subscriptions and events increase.

In order to improve the scalability, many distributed broker server based

Pub/Sub systems[3] [25] [7] [6] [5] have been proposed using routing trees to perform event delivery through multicast techniques. In Gryphon[3], events are matched with subscriptions on a matching tree that is constructed in the pre-processing phase. Based on multicast, event delivery is performed by the link matching algorithm. In this algorithm, brokers are assembled in a decision tree which an individual broker uses to determine which subset of its neighbours it should send an event to. In Siena[6], a new subscription is stored and forwarded from the originating server to all the broker servers in the network. This forms a tree that connects subscriber with servers. Notifications are then routed towards the subscriber following the reverse path of the tree. In spite of the various designs, they all share the problems of: pre-deployed broker overlay, cost of ownership, no self-organization. In addition, the scalability to an Internet-scale deployment has not been verified yet.

2.3.2 P2P Approaches

With features such as decentralization, share cost of ownership, self-organization, resilience to fault, P2P overlays are promising substrates for Internet-scale applications. Many attempts have been made in designing a P2P-based pub-/sub system. We focus on reviewing those based on DHT in this section.

Topic-based Systems

Scribe[20] and Bayeux[31] are two representative topic-based pub/sub systems built on top of Pastry and Tapstry respectively. An application-layer multicast tree is explicitly formed and maintained to disseminate the events

to subscribers. Not only are these approaches not expressive enough, but they also incur high maintenance cost of maintaining multicast tree.

Content-based Systems

Terpstra et al[23] proposed Rebeca, a content-based pub/sub system built on top of Chord. This system needs to maintain the invariants for filters, which is inefficient in the case of frequent node joins and departures. Triantafillou et al.[24] developed a content-based pub/sub system also on top of Chord. According to a preset precision, a range of values is divided into some discrete values that are stored to the ring. Therefore, a subscription is stored into nodes which are supposed to be the root nodes of these discrete values. The main drawback of their system is that subscription installation and management are expensive if a large number of nodes and messages are involved such as when a subscription's range is big and the precision is high. Zhu et al. proposed another system, Ferry[30], which is based on Chord as well. Based on the name of each attribute, Ferry hashes each attribute to the ring to act as a rendezvous point (RP) for this attribute. This system obviously doesn't scale to a large number of subscriptions with limited number of attributes because only $|S|$ nodes actually process the subscriptions and events. To overcome this problem, an extension to Ferry was proposed recently, called eFerry[28]. In this system, instead of hashing each attribute, a vector of attributes is hashed to the ring, which increases the number of RP nodes. However, this compromises the event matching performance because it has to investigate each subset of the whole attribute set, the number of which is exponential to $|S|$.

Besides the attribute based approaches reviewed above, another research direction is a multi-dimensional treatment, such as Meghdoot[13] and HyperSub[27], which treats the entire schema as a multi-dimension space. Meghdoot maps a 2n ($n = |S|$) dimensional space to the CAN[18] DHT network which can handle multi-dimensional searches in nature. However, it's not easy to adapt this approach to other DHTs. More importantly, it's not able to support multiple schemas with different dimensions. HyperSub, a newly proposed system, leverages a multi-dimensional locality-preserving hashing scheme which sacrifices DHT's load balance nature by changing a random hashing such as SHA-1 to a locality-preserving hashing. In this system, the number of nodes which a subscription is installed to could be of an exponential magnitude. Moreover, both of these two approaches are not flexible to changes on schema, such as adding or deleting attributes in a schema. Since one of our main goals is flexibility, our system takes the attribute based approach.

Chapter 3

System Design

In a content-based pub/sub system, a subscription is a conjunction of predicates over one or more attributes. Each predicate specifies a range of values for an attribute. An event normally is a set of equalities over each attribute in the schema. A subscriber will be notified of any event that matches his interest which is expressed in a subscription. To accomplish this functionality on DHT, two key problems need to be resolved: 1) Given a subscription, which node(s) on the overlay should it be stored to? 2) Given an event, which node(s) on the overlay should be queried to find matching subscriptions? In this chapter, we present our system design to address these two problems. We first overview our system in Section 3.1 and then detail our system's design in Sections 3.2-3.4. Lastly, fault tolerance is investigated in Section 3.5.

3.1 System Overview

Our system aims to serve as a platform to host multiple content-based pub-/sub services with unique schemas. For simplicity of expression, we base our discussion on a pub/sub schema $\mathbf{S} = \{A_1, A_2, A_3, \ldots, A_n\}$, which will be used in the rest of our thesis.

In our system, we map each attribute's one dimensional domain ($[Min, Max]$) to a two dimensional square logic space ($[Min, Max]$, $[Min, Max]$). In this way, a range in one dimensional space becomes a point in the 2d space. Leveraging a 2d tree, we decompose the 2d space into smaller subareas. Each small area corresponds to a node in the 2d-tree which is distributed onto the DHT through hashing each node's identifier. A subscription is stored in the form of a tuple (*subscriber*, *sid*, s), where *subscriber* is the subscriber's handle in the DHT, including its Id and IP address, *sid* is the local subscription id. And s is the subscription that this subscriber has submitted, including its content. When an event is published, it is sent to all the nodes that store the potentially matching subscriptions to match, and then this event is delivered to the interested subscribers.

Our system consists of three key mechanisms:

- Logical Space Mapping and distributed 2d-Tree (Section 3.2).
- Subscription Installation (Section 3.3).
- Event Publishing and Delivery (Section 3.4).

3.2 Logic Space Mapping

For ease of exposition, we suppose there is a schema S = {(*Attribute1*: *float*, $[Min, Max]$)} which has only one attribute *Attribute1* with type *float* and $[Min, Max]$ as its domain. A subscriber submits a subscription s = {*Attribute1*: $v_1 \leq v \leq v_2$} and there is an event e = {*Attribute1*: $v = v_3$} with $v_1 < v_3 < v_2$, which will be injected into the system. So when e is published,

the subscriber of s should be able to be notified of this event.

The first problem is how to store subscription s into the DHT overlay. Simply hashing the min and max values of the range to the DHT network makes the event matching very difficult to process because a random hash function loses all the semantics behind this range: v_1 is less than v_2, and any value larger than v_1 and less than v_2 is in this range.

To overcome this problem, we therefore map this one dimensional space to a two dimensional space, as shown in Fig 3.1, by taking the minimum and the maximum value of this range as the x and y coordinates on the 2d space. Thus, attribute *Attribute1*'s domain $[Min, Max]$ become a square shown in Fig 3.1. Range $[v_1, v_2]$ in 1d space becomes a point in 2d space with coordinate (v_1, v_2). Similarly, event e with *Attribute1* $= v_3$ is mapped to a point on the diagonal of the 2d square. From Fig 3.1, we can infer that all the subscriptions that match this event are in the up-left grey rectangle area i.e. the target searching area of the event matching process. With this logic space mapping strategy, the semantic implication of the original subscription is reconstructed if we can search the target area efficiently within this 2d space.

3.2.1 2d-Tree

To search all the points in an area on the 2d space, we adopt the traditional kd-tree[4] searching algorithm. Essentially, we build a 2d-tree structure for this 2d space. As illustrated in Figure 3.2, the 2d space is divided on the x-axis and y-axis alternatively as the tree spans, and all the points (subscriptions) are stored in the leaf nodes. For example, in Fig 3.2(a), given

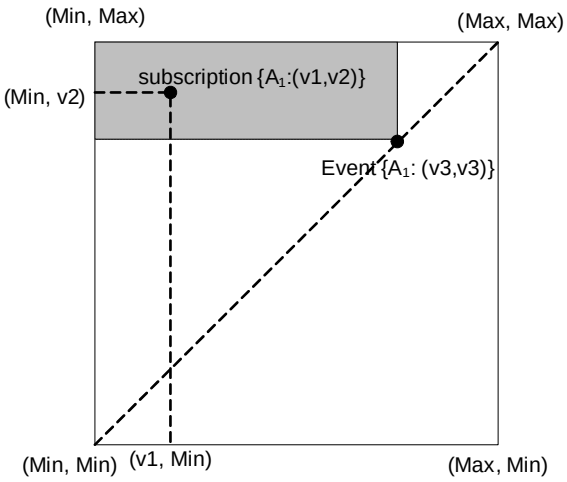

Figure 3.1: Logical Space Mapping

the domain of attribute *Attribute1* as $[0, 100]$, the corresponding 2d space is $\{[0, 100], [0, 100]\}$.[2] A 2d-tree of this 2d space is shown in Fig 3.2(b). As we can see, the aggregation of all leaf nodes {A, B, C, D, E, F, G} covers the full potential searching area. Note that only the area above the diagonal is used in our scheme since a 1d range (x, y) implies $y \geq x$. So the bottom-right area is never used. In Fig 3.2, Subscription $s = \{Attribute1\colon\ 10 \leq v \leq 80\}$ falls into the area that leaf node D is responsible for. When an event $e = \{Attribute1\colon\ v = 15\}$ is published, leaf nodes A, C and D are queried since they overlap the target searching area of this event which is the grey area shown in Fig 3.2(a). Then subscription s will be successfully matched to this event on node A.

[2]We use $\{[xmin, xmax], [ymin, ymax]\}$ to denote a rectangular area throughout this thesis where $[xmin, xmax]$ is the range on the x axis, $[ymin, ymax]$ is the range on the y axis. Note that the inclusiveness of xmin, xmax, ymin, ymax is not always true. It can be denoted as $\{(xmin, xmax], [ymin, ymax]\}$ if xmin is not in this area.

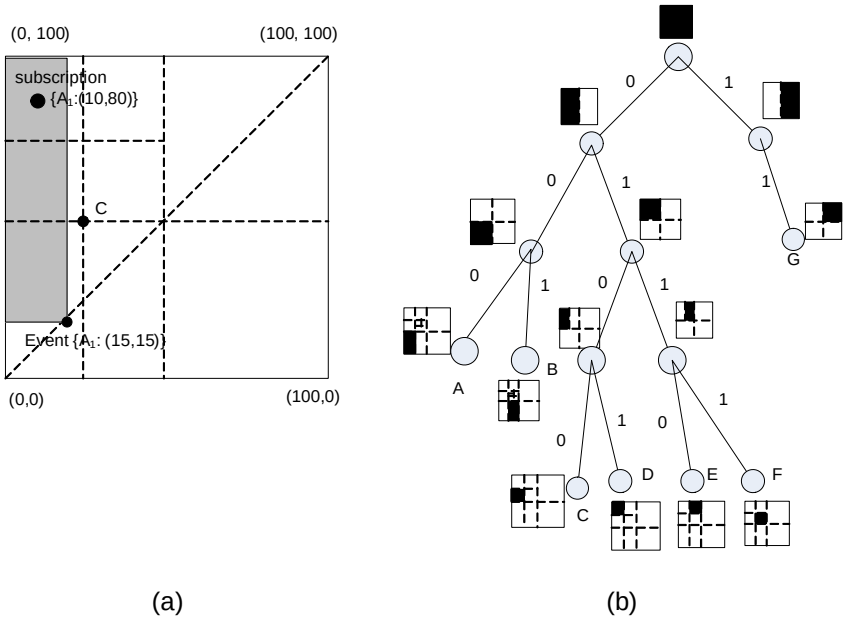

Figure 3.2: 2d-tree Structure

To deploy our solution over a DHT, the tree needs to be able to function in a distributed manner. Therefore, we assign each node in the tree a unique identifier, and through hashing this identifier, id = *hash (identifier)*, an Id within the DHT space is generated. Based on this Id, a peer in the DHT is given responsibility of each node. By treating this 2d-tree as an ordered binary tree, we can obtain each node's unique identifier as a string of 0s and 1s following the current attribute's name. We define this identifier in a recursive way: Assume a current node's identifier is *"x"* with its 2d area as $\{[x1, x2], [y1, y2]\}$ and its two child nodes as $\{[x_1, (x_1 + x_2)/2), [y_1, y_2]\}$ and $\{[(x_1 + x_2)/2, x_2], [y_1, y_2]\}$ or $\{[x_1, x_2], [y_1, (y_1 + y_2)/2)\}$ and $\{[x_1, x_2], [(y_1 + y_2)/2, y_2]\}$, then these two children's identifiers are *"x0"* and *"x1"* respectively. The root

node's identifier is this attribute's name. For example, in Fig 3.2, node B's identifier is *"Attribute1001"*, assuming attribute1's name is *"Attribute1"* in the schema.

Leveraging the logical space mapping and 2d-tree techniques, we have successfully reconstructed the semantics of a range of interest over a DHT overlay. In the next section, we will discuss how to maintain this tree over a DHT.

3.2.2 2d-Tree Maintaining

To lower the maintenance overhead imposed by the tree, the tree is maintained in a lazy manner. We do not physically maintain the links between the nodes in the tree over DHT, because:

- Extra physical links between peers cause too much overhead for updating or exchanging link information periodically to maintain the links, especially during a network churn such as frequent nodes joining and leaving, flash crowds, etc.

- By exploiting the underlying DHT functionalities, it simplifies our system's design and deployment. Utilizing DHT's fault resilience increases the robustness of our system without the need to handle network dynamics.

Without a physical link between nodes, a traversal from a parent node to a child node is accomplished through a *lookup*(child node's id) query in DHT. For example, in Fig 3.2, node B's parent node needs to issue a

lookup(*hash*("Attriubte1001")) to find which peer is storing node B. Compared to the physical link based approach, this approach obviously affects the searching latency. Since pub/sub is not a delay sensitive application, we believe this compromise is worthwhile. Furthermore, as we will later show in Chapter 4, the incurred latency is still acceptable even within large scale P2P networks.

Taking peer heterogeneity into consideration, our tree evolves on the DHT when new subscriptions are added into the system. A node splits into two child nodes if this node's current load exceeds its capacity. For example, if a leaf node can store up to 10 subscriptions, it will split by dividing its current range into halves on the x-axis or y-axis when the load hits 11 subscriptions. Note however that if the sum of two child nodes' load is below their parent node's capacity again because of subscription cancellations, we do not merge these two leaf nodes back to their parent node in our scheme since:

- It will need leaf nodes knowing each other's current load and their parents' capacity in real time. Since this information is dynamic, it can cause overhead when they try to update each other's knowledge by exchanging messages.
- Within a short time, the parent node may need to split again after some new subscriptions are inserted to this node, which makes merging pointless.

As it evolves, the tree grows deeper because of node splitting. Deep trees take very long to search if a search follows the tree links. Additionally,

different branches of the tree may have different heights. A non-leaf node doesn't have any knowledge of the height of its branches.

Thus, each node in our system keeps a routing table containing information about the leaf nodes which are its descendant nodes. As shown in Table 3.1, a routing table consists of three fields:

- Attribute Name: This states which attribute this leaf node is based on.
- Area: This specifies which area in the attribute's 2d space this leaf node is responsible for.
- Leaf Node Identifier: This stores the unique identifier of this leaf node.

Attribute Name	Area	Leaf Node Identifier
Attribute1	$\{[0, 25), [50, 75)\}$	Attribute10100
Attribute1	$\{[0, 25), [75, 100]\}$	Attribute10101
Attribute1	$\{[25, 50), [50, 75)\}$	Attribute10110
Attribute1	$\{[25, 50), [75, 100]\}$	Attribute10111

Table 3.1: A LeafNodeRoutingTable of a node

Attribute Name and Area act as a multi-field primary key to identify a leaf Node identifier. Table 3.1 shows an example of the routing table kept by node "Attribute101" in the 2d-tree in Fig 3.2(b). We name this routing table LeafNodeRoutingTable in our system. With this routing table, a search does not need to strictly follow the links in the tree because every node knows where the leaf nodes are. For example, in Fig 3.2(b), if a query for area $\{[0, 25), [75, 100]\}$ has reached node "Attribute101", it can be directly answered by this node. With one $lookup(hash(\text{"Attribute10101"}))$ message,

this query can be forwarded to its destination node D. By providing shortcuts to the leaf nodes, our LeafNodeRoutingTable obviously improves the search performance and facilitates our random probing search algorithm which we will discuss in the next section.

Unlike a regular p2p routing table, our LeafNodeRoutingTable, as shown in Table 3.1, does not keep hard links either. It only keeps soft links pointing to the leaf nodes, which are the identifiers of the leaf nodes. As we discussed before, this exempts our system from dealing with underlying network maintenance.

Instead of updated proactively, this routing table is updated in a reactive manner. When its load hits its capacity, a leaf node splits into two leaf nodes. Program 3.1 gives the pseudo code of how a node splits and how a LeafNodeRoutingTable gets updated. First, it locates two peers for the two newly generated leaf nodes based on the two new leaf nodes' identifiers which can be decided by the current node locally (line 17-41). Leveraging DHT's lookup service, this task can be easily done. Since all the content objects, which in our system are subscriptions, are stored on leaf nodes, three operations need to be done on the current node's subscription repository: 1) The current subscription objects are divided into halves for the two new leaf nodes(line 42-44) ; 2) These two new sets of subscription objects are transferred to the two new leaf nodes respectively(line 45-50); 3) The subscription objects are deleted on the current node after transfer is done. By now, our node has become a parent node. Hence, it must update its LeafNodeRoutingTable by creating two records pointing to newly created leaf nodes (line 52). To inform the ancestor nodes of this change, this node

sends a LeafNodeUpdateMessage to its parent node, which specifies the attribute name and which area of this attribute's space is splitted (line 54-55). When its parent node receives this message, it updates its own LeafNodeRoutingTable (line 63) and forwards this message to its parent again (line 65). This message is forwarded recursively until it reaches the root node.

Although this routing table boosts our search performance, we have to point out that this routing table is not a requisite of our system. In our system, a leaf node routing table doesn't have to contain the latest leaf nodes information. Some obsolete leaf node information doesn't affect our system's correctness since our system only grows and does not shrink. It only affects the system's search performance. In a deployment on the Internet, LeafNodeUpdateMessage can be sent by UDP instead of TCP to save computing resources. A node can also merge some leaf node records locally to save memory usage because nodes close to the root might contain too many records in their LeafNodeRoutingTable. This merge doesn't affect our system's correctness either.

Having discussed about logical space mapping and a 2d distributed tree structure, we now present how these techniques are used to build our Pub-/Sub system.

3.3 Subscription Installation

When a user wishes to subscribe some events, he firstly expresses his interest through a subscription language, which we introduced in Chapter 2. This subscription is then delivered to the underlying DHT to be stored.

Program 3.1 Leaf Node Split Algorithm

```
1  /*
2   * @param node the current peer
3   * @param subs the subscriptions asscociated with this area stored on this node
4   * @param identifier the unique identifier of this node
5   * @param attr the current attribute
6   * @param area the area to split
7   */
8  public void nodeSplit(node, subs, identifier, attr, area){
9          int level = identifier.getCurrentlevel();
10         Identifier subAreaIdentifier0;
11         Area subArea0; // the area with subAreaIdentifier0;
12         SubscriptionCollections sub0; //the subscriptions with subArea0;
13         Identifier subAreaIdentifier1;
14         Area subArea1; // the area with subAreaIdentifier1;
15         SubscriptionCollections sub1; //subscriptions with subArea1;
16         //to split the area
17         if(level % 2 == 1){
18                 //current level is on x axis, it should be divided on y axis
19                 float ymid=(area.getYRange().ymin + area.getYRange().ymax)/2.0f;
20                 YRange bottomYrange= new YRange(area.getYRange().ymin, ymid);
21                 YRange upperYrange = new YRange(ymid, area.getYRange().ymax);
22
23                 subArea0 = new Area(area.getXRange(), bottomYrange);
24                 subAreaIdentifier0 = new Identifier(identifier +"0");
25
26                 subArea1 = new Area(area.getYRange(), upperYrange);
27                 subAreaIdentifier1 = new Identifier(identifier +"1");
28
29         }else{
30                 //current level is on y axis, it should be divided on x axis
31                 float xmid=(area.getXRange().xmin + area.getXRange().xmax)/2.0f;
32                 XRange leftXrange= new XRange(area.getXRange().xmin, xmid);
33                 XRange rightXrange = new XRange(xmid, area.getXRange().xmax);
34
35                 subArea0 = new Area(leftXRange, area.getYRange());
36                 subAreaIdentifier0 = new Identifier(identifier +"0");
37
38                 subArea1 = new Area(rightXRange, area.getYRange());
39                 subAreaIdentifier1 = new Identifier(identifier +"1");
40         }
41
42         sub0 = subs.split(subArea0); //get all the subscriptions in subArea0;
43         sub1 = subs.split(subArea1); //get all the subscriptions in subArea1;
44
45         //to locate two new leaf nodes
46         Node node0 = node.lookup(hash(subAreaIdentifier0));
47         Node node1 = node.lookup(hash(subAreaIdentifier1));
48         // transfer subscriptions to two children
49         node.send(node0, new TransferSubMessage(attr, subArea0, sub0));
50         node.send(node1, new TransfterSubMessage(attr,subArea1, sub1));
51         //update myself's LeafNodeRoutingTable
52         leafnoderoutingtable.update(attr, area, identifier);
53         //send an update message to my parent
54         Node parentnode = node.lookup(hash(identifier.subString(0, identifier.
               length()-1)));
55         node.send(parentnode, new LeafNodeUpdateMessage(attr, area, identifier));
56 }
57
58
59 public void receiveMessage(Message msg){
60         if(msg instanceof TransferSubMessage){
61                 node.updateStorage(msg); //stored the subscriptions
62         }else if(msg instanceof LeafNodeUpdateMessage){
63                 node.leafnodeRT.update(msg); //update my LeafNodeRoutingTable
64                 Node parentnode = node.lookup(hash(parentidentifier));
65                 node.forward(parentnode,msg); //forward this message to my parent
66         }
67 }
```

Our system randomly chooses an attribute out of the attributes specified in the subscription to be the base attribute. This attribute's value is used as the key to decide which node in the DHT this subscription should be stored in. Assuming that the base attribute chosen is A_i {"Attributei", float, min, max} and the value of A_i in this subscription is a range $[v_1, v_2]$. The key problem of subscription installation is in deciding how to find the node responsible for this subscription (more accurately, for the value $[v_1, v_2]$ of attribute A_i). In other words, it is a problem of finding the leaf node in the 2d-tree, whose area covers the subscription point (v_1, v_2) in the corresponding 2d space.

A naive way is to search from the root node and follow the tree links all the way down to the correct leaf node. But in a distributed system, this method suffers from a severe problem: it overwhelms the root node since all the query messages are routed through this node. To overcome this problem, we propose a random probing (RndLeafProbe) algorithm which does not overload any node.

The basic idea behind our RndLeafProbe algorithm is that we do not query the root node. Instead, the subscriber selects a random node locally to query. Since the subscription point can be stored in only one of the leaf nodes, one random level l in the tree is enough to decide which random node to probe through dividing the subscription point's coordinate value (v_1, v_2) on the x-axis and y-axis alternatively until level l is reached. Meanwhile, the identifier of this random node in the 2d-tree is automatically obtained, which is used to locate the peer in DHT for this random node. Then a probing message is sent from the subscriber to this random node next. Program 3.2

Program 3.2 Subscription Installation Algorithm

```
1  /*
2   * @param sub the subscription about to be installed
3   * @param subscriber the user sending out this subscription
4   */
5  public void installSub (Subscription sub, Subscriber subscriber) {
6          ArrayList<Integer> attrtoinstall_list = new ArrayList<Integer>();
7          //to decide the key attribute to install
8          Attribute attr = sub.getRandomAttr();
9          //get the range of this key attribute
10         Range range = sub.getRange(attr);
11         //decide a random level
12         short level = getRandomTreeLevel();
13         //the identifier for the random node
14         Identifier idforhash = attr.getName();
15         //start to install subscription
16         //firstly, decide which node this subscription should be stored to
17         Float x = range.getMin(); Float y = range.getMax();
18         Float minx = attr.getMin(); Float maxx = attr.getMax();
19         Float miny = attr.getMin();      Float maxy = attr.getMax();
20
21         //decide the string to hash
22         for(short i=1; i<=level;i++){
23                 if (i % 2 ==1){
24                         //level for x
25                         float mid = minx+maxx/2.0;
26                         if (x< mid)
27                         {
28                                 maxx = mid;      idforhash += "0";
29                         }else if(x>= mid)
30                         {
31                                 minx = mid;      idforhash += "1";
32                         }
33                 }
34                 else{
35                         //level for y
36                         float mid = (miny+maxy)/2;
37                         if (y < mid){
38                                 maxy = mid;      idforhash += "0";
39                         }else if(y >= mid){
40                                 miny = mid;      idforhash +="1";
41
42                         }
43                 }
44         }
45         //look up the peer for the randome node
46         Node randomNode = subscriber.getNode(hash(idforhash));
47         //look up the target leaf node
48         Node leafNode=subscriber.send(node,new ProbeSubMessage(attr,level,sub));
49         //install the subscription to the leaf node
50         subscriber.installSubscription(leafNode, sub);
51 }
52
53 public void receiveMessage(Message msg, Node node){
54         Subscription sub = msg.getSubscription();
55         if (msg instanceof ProbeSubMessage){
56                 if(node.checkStorage(sub))
57                         //the current node is the target leaf node
58                         node.ack(msg.source()));
59                         else if(node.checkLeafNodeRoutingTable(sub))
60                         {//this node knows the leaf node
61                                 Identifier id = node.
                                        getLeafNodeRoutingTableRecord(sub);
62                                 node.forward(hash(id), sub);
63                         }
64                         else {
65                                 //guessing too far, we jump backwards along the
                                        tree
66                                 int newlevel = node.identifier/2;
67                                 Identifier newid = node.identifier.subId(newlevel
                                        );
68                                 node.forward(hash(newid),msg);
69                         }
70         }
71         if(msg instance of InstallSubMessage)   node.install(sub);
72 }
```

outlines this algorithm.

Obviously, this random node might not be the correct leaf node. There are three possible cases:

- Through sheer luck, this random node is the leaf node. In this case, the subscription can be stored to this peer directly.
- This random node is above the leaf node in the tree, i.e. this random node is an ancestor node of the correct leaf node. In this case, by looking up this node's leaf node routing table, the probing message will be directed right to the target leaf node.
- This random node is below the correct leaf node, which means that this random node doesn't exist in the 2d-tree yet. So we need to jump upwards along the tree. A binary search is adopted here so that the next node which will be probed is of level $l/2$, where l refers to the current tree level, and so on.

Once the target leaf node is found, the subscription object is inserted and a subscription is successfully installed into the pub/sub system.

It's worth pointing out that a minor problem still exists in our subscription installation approach. It's possible that some subscription points overlap with some corner points of the sub areas in the square space, for example, the point C in Fig 3.2(a). If there are too many points overlapping with one corner point, node splitting in the node covering this corner point will not split the points into halves, which means that the number of these points remaining is still likely to exceed the child nodes' capacity.

As a consequence of this, the node splitting process can become infinite. In order to solve this, we change the subscription's original range from $[v_1, v_2]$ to $[v_1 - \varepsilon, v_2 + \varepsilon]$. Since the interest range is now widened, there can be some false positives which can be removed by a subscriber's local filtering system.

Another subscription installation scheme adopted in some existing systems [24] is that installing one copy of the subscription into the network for each attribute of the schema, then when an event is published, only one attribute is used to match the event to the subscriptions. In this approach, every attribute that is absent in the subscription is treated as an attribute with value $[min, max]$ where min and max are the minimum value and maximum value of the domain respectively. This approach however is not applicable to our system. If this approach were adopted in our system, the range of values would be mapped to the up-left corner of the square in 2d space. Consequently, this would cause a similar problem as the problem discussed above, except in this case the range can't be widened.

3.3.1 Subscription Management

In our system, a subscription is stored in DHT as a subscription object in a form: SubObject = (*subscriber*, *sid*, *subscription*), where *subscriber* is the handle of the subscriber, such as the subscriber's Id and IP address. *sid* refers to the local subscription id for the subscriber and *subscription* is the actual subscription content. While there certainly are some special data structures or a light-weight database that can be applied as the repository, such investigation is out of the scope of this thesis.

A subscriber marks the base attribute of each subscription used to install the subscription. When a subscriber plans to unregister the subscriptions that has been already installed in the system, the subscriber locates the latest peer of the subscription first in a way similar to the subscription installation process. Then the subscriber sends a request to the peer to remove the subscription. Similarly, a subscriber is also capable of changing his/her interest.

Our system, being flexible, allows adding new attributes to a schema as well, since a new attribute does not affect the subscriptions that have already been installed in the system at all.

3.4 Event Publication, Matching and Delivery

3.4.1 Event Publication

When an event is published, the system should be able to find all the subscription objects that match his event. Because of the nature of DHT, these subscriptions could be scattered all over the network, which means that our system should be able to find only the peers that store potentially matched subscriptions. Specifically, in our system, when an event $e = \{A_1 : v = v_3\}$ is published, as illustrated in Fig 3.1, we need to find all the leaf nodes that are covering the target searching area (i.e. the grey area $\{[min, v_3], [v_3, max]\}$ in Fig 3.1).

This is similar to subscription installation except here we are looking for a set of leaf nodes instead of only one. We introduce a random region probing algorithm (RndRegionProbe) to solve this problem outlined in Pro-

gram 3.3. The basic idea behind RndRegionProbe is similar to RndLeafProbe algorithm. The set of nodes whose areas are overlapping with the target searching area is locally decided by using a minimum and maximum tree level: *minl* and *maxl*. In order to even out the probing load, we do not only select the nodes on level *maxl*. Instead, every node above level *maxl* has a chance to be selected. We chose a breadth-first search algorithm to execute this node selection process. With a list of candidate nodes and an empty set of probe nodes, this process starts from level *minl*. If the current node's area overlaps the target searching area, this node is selected with a probability p (p =1 when this node reaches level *maxl*), otherwise this node is removed directly from the candidate nodes list. If this node is selected, then it is added to the probe node set and removed from the candidate nodes list. If it is not selected, this node's two child nodes will be added to the tail of the candidate list and this node is removed from the candidate list. This process stops when the list is empty. Whereafter, the publisher sends out a ProbePubMessage to each node in the probe set.

Since these are locally selected nodes, it is likely that some of these nodes might not be leaf nodes. By probing them, target leaf nodes can be found. When a node receives a ProbePubMessage, there are three different cases:

- The node is a leaf node. In this case, an event matching can be executed right away.

- The node is above the leaf node in the tree. In this case, this node looks up its LeafNodeRoutingTable to acquire the records of any leaf nodes which overlap the target search area. Then, the ProbePubMessage is

Program 3.3 RndRegionProbe Algorithm

```
1  /*
2   * @param event the event that is published
3   * @param maxlevel the max tree level to probe
4   * @param minlevel the min tree level to probe
5   * @return a set of destinations
6   */
7  public void publishEvent(Event event, int maxlevel, int minlevel){
8          DestinationSet destidset = new DestinationSet();
9      //deciding nodes to probe
10         for( each attribute in the event){
11                 List arealist;
12                 Area target_area = event.getTargetArea(attribute);
13                 arealist.add(attribute.getRootArea());
14                 while(!arealist.empty()){
15                         Area area = arealist.First();
16                         if(area.level < maxlevel){
17                                 if(area.level > minlevel && random()<p)
18                                         destidset.add(area);
19                                 else{
20                                         arealist.add(area.split());
21                                         arealist.remove(area);
22                                 }
23
24                         }else
25                                 destidset.add(area);
26                 }
27         }
28         //sends out prob pub messages to all these nodes
29         sendoutProbPubMessage(event, destidset);
30 }
31
32 /*
33  * @node the current node that receives this message
34  * @msg the message received
35  */
36 public void receiveMesssage(Node node, Message msg){
37         if(msg instanceof ProbePubMessage){
38                 Event event = msg.getContent();
39
40                 if(node.isLeafnode(event))
41                         //if the current node is the leaf node
42                         //deliver this to the application for event matching
43                         deliver(msg);
44                 else if(node.hasLeafRecord(event)){
45                         //if the current node has the leaf node record
46             //for this event in the LeafNodeRoutingTable
47                         //getLeafnodeRecords will get all the nodes that
48             //overlapping this event's target area
49                         leafset = node.getLeafNodeRecords(event);
50                         sendoutProbPubMessage(event, leafset);
51                 }else
52                         //we are guessing too far, need to jump back.
53                         sendoutProbPubMessage(event, new Id(currentlevel/2));
54         }
55 }
```

forwarded to those leaf nodes.

- The node is below the leaf node in the tree. The ProbePubMessage is forwarded to the ancestor node on level $l/2$ where l refers to the current level. A possible problem this might cause is that some nodes are visited multiple times because two different ProbePubMessages can jump upwards to the same ancestor node. Therefore, when a node has already been probed by a probe message, any following probing messages from the same subscription will be discarded on this node.

Once a ProbePubMessage reaches its target node(s), an event matching process starts to extract all the matched subscription objects stored in this node. A simple linear matching algorithm tests each subscription with the event one by one. The matching from an event to a large number of subscriptions could be very inefficient. To overcome this, some *sublinear* matching algorithms such as [2], could be adopted.

As we discussed in section 2.2.2, the content-based publish/subscribe model that we use in our system assumes that an event is a set of equalities over every attribute in the schema. In a practical scenario, however, some publishers may only specify values to a partial set of attributes in the schema. In order to handle this situation in our system, the concept of a match between an event and a subscription needs to be clarified first. For example, given a subscription $s = \{(v1 < A_1 < v2) \wedge (v_3 < A_2 < v_4)\}$ and an event $e = \{(A_1 = v_5)\}$ where $v_1 < v_5 < v_2$, whether the event e matches the subscription s is ambiguous. It depends on the system's definition of match. If this is not match, then our system can handle this case naturally by only

searching the attributes that are present in the event. If this counts as a match, then the constraint of A_2 in s is essentially ignored. Since attribute A_2 might be the base attribute of subscription s, our system needs to search the entire searching space of A_2 in order not to miss a match, s, in this case. In other words, every attribute in the schema needs to be searched and the target search area of any absent attribute is the entire searching space of that attribute.

3.4.2 Application Layer Multicast

As we discussed, a subscription is installed based on one attribute randomly chosen from its attributes. Consequently, when a publisher is about to publish an event, it will need to be published by each of its attributes. According to our current event publishing scheme, a publisher may need to probe a large number of nodes at the same time in the DHT even for one attribute. Using a unicast communication model, probing all the nodes for all the attributes could result in a lot of ProbePubMessages which causes overloading and high bandwidth consumption problems.

In this section, we propose an application layer multicast algorithm to solve this problem. First, let us revisit how messages are routed in Pastry. In Fig 3.3(a), source node 0x65a1fc sends out a lookup message with target Id 0xde74f7. This message is first routed to node 0xd0320b because it shares the first digit with the target Id. As this goes on, the message is routed to nodes which are closer and closer to the target. Considering the case that the source node 0x65a1fc has a list of target Ids to query as shown in Fig 3.3, we can observe that Ids("0xd09b2a", "0xde3981" and "0xde74f7") share the

same first digit *'d'* which incurs that their next hops will be the same node 0xd0320b. Based on this observation, the number of query messages sent out by the source node 0x65a1fc can be reduced by only sending one query carrying these three Ids as a list of target Ids to 0xd0320b instead of sending individual queries for each of these three Ids. Once a target Id arrives at its destination node, it is removed from the target Id list. Assuming node 0xd0320b is the correct destination node for target Id 0xd09b2a in Fig 3.3, Id 0xd09b2a is removed from the Id list at hop 0xd0320b. This process can be repeated on each hop on the route until all target Ids arrive at their destination nodes.

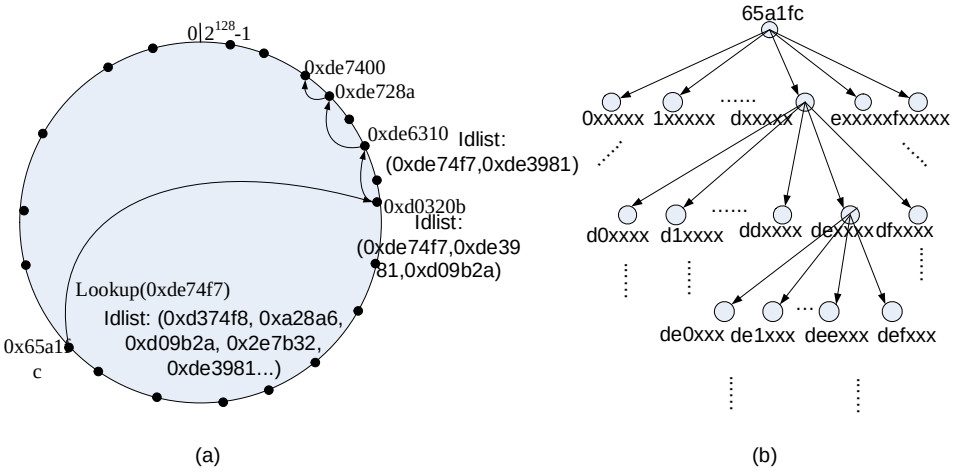

Figure 3.3: Application Layer Multicast on Pastry

The basic idea behind this approach is that at each hop the list of target Ids is regrouped by their shared prefixes and then the message is forwarded to the next hops in groups accordingly. The algorithm is outlined in Program 3.4. This approach essentially forms an application layer multicast

tree, as shown in Fig 3.3(b). The fan-out degree of each node is B at most, where $B = 2^b$ in Pastry. On average the height of this tree is $O(log_B N)$, which implies that our multicast approach does not increase the message delay on average.

It's worth us noting that unlike many other application layer multicast approaches [20] [31] that need to maintain an explicitly built multicast tree, our approach doesn't impose any overhead onto the overlay network. It's just an exploitation of the underlying Pastry DHT. Leveraging this multicast scheme, a publisher's load can be dramatically reduced and the bandwidth consumption of the whole network can be saved significantly.

3.4.3 Event Delivery

After the matching procedure is done on a node, a list of matched subscribers is obtained. Delivering events to these subscribers is trivial. Because a subscriber object includes his IP address, a point-to-point communication can deliver the event to the subscriber. In addition, all the subscriptions in our system are scattered over the nodes in the DHT and each node has a capacity limit. As such, the number matched subscribers on each node is not a concern to overburden the node. In spite of this, a node can still adopt the application layer multicast idea described above to save its bandwidth consumption and local computing resources. The only difference is that in event delivery, the targets are not Ids in DHT but IP addresses, which means that no DHT routing table is needed. Thus, the grouping method of target subscribers does not necessarily conform to their Id prefixes. In fact, any grouping strategy can be used, although grouping according to their Ids in

Program 3.4 Application Layer Mulicast Algorithm

```
1  /*
2   * @param event The event that will be sent out
3   * @param destIdset the set of destionation Id.
4   * @param prefix the current shared prefix of all the ids
5   */
6  public void sendoutProbePubMulticast(Event event, DestinationSet destIdset,
       String prefix){
7          HashMap<String, DestinationSet> destIdset_map = new HashMap<String,
               DestinationSet>();
8          //regroup the destIdset according to new prefixes.
9          for(each destid in destIdset){
10                 String newprefix = destid.toStringFull().substring(0, prefix.
                       length()+1);
11                 if(destIdset_map.containsKey(newprefix)){
12                         destIdset_map.get(newprefix).
13                 addProbePubDest(destid, destIdset.getProbePubCont(destid));
14                 }else{
15                         DestionationSet destSet = new DestionationSet();
16                         destSet.addProbeDest(destid);
17                         destIdset_map.put(newprefix, destSet);
18                 }
19         }
20         //send out the messages by groups
21         for(each IdGroup in destIdset_map){
22                 String newprestr = IdGroup.getKey();
23                 Id randomid = IdGroup.getValue().getRandomDest();
24                 probemsg = new ProbePubMulticastMessage(event,dest,newprefix);
25                 node.route(randomid, probemsg, null);
26         }
27
28         /*
29          * @param node the current node that receives this msg
30          * @param msg the message that is received
31          */
32         public void receiveMessage(Node node, Message msg){
33
34                 if(msg instanceof ProbePubMulticastMessage){
35
36                         //get the destIdset in the multicast message
37                         DestinationSet destIdset = msg.getDestSet();
38
39                         //examine whether I'm the home node for some ids in the
                               idset
40                         for(each id in the destIdset){
41                                 if (node.homenode(id)){
42                                         //deliver this message to my application
43                                         deliver(msg);
44                                         destIdset.remove(id);
45                                 }
46                         }
47                         //forward this multicast message
48                         sendoutProbePubMulticast(msg.getEvent(),destIdset, msg.
                               getPrefix());
49                 }
50         }
51 }
```

the DHT might be the easiest way.

3.5 Load Balancing

Taking a closer look at our 2d space searching space, it is not hard to see that the upper-left corner, illustrated in Fig3-4, is likely to become a very hot search region because it is included in almost every event's target search area. The reason for this is that the points in this region correspond to broad interest ranges, i.e. the ranges whose minimum values are close to their domains' minimum values, and whose maximum values are close to their domains' maximum values. Thus the points in this area have a high chance of matching any event, which requires almost every event to search this area. However, simply splitting this area can't solve the problem because the areas after the splitting will still be in this hot area. Therefore, peers responsible for this area will be overloaded by too many event matching requests.

To overcome this problem, we propose a load balancing scheme for our system by defining a cut-off line ($y = x + c$) for each attribute. As illustrated in Fig 3.4, the area above this cut-off line is cropped from the searching space. The subscription points in this area are mapped to the searching area below this cut-off line. Obviously, a one-to-one mapping can't guarantee not losing any potentially matched events. A subscription point is mapped to several points on a line below the cut-off line, which is $y = x + h$ where $h < c$ in Fig 3.4. In order not to make some areas overcrowded by adding these new points in the area below the cut-off line, h is randomly, locally selected when the subscription is submitted by the subscriber. For example, in Fig 3.4 sub-

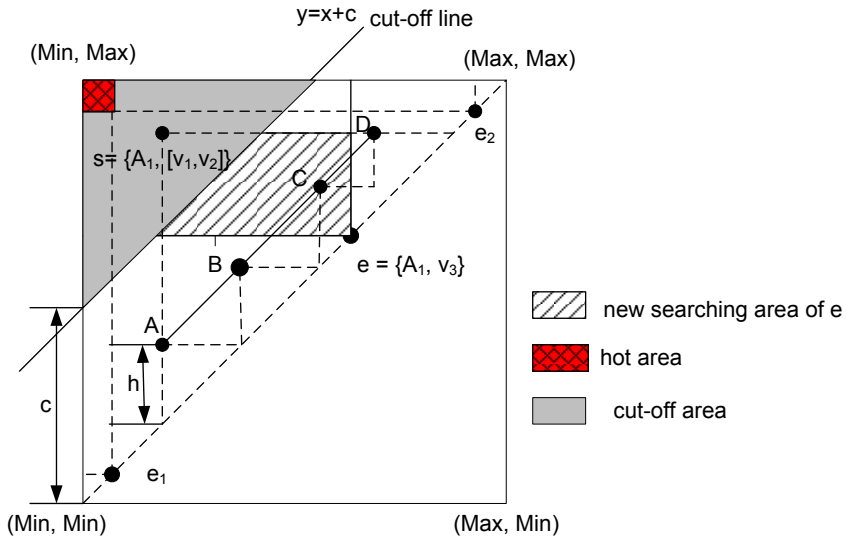

Figure 3.4: Load Balancing Scheme

scription point s = $\{A_1 : [v_1, v_2]\}$ is mapped to (A, B, C and D) four points, whose coordinates are $(v_1, v_1 + h), (v_1 + h, v_1 + 2h), (v_1 + 2h, v_1 + 3h)$ and $(v_1 + 3h, v_2)$ respectively. The target searching area of an event is changed as well. It is the area below the cut-off line instead of the old area containing the up-left corner. Note that a subscriber will not be notified of a same event multiple times using this approach.

3.6 Fault Tolerance

Our initial system design goal is to have our system exploit the underlying DHT infrastructure's robustness and fault resilience to handle faulty nodes. This way not only reduces our system's design, deployment and maintenance complexity, but also makes our system as robust as the underlying DHT. In

this section, we look into the details of how the DHT can be exploited to enhance our system's fault tolerance.

Suppose that a subscription s is installed on node N_i called the root node of s. Then, s is replicated to a set of neighbour nodes of root node N_i. These neighbour nodes$[N_{i-k}, \ldots, N_{i-1}, N_i, N_{i+1}, \ldots, N_{i+k}]$ are the numerically closest nodes to N_i in the Id space so that N_{i-1} or N_{i+1} can automatically become the new root node of s when node N_i leaves. Moreover, keeping several copies of a subscription increases the subscription object's availability and durability in DHT, especially when there is a severe churn in the network such as multiple simultaneous node failures.

As we discussed before, a node in our system is also keeping a LeafNodeRoutingTable. Since this routing table can be changed frequently, it is not replicated to its neighbour nodes to reduce the replica maintenance cost. Instead, we replicate a node N_i's current position in the 2d tree in its neighbour nodes. This position information includes the identifier of this node in the tree and which area in the 2d space this node is covering. Since this position information does not change as the 2d tree evolves, we are spared the cost of frequent updates. After node N_i leaves, its neighbour N_{i-1} or N_{i+1} becomes the home node of N_i's identifier and updates its LeafNodeRoutingTable by grabbing a copy of its parent node's LeafNodeRoutingTable. If a probe message comes in before the LeafNodeRoutingTable is updated, this probe message is simply forwarded to the node's two children nodes.

When a node joins, some concurrency problems need to be considered. Suppose N_{i-1} and N_{i+1} are currently two neighbour nodes in the network. When a new node N_i comes in with its Id between N_{i-1} and N_{i+1}' ids, nor-

mally some contents from N_{i-1} and N_{i+1} are transferred to N_i, including subscription objects and LeafNodeRoutingTable. However, before the transfer process is done, some messages might be delivered to this newly joined node N_i. To ensure our system's correctness, a node will not be marked completely ready until the transfer process is done. Until N_i is ready, all the pub/sub application layer messages are forwarded to N_{i-1} or N_{i+1} to handle. Of course, it can still participate in the DHT layer routing during this time.

As we can see, handling faulty nodes in our system is very simple and has a low overhead. There are two key reasons for this: (1) A thorough exploitation of the underlying DHT; (2) No physical network maintenance imposed on DHT.

Chapter 4

Evaluation

In this section, we evaluate the performance of our proposed architecture through extensive simulation experiments. We start our discussion by describing the experimental setup and parameters used for evaluation. Afterwards, the experimental results are presented and discussed.

4.1 Simulation Setup

4.1.1 Simulator Configuration

We implement our pub/sub architecture on top of FreePastry[26], an actively maintained java implementation of Pastry. Through a virtual socket layer, simulations and interfaces for real Internet-deployable applications are integrated together in FreePastry. The simulator component in FreePastry is discrete event-driven and on transport layer level. Here are some Pastry parameters that are used in our simulation: (1) Number of bits in Id is 160. (2) Length of the routing table base is 4 bits. (3) LeafSet size is 16.

The network model used in the simulation is derived from the King dataset[9], which includes the pairwise latencies of 1740 DNS servers in the Internet measured by the King method. The average round-trip time of the

simulated 1740-node network is 180 milliseconds

Our simulation consists of 3 stages:

- Node joining: The simulator first initializes a certain number of nodes (*num_nodes*) to join the DHT network.

- Subscription installation: After a DHT network is formed, we start a subscription installation process. The interarrival time between subscribing events is exponentially distributed with an average value μ_{sub}. Each subscription is generated based on a pre-defined schema, which we will describe in the next section. A subscriber is randomly drawn from all the nodes alive. Average number of subscriptions per node (*aver_sub/node*) and the number of nodes (*num_nodes*) decide the total number of subscriptions installed, *num_subs* = *num_nodes* * (*aver_sub/node*). A random capacity of subscriptions (*node_capacity*) is also assigned to each node.

- Events publication: After system stabilization, a certain number of publications (*num_pubs*) are scheduled to be injected into the system. Similarly, events are published in a poisson distribution with parameter $\lambda = 1/\mu_{sub}$. The publisher of each event is randomly selected from all the nodes.

Unless otherwise specified, the parameters used in our simulations are the same as those shown in Table 4.1.

num_nodes	μ_{sub}(ms)	*aver_sub/node*	*node_capacity*	*num_pubs*	μ_{pub}(ms)
1000	2000	5	rand(10,30)	10000	1000

Table 4.1: Simulation Parameters

4.1.2 Dataset Configuration

We use synthetic datasets in our simulations. Table 4.2 shows the publish/-subscribe schema that we use in our simulation. The schema has 4 attributes with each attribute having its own name, type and domain.

Attribute Name	Type	Min	Max
Attr1	Float	0	10,000
Attr2	Float	-10,000	10,000
Attr3	Float	0	3,000
Attr4	Float	0	200

Table 4.2: Publish/Subscribe Schema in simulation

To generate subscriptions and events based on this schema, we define a set of properties for each attribute, as shown in Table 4.2. Zipfian distribution is heavily used in our data generating scheme, which is a common distribution of real world datasets. The cumulative distribution function for Zipfian distribution is $H_{k,s}/H_{N,s}$, where $H_{N,s}$ is the N^{th} generalized harmonic number with skew factor s and k is the rank with $1 \leq k \leq N$. In our data generating scheme, a domain's range is divided into some sub-ranges according to a *subrange_size*. These sub-ranges are distributed in a Zipfian distribution with skew factor *subrange_skew_factor*. When an event needs to be generated, a sub-range is selected according to this zipf distribution for each of its attributes. Then a random value is drawn from this range. After a value is generated for each attribute, the event is generated. Generating a

subscription takes a bit more work than an event because for each attribute, a range of interest has to be generated. In our simulation, the distribution of the sizes of interest range also obeys a Zipfian distribution with skew factor *size_skew_factor*. We also have a *size_bound* parameter to specify the maximum size of a range. To generate a range for an attribute, a point is generated from the domain first in a same way how an event is generated. Then, a size is generated based on *size_skew_factor* and *size_bound*. Combining this point value with the size, a range is generated. In practice, not all attributes are required to be present in the subscription. Therefore, we set a present probability *present_prob* for each attribute to decide whether this attribute is in the subscription or not. After all this, a subscription is successfully generated. The values of the aforementioned parameters are listed in Table 4.3.

Attribute Name	present _prob	subrange _skew_factor	subrange _size	size _skew_factor	size _bound
Attr1	0.5	0.6	100	0.8	50%
Attr2	0.5	0.5	20	0.6	40%
Attr3	0.5	0.4	10	0.5	30%
Attr4	0.5	0.3	1	0.4	20%

Table 4.3: Schema Parameters for Simulation

4.2 Simulator Architecture

Our simulation is implemented on the application layer in FreePastry, which makes our simulator independent of lower transport layer. As described in Table 4.4, layers are separated from each other, and are replaceable. This

simulation implementation therefore is flexible and easy to export to a real-world application.

Layer	Description
Application	The application layer provides an application interface to our simulator, which includes the subscribe and publish operations.
Pub/Sub infrastructure core	The Pub/Sub infrastructure core layer is responsible for the core implementation of message processing, routing table maintaining, subscription insertion, event matching and delivery.
DHT(Pastry and PAST)	This layer constructs and maintains a P2P network and provides the distributed data placement and lookup service. PAST is used as a storage preserving enhancement.
DirectTransport Layer / Socket Layer	This layer is the low-level communication layer. For a simulator,the DirectTransport Layer is used to simulate a socket layer. This layer can be replaced with a Socket Layer to export as an application runnable on the Internet.

Table 4.4: Simulator Architecture

4.2.1 Message Types

Seven different types of messages are implemented in our system:

1. ProbeSubMessage. This message is used to probe the home node for a subscription.

2. InsertSubMessage. When a home node is found, the subscription is inserted into the home node through this message.

3. LeafUpdateMessage. This message is sent when a leaf node splits to update node's LeafNodeRoutingTable. It is sent to the node's ancestor

nodes recursively.

4. TransferSubMessage. When a node splits into two new leaf nodes, this message is used to transfer the subscription objects stored on the local node to its child nodes.

5. ProbePubMulticastMessage. The publisher multicasts its probing messages through this message to their destinations.

6. ProbePubMessage. This message is sent out to locate all the nodes that store matched subscriptions. This message is currently sent by unicast instead of multicast.

7. EventDeliveryMessage. An event is delivered to a matched subscriber using this message.

These seven messages basically belong to three categories according to their functionalities: Subscription Installation messages (ProbeSubMessage, InsertSubMessage), LeafNodeRoutingTable maintaining messages (LeafUpdateMessage, TransferSubMessage) and Event Publishing messages (ProbePubMulticastMessage, ProbePubMessage, EventDeliveryMessage).

4.3 Simulation Results

4.3.1 Evaluation of Subscription Installation

In this section, we evaluate the system's performance during the subscription installation phase by looking at the delay of subscription insertions and the

bandwidth cost for each node. The results are obtained from a simulation with a 1000-node network with 5 subscriptions per node on average.

Fig 4.1 shows a latency distribution for subscription insertions. The average latency is 331ms. It shows that about 70% of insertions are done within 600ms, and about 90% of installations take less than 1s.

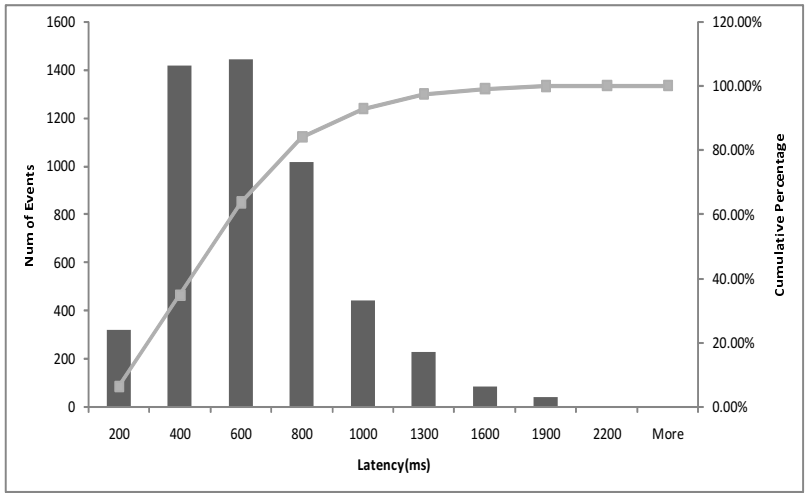

Figure 4.1: Subscription Installation Latency

Fig 4.2 illustrates the bandwidth cost for each node during subscription installation. The messages that are measured include ProbeSubMessages, InsertSubMessages, LeafUpdateMessages and TransferSubMessages. Here, the bandwidth cost is measured per subscription, which means how many bytes a node contributes for each subscription on average. The average bandwidth cost is 14.32 bytes per subscription for each node. As it shows, about 78% percent of nodes' bandwidth cost is within 20 bytes/sub, nearly 98% nodes' bandwidth cost is less than 60 bytes/sub. Although there are

a few nodes whose bandwidth cost is about 100 bytes/sub, the bandwidth cost is still relatively low and acceptable. Thus, we can conclude that our subscription installation mechanism is efficient and incurs low bandwidth cost. Note that the message size is calculated by serializing each message using Java's object serialization method, which has a lot of overhead. In a practical system, the message size can be reduced by using raw serialization and compression.

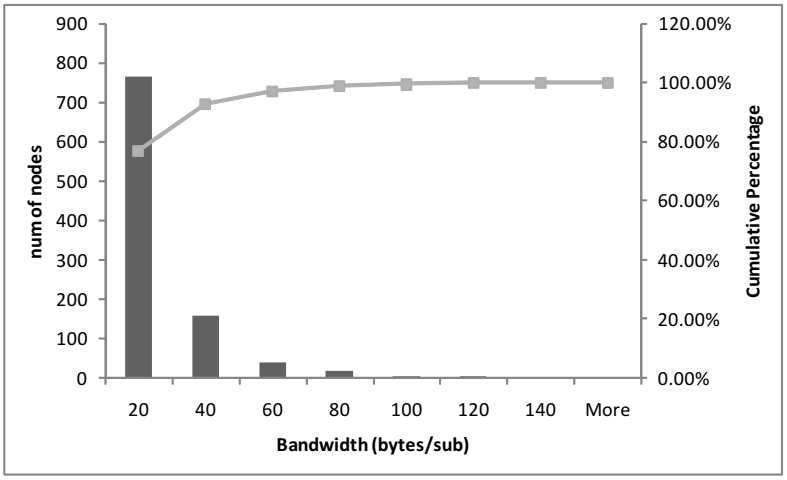

Figure 4.2: Bandwidth distribution during Subscription Installation

4.3.2 Evaluation of Event Publication

In this section, we focus on investigating our system's performance during the event publishing, matching and delivering phase.

We start by studying the latency of events delivery first. The latency is defined as the time it takes for an event to be delivered to an interested subscriber from the time it's published by a publisher. Fig 4.3 shows a

latency distribution for each delivery on a network with 1000 nodes and 10000 events. The average latency is 333ms. From Figure 4.3 we can see that about 70% of event deliveries are done within 400ms and nearly all the deliveries are within 1s. Based on a network model derived from the Internet, our simulation shows that our system can efficiently notify any subscriber of an event that match his interest.

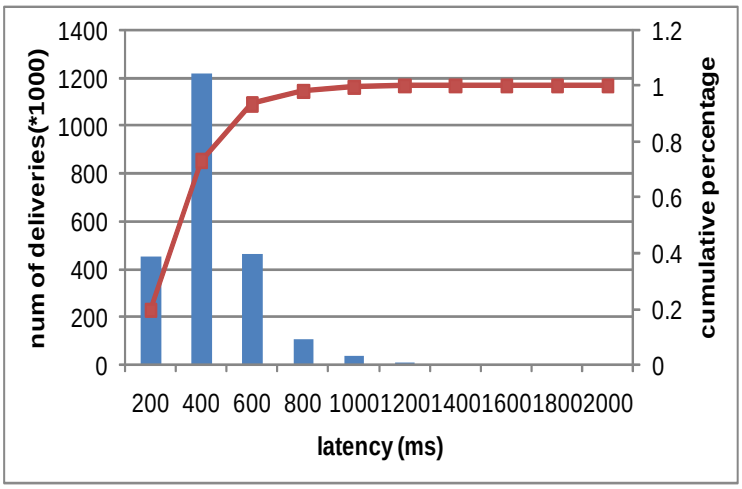

Figure 4.3: Publication latency distribution

We also study the bandwidth consumption of each node during this period of time. The bandwidth cost is measured per event and includes ProbePubMulticastMessages, ProbePubMessages and EventDeliveryMessages. As we explained before, the size of these messages are also decided by Java serialization. The average bandwidth cost is 1.32k bytes per event. As shown in Fig 4.4, it costs no more than 1.6k bytes per event for about 70% of nodes and 3k bytes for about 90% of nodes. There are about 7 nodes that spend more than 6k bytes but no more than 11k bytes. With a skewed distribution

of our subscription and event data, we believe this is still acceptable. Part of our future work is to investigate how to handle highly skewed data.

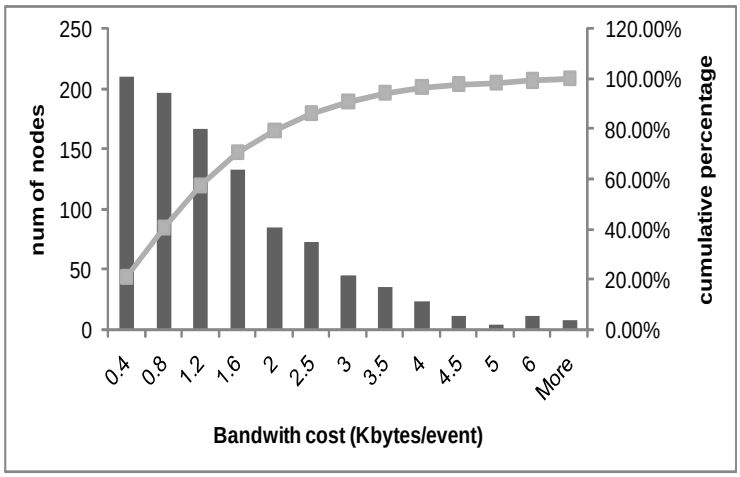

Figure 4.4: Bandwidth cost distribution

Performance of Application Layer Multicast

In this section, we evaluate the performance of our multicast algorithm and load balancing scheme. Fig 4.5 and Fig 4.6 present a bandwidth cost comparison of multicast model and unicast model. Here we ignore the bandwidth cost of EventDeliveryMessages because these messages are sent after the matching process is completed. Fig 4.5 shows the comparison of bandwidth cost distribution. Compared to an average cost of 1.48k in the unicast model, the average cost in the multicast model is decreased by almost 30% to 1.05k. In Fig 4.5, it shows that in the multicast model 70% of nodes' bandwidth cost is less than 1.2k, but in the unicast model only about 55% of the nodes' bandwidth cost is in this range. Fig 4.8 plots a bandwidth

cost against the first 100 nodes with the most bandwidth cost. It shows the maximum bandwidth cost reduced from 12.2k to 9.4k. We observe that all the top 100 nodes' bandwidth costs are decreased. Thus, we can conclude that our multicast algorithm is effective in reducing the bandwidth cost of publishing events. In addition, in a practical application, we believe our multicast algorithm can benefit more because unlike the events used in our simulations, the events can become very big in size, which incurs significantly increase in bandwidth cost in a unicast model.

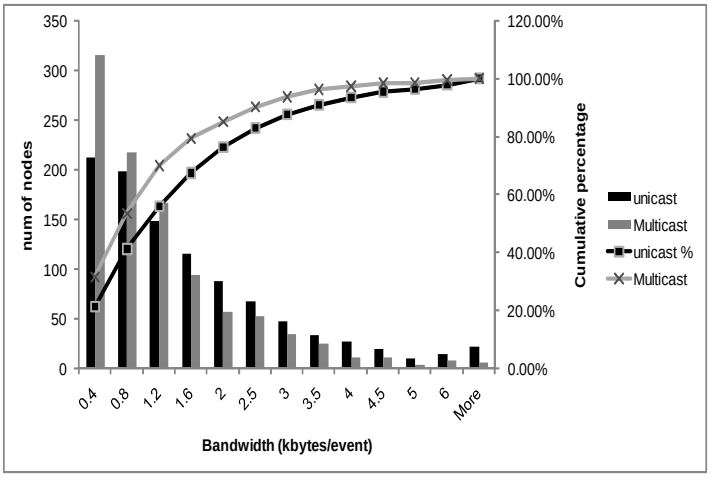

Figure 4.5: A performance comparison of multicast and unicast

Performance of Load Balancing

In this section, we investigate the performance of our load balancing scheme. As with the previous experiments, the cost incurred by EventDeliveryMessage is not factored in. Fig 4.7 and Figure 4.8 illustrates a bandwidth cost comparison of a system with and without load balancing scheme. According

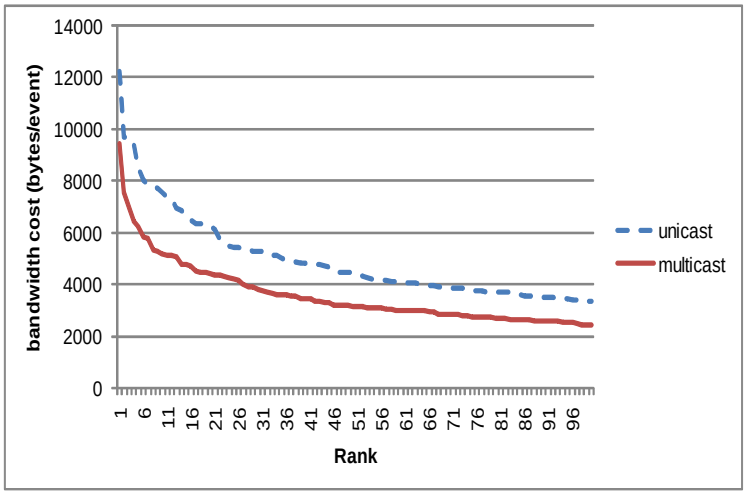

Figure 4.6: A comparison of mulicast and unicast on first 100 nodes

to what we discussed in section3.4.2, there are three kinds of nodes whose load are alleviated by our load balancing scheme: 1)the nodes that cover the inherently hot area in the 2d search space; 2) the nodes that on the searching paths to the hot area; 3) the publisher nodes by not probing any nodes in the hot area. As shown in Fig 4.7, the number of nodes with high bandwidth costs are dramatically reduced, especially the number of nodes whose bandwidth costs are more than 6k. While the average bandwidth is 2.12k bytes/event without the load balancing scheme, our load balancing scheme reduces it by about 50% to 1.05k bytes per event. Fig 4.8 plots the bandwidth cost against the top 125 nodes. From this figure, we can observe that the bandwidth costs of the top 20 nodes are reduced significantly leading to more balanced system.

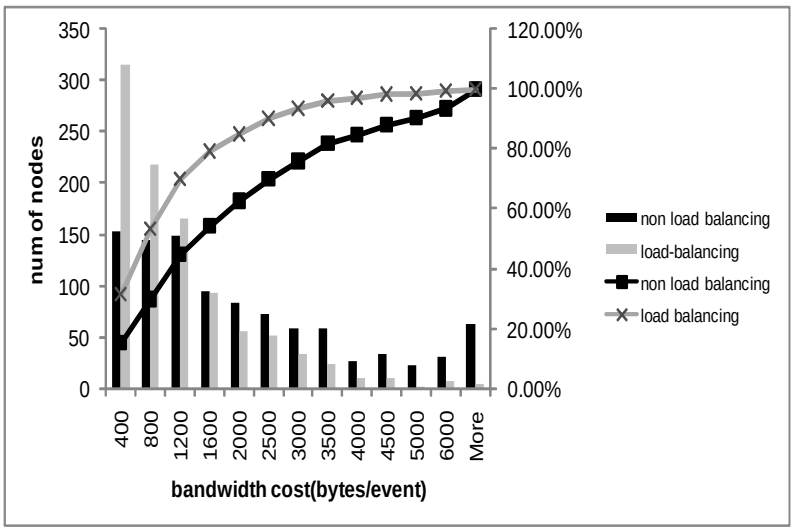

Figure 4.7: Bandwidth cost comparison of schemes with and without load balancing

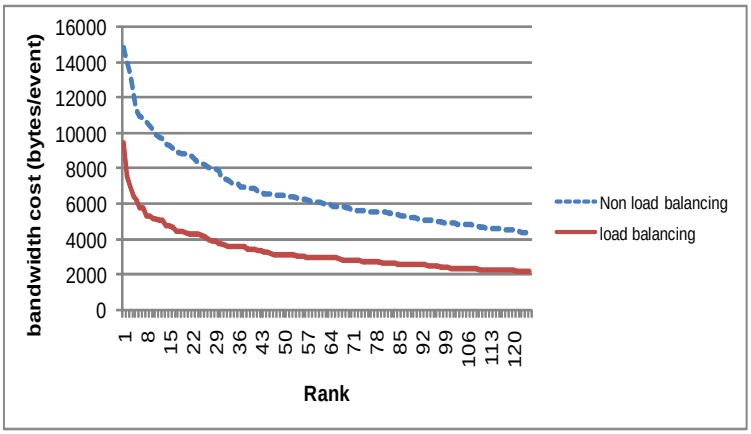

Figure 4.8: A comparison of load balancing performance on top 125 nodes

4.3.3 System Scalability

In this part, we evaluate the performance of our system in networks of various sizes, derived from the King data. The average RTTs are shown in Table 4.5.

size	1000	1740	3000	4000	5000
Avg RTT(ms)	175	180	176	177	176

Table 4.5: RTTs for different networks

Fig 4.9 plots average delivery latency and maximum delivery latency against network size. It shows that average latency slightly increases as network size increases. It is because as the network size increases it takes more hops for a DHT lookup. However, as the network size increases 5 times from 1000 nodes to 5000 nodes, the average latency only increases from 330ms to 430ms which is acceptable for a pub/sub application on a large-scale network. Figure 4.9 also shows that the maximum latency oscillates between 1.6s to 1.8s. It doesn't increase as the network size increases. This proves that our LeafNodeRoutingTable is efficient to locate the leaf nodes and our application layer multicast scheme doesn't incur any extra delay.

Fig 4.10 plots a relationship between bandwidth cost and network size. As it shows, as the network size increases, the average bandwidth cost actually decreases slightly, though the maximum bandwidth cost increases. The reason behind this is that as more nodes exist in the network, more subscriptions are inserted, which causes our 2d-tree to expand more. But in our simulation, we set the maximum probing tree level to 10, which means that the nodes below this level in the tree wouldn't have a chance to help

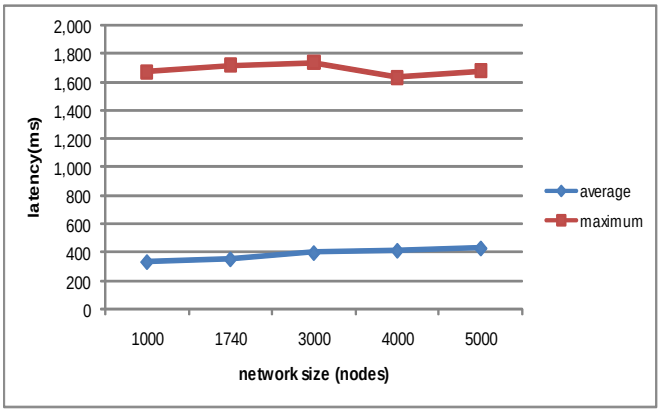

Figure 4.9: Event delivery latencies on different network size

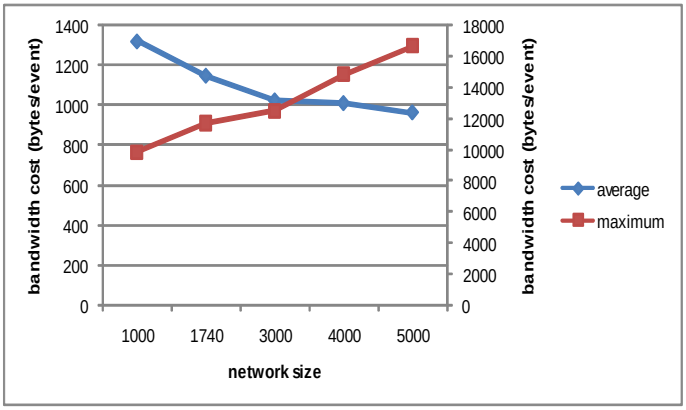

Figure 4.10: Bandwidth cost on different network size

alleviate the probing load, causing the nodes above this level to have to contribute more bandwidth to forward the probing messages to leaf nodes. Therefore, we believe that a dynamic max probing level will solve this problem. From the analysis above, we can conclude that our system is scalable to a large-scale network.

Chapter 5

Conclusion and Future Work

5.1 Conclusions

In this thesis, we have proposed a novel low-cost content-based publish/-subscribe system over a DHT network. In our system, subscriptions are distributed into an underlying DHT network. When an event is first published, the process that matches an event with subscriptions is done in a distributed fashion by only the nodes that store potentially matching subscriptions. Finally, this event is delivered to the matching subscribers by these nodes. To build this system over a heterogeneous DHT network, four key techniques are presented: (1) A content space mapping technique and a distributed 2d-tree over DHT. (2) A subscription installation mechanism, which distributes all the subscriptions over the DHT network through a random probing search algorithm. (3) An event publishing and delivery algorithm, which is able to locate all the potentially matching nodes efficiently and deliver events to them with a low overhead. (4) Fault tolerance handled by exploiting the underlying DHT. Therefore, our pub/sub system can simultaneously support multiple pub/sub schemas without the overhead of the maintenance of additional in-network data structures. Meanwhile, it enables the flexibility of schema change.

To evaluate the performance of our proposed architecture, we conducted extensive simulation experiments on a network model inferred from the Internet. The simulation results show that the proposed system can efficiently deliver an event to any users interested with low latency and bandwidth cost. We also evaluated that our application layer multicast algorithm and load balancing scheme, concluded that they can effectively reduce the bandwidth cost and relive some overloaded nodes. It is also shown that our system scales to large-scale networks.

5.2 Future Work

This thesis constitutes an initial step to building an efficient and scalable platform for supporting content-based publish/subscribe services in peer-to-peer networks. A number of issues need to be explored to further our work:

First, our simulation uses synthetic datasets due to the lack of publicly available publish/subscribe user data. A full understanding of a real publish/subscribe scenario can let us discover problems that our system might be faced with when applied to a practical application. Also, an extensive testing on PlanetLab[17] can help us analyze our system's performance in an Internet-like environment.

Second, currently our load balancing scheme is static by presetting a cut-off line for each attribute. This scheme might not be very effective without considering the runtime load distribution over nodes. As we discussed in section 3.4.2, an inherently hot search area can overload the nodes responsi-

ble for this region. But highly skewed data sets could be another overloading source. For example, if all the events are published on a very small range, it would be unavoidable that the target search areas of all of these events are largely overlapped. This could overload the nodes responsible for this area. Therefore, a more sophisticated and efficient load balancing algorithm which can balance nodes' load dynamically could be part of our future work.

Third, our system mainly relies on the underlying DHT to deal with node joins/departures/failures. Although this frees our system from actively maintaining a physical network's stability and availability, the performance of the proposed architecture under high node churn rate has not been explored. Moreover, some specific strategies could be investigated in the future to ensure at least the durability of subscription objects under such situation. On the other hand, a profile of real user activities on a p2p network can also be used to verify our system's performance under practical scenarios.

Another problem we will study is how cooperative peer nodes have to be in our system. For example, if we are disseminating stock data, there is inherent interest for an intermediate node to delay delivery until it can take advantage of the data first. Thus, potential applications for large-scale pub/sub have to consider this issue, i.e., to provide incentives for nodes to cooperate in event delivery.

Finally, enhancing our system's expressiveness is also a very interesting future research direction. Like most of the research on content-based pub/-sub service over p2p, our system focuses on supporting the model proposed by Fabret et al.[12]. However, this model is not expressive enough. For example, it can only support prefix or postfix matching for strings, failing to

support regular expressions, which can be a very valuable feature. Therefore, supporting more expressive pub/sub models is another direction we plan to pursue.

Bibliography

[1] Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, pages 179–194, 2001.

[2] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, New York, NY, USA, 1999. ACM.

[3] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. *icdcs*, 00:0262, 1999.

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] Fengyun Cao and Jaswinder Pal Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. In *Middleware*, pages 292–313, 2005.

[6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[7] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 163–174, New York, NY, USA, 2003. ACM.

[8] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, 2000.

[9] King Dataset. Available: http://pdos.csail.mit.edu/p2psim/kingdata.

[10] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[12] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30:115–126, 2001.

[13] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[14] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275. IEEE Computer Society Press, 1999.

[15] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[16] Napster. Website: http://www.napster.com.

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

[18] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[19] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems.

In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.

[20] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 30–43, London, UK, 2001. Springer-Verlag.

[21] Clay Shirky. What is p2p and what isnt. http://www.oreillynet.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html . OReilly, 2000.

[22] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[23] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, New York, NY, USA, 2003. ACM.

[24] Peter Triantafillou and Ioannis Aekaterinidis. Content-based publish-subscribe over structured p2p networks. In *Third International Work-*

shop on Distributed Event-Based Systems - DEBS '04, Edinburgh, United Kindom, May 2004.

[25] Peter Triantafillou and Andreas Economides. Subscription summarization: A new paradigm for efficient publish/subscribe systems. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 562–571, Washington, DC, USA, 2004. IEEE Computer Society.

[26] FreePastry website. http://freepastry.rice.edu.

[27] Xiaoyu Yang, Yingwu Zhu, and Yiming Hu. A large-scale and decentralized infrastructure for content-based publish/subscribe services. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 61, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Xiaoyu Yang, Yingwu Zhu, and Yiming Hu. Scalable content-based publish/subscribe services over structured peer-to-peer networks. In *PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 171–178, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA, 2001.

[30] Yinwu Zhu and Yiming Hu. Ferry: An architecture for content-based publish/subscribe services on p2p networks. In *ICPP '05: Proceedings*

of the 2005 International Conference on Parallel Processing, pages 427–434, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM.