2007

Techniques for building a scalable and reliable distributed content-based publish/subscribe system

Zhenhui Shen
Iowa State University

Techniques for building a scalable and reliable distributed content-based publish/subscribe system

by

Zhenhui Shen

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Srikanta Tirthapura, Major Professor
Arun Somani
Johnny S. Wong
Soma Chaudhuri
Manimaran Govindarasu

Iowa State University

Ames, Iowa

2007

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, my advisor Dr. Srikanta Tirthapura for his education, support and patience throughout my Ph.D. study and the writing of this thesis. Dr. Tirthapura taught me a lot on how to conduct research, write quality papers and present academic talks. Many research projects I completed were attributed to his insights and innovative ideas. I feel very fortunate to have Dr. Tirthapura as my major professor.

I would also like to thank my other committee members, Dr. Arun Somani, Dr. Johnny Wong, Dr. Soma Chaudhuri, and Dr. Manimaran Govindarasu for their efforts and contributions to this work. All the committee members provided me with valuable suggestions to improve my work during my preliminary and final oral exams. In addition, I want to thank Dr. Govindarasu to help me find a summer internship. I also want to thank Dr. Zhang Zhao for serving as my temporary committee member, due to the emergency leave of Dr. Chaudhuri.

I want to thank my colleagues, Bojian Xu, Bibudh Lahiri, Puviyarasan Pandian and Jason Stanek to make my study here a memorable one. I am grateful to their support during the rehearsals of my talk, as they suffered a lot from my unorganized speech.

I also want to thank my wife, Qi Xu, for her patience and words of encouragement during various stages of my Ph.D. study. I feel very lucky to choose her as my lifetime companion. Last, but not least, I would like to thank my parents Pengfei Liu and Aifen Shen for their endurance, great love and confidence in me.

CHAPTER 1. Introduction

Today's distributed systems running at Internet scale are heterogeneous, asynchronous, and loosely coupled. A variety of computing devices can be connected into the system, ranging from traditional workstations, mainframes to intelligent embedded devices, wireless PDAs and mobile phones. The system must support applications that continuously monitor or react to changes in the environment, inform other components asynchronously about transitions in their internal state, request services from and provide services to other components.

Publish/subscribe(pub/sub for short) stands for a type of communication paradigm. It allows clients to publish events(useful information), subscribe to a pattern of events and asynchronously get notified of interested events once they become available. Unlike traditional RPCs where communication is based on knowing the other party's address, clients in pub/sub can send and receive events without knowing each other's identity. As such, it is considered as a natural fit to the loosely coupled nature of modern distributed applications. Although pub/sub is not a new invention [55, 5], its use in large-scale wide-area communication has vastly increased its popularity in recent years.

Among all the variants of the pub/sub abstraction, content-based pub/sub is an especially powerful type of pub/sub service, which allows users to precisely specify what information they need. But the enhanced expressiveness in the selection mechanism slows down the speed of event matching. Meanwhile in order to accommodate a large number of information producers and consumers, the system's architecture is changing from a centralized "event broker" to a decentralized peer-to-peer network. The expanding network infrastructure introduces longer delay in event delivery, incurs significant message processing overhead and is prone to all kinds of faults. The goal of our research is to develop necessary data structures and algorithms to

improve the scalability and reliability of a distributed content-based pub/sub system.

1.1 Large-scale distributed applications

The Internet has dramatically changed our life and hosted many exciting applications at a worldwide scale: We see the rise of Web 2.0 as the next generation of World Wide Web. The term refers to a set of new technologies - social networks, blogs, podcasts, wikis and RSS feeds - all of which promote online collaboration and information sharing among users. Internet auction sites and marketplaces like eBay and Amazon continue to expand their online trading platform to create a virtual market where sellers meet buyers. Online games are conceived as a lucrative business and face the big challenge of updating the unique view of the shared virtual world for every player instantly. Traditional web-based content service already goes beyond the desktop to provide personalized content delivery to mobile users.

These applications strive to establish connections between numerous and anonymous information producers and consumers. The big challenge here is to efficiently disseminate useful information from its source to interested destinations. We first argue that publish/subscribe model provides a natural solution to this problem. Before justifying our argument, it is necessary to study some application scenarios to understand the requirements and to see why existing approach is limited in terms of the efficiency or usability.

1.1.1 Content distribution and access

How can we stay tuned with daily news, tech trends and blog posts? We may google the topics or bookmark the favorites. But at least we have to click on every link to visit the corresponding web page and search for the contents actively. This approach lacks efficiency due to a long list of urls to follow. In addition, web pages are not updated frequently. You are likely to waste the effort by tracking them everyday. But if you don't see it very often, you take the risk of missing valuable posts. On the other hand, information providers want to improve the visibility of their sites. A traditional approach is to use mailing list to keep track of interested readers. But a mailing list is monolithic and usually expensive to maintain.

1.1.2 Online games

Online games are predicted to be the future of the interactive entertainment industry. Usually several instances of an online game are running on multiple connected machines, and one of the players' machines acts as the server. All the players share a single instance of the virtual game world. But each individual only sees his surrounding area at any time instant. Although many entities are moving and changing status concurrently, it is only necessary for each player to get updated information about the entities that are visible to him based on his current location.

1.2 Why publish/subscribe?

The above applications demand a new communication style, in which the flow of information - from senders to receivers - is determined by the interests of the receiver rather than by an explicit destination address assigned by a sender. This task can be simplified if we are equipped with a mechanism which allows receivers to *subscribe* to the topics in advance and let senders *publish* content without knowing the receivers' identities.

For example, RSS [49] is a simple pub/sub solution to realize lightweight content distribution and access. RSS technology uses feeds, which are small XML files, to publish a summary of web content. Feeds are posted on the web and can be subscribed by interested readers. Programs known as feed readers can check a list of feeds on behalf of a user and display any updated articles that they find.

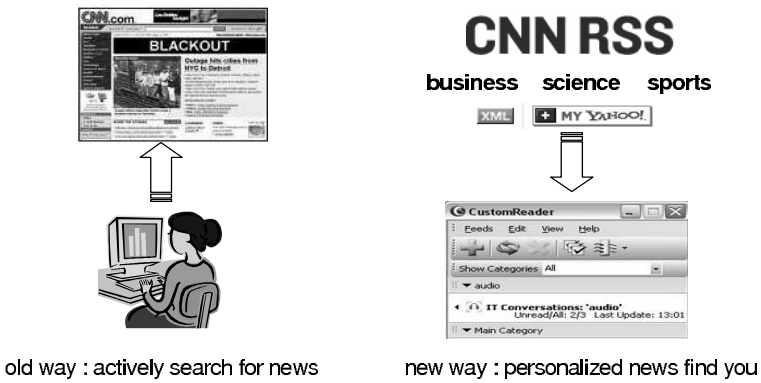


Figure 1.1: Personalized news delivery through RSS

Since RSS formats are specified in XML, the feeds can be automatically processed by software. This makes it possible to syndicate content without incurring further work from the original publisher. Companies no longer need a mailing list to keep track of interested readers. On the other hand, a user can depend on a feed reader to automate the retrieval of latest information. Some readers poll a feed server periodically to check updates. Other readers can register a callback function with a feed server. Thereafter the server notifies the reader only when it has new items.

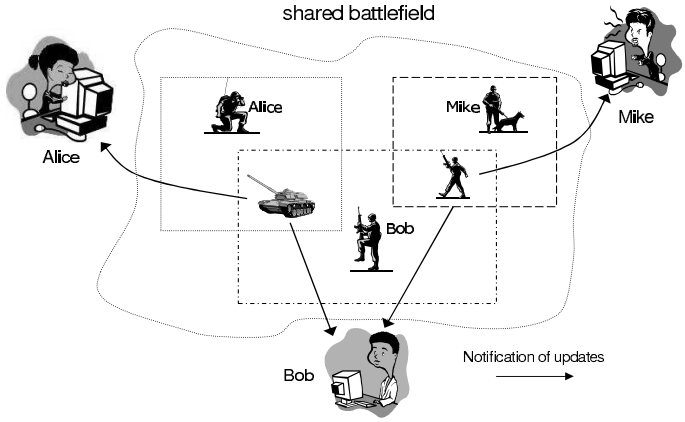


Figure 1.2: Real-time updating of game scenes

Message communication in online games can also be modeled as a pub/sub service [3]. We can draw a rectangle to encompass a player's territory. Each player then expresses his interest in the game environment by subscribing to the rectangle which specifies his arena. The location of each moving entity can be described by a set of coordinates. The position information is published and sent to all the players whose subscribed arena contains the advertised coordinates. In this way, every player is guaranteed to receive only the information which is necessary to update his view of the game world.

1.3 Research challenges for publish/subscribe

In this section, we briefly introduce the main features of the pub/sub paradigm, and then highlight some research challenges in this area.

There are two interacting parties in a pub/sub system. One is the *publisher* who generates

events. The other is the *subscriber*, who consumes events and specifies his interest in receiving certain type of events by registering *subscriptions*. A subscription is a content filter used to screen events. A pub/sub system is an event-based middleware, which corresponds to the "cloud" in Figure 1.3. This middleware bridges publishers to subscribers by delivering events from a publisher to interested subscribers based on their contents. It exposes "subscribe" and "publish" interfaces to external parties. The form of the middleware can be as simple as a single computer or as complicated as a distributed network of many computers.

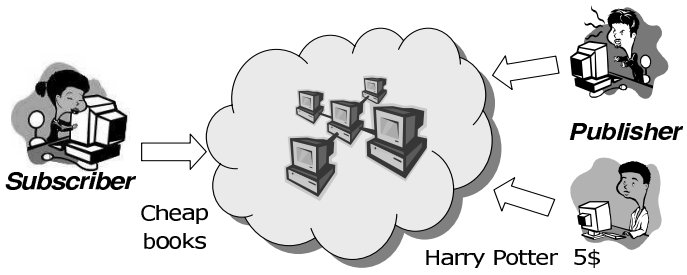


Figure 1.3: The abstraction of a pub/sub service network

Because a centralized server does not scale with the increasing number of clients and suffers from a single point of failure, we are interested in a distributed pub/sub system. A distributed pub/sub network consists of many interconnected pub/sub routers. Every client joins the system by choosing a nearby router as the access point. The client interacts with the gateway router to submit his subscriptions, advertise new events and receive interested events.

Among all the variants of the pub/sub abstraction, content-based pub/sub is the most powerful type of service, as it provides fine-grained event filtering capability. But distributed content-based pub/sub is generally harder to implement and existing implementations face the following challenges.

1.3.1 Event forwarding

Event forwarding is the primary purpose of a pub/sub system. In a distributed network, event forwarding has both local and global aspects. Locally an event is matched against a set of registered subscriptions based on its content, which we call a "content-match". Content-match is a hard combinatorial problem, as the matching now is based on an event's content, rather

than on a fixed destination address. Moreover the amount of registered subscriptions can be huge. Kale *et.al.* [28] proved that the complexity of content-match is within a constant factor of that of the Partial Match, which is a notoriously difficult pattern match problem. Therefore it is accepted that content-match should be done as few times as possible.

Globally an event needs to traverse many routers before reaching interested subscribers. This is because we no longer rely on a single router to find out all the matching subscriptions. During the traversal of an event, similar content-match will be performed at multiple pub/sub routers. This will further aggravate the situation. As users are sensitive to the end-to-end delay reflected in event forwarding, our first challenge is to conquer the complexity inherent in content-based event forwarding to shorten the delivery time.

1.3.2 Subscription propagation

In a distributed network, new events can arrive at different routers. It is necessary for a pub/sub router to register its subscriptions at other sites so that interested events can be sent back. Currently broadcasting is the common approach used to propagate subscriptions inside a pub/sub network. But the cost of replicating all subscriptions at all pub/sub routers grows super linearly with the total amount of subscriptions in the system. This also leads to a larger routing table at each site, which in turn slows down the event matching.

To alleviate this, multiple groups [58, 39, 50, 30, 57, 61, 10] suggested exploiting the covering relations among subscriptions to optimize subscription propagation. Covering represents the containment relationship among subscriptions. We say one subscription s_1 covers another subscription s_2, if and only if the set of events selected by s_1 subsumes the set of events selected by s_2.

We can take advantage of covering relations to optimize subscription propagation as follows: Let I represent an interface at a pub-sub router. The router received a subscription s_2 and decided to forward it through I. We can check if there exists some subscription s_1, which covers s_2 and was forwarded through I. If s_1 exists, then s_2 is not forwarded since all events matching s_2 will be received due to the prior propagation of s_1. Repeating this process at

every router can result in a significant reduction in the number of forwarded subscriptions. The advantages are twofold: (1) The number of subscribe and unsubscribe control messages is reduced, and (2) The sizes of routing tables decrease, leading to faster notification forwarding.

But the introduction of covering yields new problems. Firstly, for every new subscription, we need to examine whether it is covered or not. This is a hard problem, as its special case can be formulated as multidimensional point dominance problem, for which no worst-case efficient solution is known. Secondly in the presence of covering, if a general subscription s is deleted from one router, some specific subscriptions may no longer be covered due to the removal of s. Such subscriptions should be identified and forwarded to other routers. Otherwise clients may not receive interested notifications in the future. This calls for an explicit tracking of detected covering relation. Since subscriptions are frequently added/deleted from the system, the exhibited dynamism poses a challenge to the efficient management of subscription covering.

1.3.3 Fault tolerance

The third challenge is related to the system's reliability. Like other large-scale distributed systems, faults do occur in a pub/sub network and can take various forms. An arbitrary fault can destroy the network's connectivity, corrupt the routing configuration or disable a pub/sub router. From a user's perspective, faults will let them lose interested events or receive unwanted events. This will void the service contract promised by the pub/sub model. However due to the huge diversity of possible faults, it is expensive (if not impossible) to identify and tackle each fault separately. Moreover transient local faults may exert negative effect over a large portion of the system. It often needs repairing at many pub/sub routers to fully recover from a fault. Thus a good correction scheme should be scalable and minimize the processing overhead. Our challenge is to devise a lightweight fault detection and recovery mechanism that can restore a pub/sub system from various transient faults.

1.4 Contributions of the thesis

This thesis presents the results of our study on developing necessary data structures and algorithms to improve the scalability and reliability of a distributed content-based pub/sub system. Parts of the thesis were previously published as [53, 52, 50, 51]. The contributions we made respond to the challenges mentioned in Section 1.3 and can be summarized as follows.

1.4.1 Faster event forwarding through lookup reuse

As pointed out earlier, event forwarding in a pub/sub system is slow, because content-match is complicated and often repeated at many pub/sub routers. We propose a novel technique called "lookup reuse" to effectively replace an expensive content-match with a much cheaper hash-table lookup. Lookup reuse reduces or sometimes completely eliminates the need of content-match at many pub/sub routers. As a consequence, events can pass through a router much faster. We compare several reuse strategies and show that Partial-Reuse yields the best performance.

Many researches in pub/sub have been devoted to the study of faster event forwarding. However most of them focus on building efficient indexes to accelerate event matching at a single router. In contrast, our lookup reuse strategy addresses the inefficiency of event forwarding arising at the network level. It complements existing studies and touches the distributed nature of the event forwarding process.

1.4.2 Efficient subscription management for covering

The introduction of covering brings new problems, mainly fast detection and maintenance of covering relations. Existing solution integrates the two functions into a single monolithic data structure. We take a different approach by decoupling the detection of covering from its maintenance. This modular design allows us to analyze the unique needs of each task and work out the best solution for each piece. To be specific, we build a subscription index to quickly answer covering query and construct a simple relation graph to dynamically maintain discovered covering relations. We are also the first group to investigate the interaction between

the two modules in order to combine them into a coherent general purpose two-layer framework. Experimental results show that this framework runs much faster than existing solutions.

For the above two tasks, covering detection is much more expensive than the maintenance. Its special case, the detection of covering for numeric subscriptions, can be formulated as a multidimensional point dominance problem, for which no efficient worst-case solution exists. However we realize that covering is just an optimization, which needs not to be strictly followed all the time. This motivates us to propose a new concept called "approximate covering" to obtain most benefits of exact covering at a fraction of its cost. A practical solution based on Z space filling curve is presented. Our contributions are two-fold: Firstly, there is little work on approximate covering detection. The only known result [39] is a probabilistic algorithm whose complexity is in linear with the number of subscriptions, while we give a sublinear solution. Secondly, relevant researches are limited to empirical studies. We present a formal analysis of the algorithm performance.

1.4.3 Self-stabilization for fault tolerance

Self-stabilization is a notion first developed by Dijkstra [17] in 1974, which indicates a system's self-healing ability. The advantage of self-stabilization is that it addresses all faults through an uniform mechanism, rather than enumerate all possible faults and propose separate corrections for each of them. As such, we use it as a tool to fortify the pub/sub infrastructure, which is prone to many kinds of faults.

We design a self-stabilizing algorithm to maintain the consistency among distributed routing tables in a pub/sub system. The algorithm achieves consistency by letting neighboring routers periodically exchange their routing information and correct faults when necessary.

We further propose several optimizations to the algorithm. To be specific, we have the routers exchange the sketches of their routing information, which are much smaller in size. An analysis of the associated space/accuracy tradeoff is presented. For a special case which involves a transient edge failure, we devise an online algorithm to optimize the fault recovery such that the cost is within the twice of the optimal.

1.5 Dissertation outline

The rest of the thesis is structured as follows.

In Chapter 2, we give an overview of the pub/sub system and show a formal modeling of distributed content-based pub/sub. We also survey some popular system prototypes.

In Chapter 3, we discuss the lookup reuse strategy. The benefit of lookup reuse is demonstrated through simulation study. The impact of subscription covering to various reuse strategies is also investigated.

Chapter 4 is about the efficient management of subscription covering. First the notion of covering and its benefit are explained. Next we describe our two-layer management framework and compare it against existing approaches. Approximate covering and its practical solution are presented at the end.

In Chapter 5, we describe the self-stabilizing algorithm used to maintain the consistency among distributed routing tables in a pub/sub system. Two optimizations of the algorithm are presented. Their effectiveness are validated through formal analysis as well as simulation study.

Chapter 6 completes the dissertation with concluding remarks.

CHAPTER 2. Background

The purpose of this chapter is to provide necessary background in understanding the concepts that are related to a publish/subscribe system. We classify current implementations into subject-based and content-based systems, with an emphasis on the latter. We present the general architecture of a distributed content-based pub/sub system, in which we illustrate two design issues: How expressive can subscriptions be? And how are events routed in a distributed system? We conclude this chapter by reviewing some current systems.

2.1 Classification of publish/subscribe systems

Many existing systems implement different variants of the pub/sub abstraction. A major distinction among them is the granularity of event selection. Two main models have resulted from this: subject-based and content-based. We will use this distinction to classify current systems in the remainder of this section.

2.1.1 Subject-based system

In a subject-based system, the event space is partitioned into disjoint zones called "subjects". Clients can subscribe to one or more subjects and they will receive all events which belong to these subjects.

ROSS is an example of subject-based pub/sub system. Using ROSS, CNN.com divides its content space into a set of news channels, including "world", "U.S.", "business", "science" etc. Readers who are interested in events happening outside the U.S. can subscribe to the "world" feed. Thereafter they will get notified of latest events that are classified as belonging to "world" by CNN.

The implementation of subject-based pub/sub is simple and efficient, since we can take advantage of group communication mechanisms such as IP multicast. Predefined subjects can be considered as multicast groups. Event forwarding reduces to a lookup on the event's subject in the routing table to find the associated multicast address, followed by a regular multicast.

However subject-based system is limited in terms of its expressiveness. Users often have a more precise specification of their interests than a general subject name. For example, readers may want to know the events happening outside the U.S. within a certain period, for which the "world" subject by itself is not selective. Users need to further filter the received events based on the time they occurred. Moreover a single event often fits more than one subject. For example, the discovery of a new protein by an European professor can be published to both "world" and "science" feeds. Any user who subscribes to both feeds is going to read the same message twice. The above constraints have limited the wide deployment of a subject-based system. At present, many commercial pub/sub systems are subject-based, such as Java JMS [27], TIBCO Rendezvous [59] and IBM WebSphere MQ [26].

2.1.2 Content-based system

A content-based interface allows a user to express his interest in events in a much more flexible and precise way. The unprecedented expressiveness is based on a well-defined data model formalized in [34]. The model uses an *event schema* to define the type of information contained in each event. The event schema advertising a digital camera could be a pair containing two *attributes*: zoom factor and resolution, both are floating point numbers. For example, an event can be (zoom = 10.5×, resolution = 5.0mp). A subscription is a conjunction of a set of *predicates*. Each predicate is a constraint on the value of an event attribute and consists of an attribute name, an operator, and a value. In the digital camera example, a valid subscription might be (zoom $\geq$ 6, resolution $>$ 4). In this example, we say the event *matches* the subscription because the value of each attribute in an event is accepted by the corresponding predicate in the subscription.

If content-based systems are to scale to large networks, event forwarding must be performed

in a distributed fashion. Existing systems, such as Siena [54], Gryphon [24], Rebeca [33], Hermes [42] and Jedi [16], have demonstrated how to construct a distributed network of routers to accomplish this task. The typical setup is illustrated in Figure 2.1. Each router manages a set of local clients and is connected to a set of peer routers. A client expresses his interest in receiving certain types of events by registering a set of subscriptions at a nearby router. Every router forwards its registered subscriptions to its neighboring routers. These subscriptions are further forwarded in the network to create a reverse path for matching events to flow back to the subscriber. Whenever an event is published at a router, the router performs a content-match and delivers it to all local clients who have issued matching subscriptions, and also forwards it to neighboring routers from whom matching subscriptions were received. This process is repeated at each subsequent downstream router until the event reaches all interested subscribers.

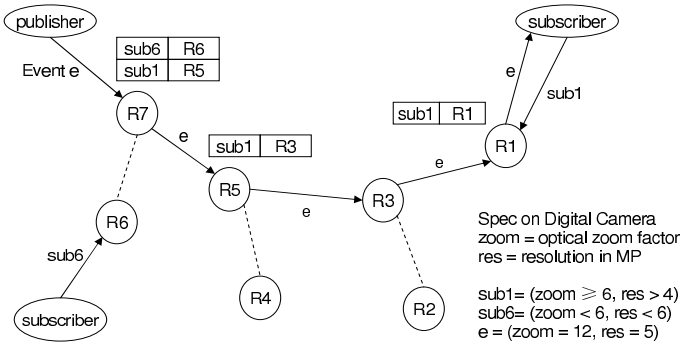


Figure 2.1: Architecture of a distributed content-based pub/sub system

In comparison with subject-based system, content-based system delivers to the users only the events they want. There is no additional filtering to be performed by the user like in ROSS. However content-based pub/sub is generally harder to implement and existing systems run much slower than subject-based pub/sub.

2.2 Examples of publish/subscribe systems

In this section, we describe some real implementations of content-based pub/sub system. Note that the set of systems cited is not meant to be exhaustive. They are used as illustrations

to exemplify the issues involved and their impact.

2.2.1 Gryphon

Gryphon is a content-based pub/sub system developed at IBM Watson research center [24]. The system is constructed as a redundant overlay network consisting of routers distributed across multiple geographic locations. Clients can use the Java Message Service API to access the service. Gryphon has been deployed over the Internet to support real-time applications such as score distribution for Wimbledon 2001 and statistics reporting at the Sydney Olympics.

Some important research contributions of Gryphon include an event matching algorithm with sub-linear complexity [65]; a scalable protocol to achieve exactly-once delivery of information to a large number of subscribers in either publisher order or uniform total order [4]; a subscription propagation algorithm which supports in-order gapless delivery [66] and a congestion control mechanism to protect a pub/sub system against network failure and link congestion [43].

2.2.2 Siena

Siena is a research project developed at University of Colorado [54]. Siena aims to provide efficient and scalable event routing over a wide-area network. They formalized the content-based data model [10] used to define the structure of events and subscriptions. They also introduce the notion of subscription covering to optimize the propagation of subscriptions inside a pub/sub system.

Siena uses a counting algorithm [12] to realize event matching at a pub/sub router. The counting algorithm assigns a counter to each subscription to keep track of the number of predicates satisfied by an event. After iterating over all the attributes in an event, the algorithm returns subscriptions whose counter value equals its number of predicates.

To forward an event, Siena router uses a two-layer scheme [11] which is a combination of a traditional broadcast protocol and a content-based routing protocol. The broadcast protocol treats each event as a broadcast message, and the content-based routing protocol prunes the

branches in the broadcast tree to direct an event only to the nodes that have matching subscriptions. This modular approach allows a pub/sub infrastructure to reuse existing broadcast methods to realize one-to-many communication over the physical network, while the system design can focus on building logical paths from possible publishers to interested subscribers.

2.2.3 Rebeca

The Rebeca project [33] at Darmstadt University of Technology investigates the usage of pub/sub technology for large-scale e-commerce applications. The system was used as a testbed to evaluate several content-based event routing algorithms [35]. The study indicated that the usage of advertisement and clustering of similar user interests can benefit event routing.

They also explored building a pub/sub system on top of the Chord [56] peer-to-peer network to take advantage of the extra features offered by a peer-to-peer substrate, such as bounded routing depth, load balance, self-organization and congestion control. Similar to our work, Rebeca uses self-stabilization [36] to automate fault recovery. The main idea is that routing entries should be leased. Any entry which is not renewed before the expiration of its leasing period will be purged from the system. A potential drawback of this approach is that the amount of renewing messages issued by a client could impose a significant processing overhead to the system.

2.2.4 Hermes

Hermes is an event-based middleware developed at University of Cambridge [42]. Hermes distinguishes itself from simple pub/sub system by augmenting additional functions commonly found in traditional middleware, such as programming language integration, access control and composite pattern detection.

Hermes introduces a type-based data model to define events and subscriptions. Each event is an instance of an abstract event type which defines its content schema. Types can be inherited to create more sophisticated subtypes. This is analogous to the model of object-oriented programming. A subscription must choose an event type, and then define constraints

to further filter events of the specified type.

The system selects a rendezvous node for each type. Paths are created to route events from publishers to the rendezvous node. From there, a dissemination tree is built to include each subscriber who is interested in this type as leaf nodes. Events are first broadcasted inside the diffusion tree, and content-based filtering is performed directly by subscribers. To help publisher/subscribe create a path leading to the rendezvous node, Hermes uses a distributed hash table based on the Pastry [13] substrate to set up the necessary state.

CHAPTER 3. Faster event forwarding through lookup reuse

When compared with a subject-based system, a content-based publish-subscribe system enables a client to express his interest in the events more precisely. This flexibility simplifies the design of distributed applications and is greatly desired. For instance, IBM has designed a real time scoreboard, powered by a content-based pub-sub middleware called Gryphon [24], for the Australian Open.

However, the flexibility provided by a content-based system currently comes at a significant cost. Event forwarding is an expensive task, since it is now based on the content of an event, rather than on a fixed destination address (as in IP routing), or a fixed subject name, as in subject-based pub-sub. When an event arrives at a router, its content must be matched with the set of all registered subscriptions to find interested subscriptions, an operation that we call a *content-match*. The set of subscriptions maybe large (tens of thousands), and there is no simple way known to index these subscriptions to guarantee fast matching.

A large body of research (see Section 3.2) has been devoted to indexing subscriptions to perform content-match efficiently. However, it can be shown [50] that high-dimensional range-searching (for which no worst-case computationally efficient solution exists) is a special case of the content-match problem, implying that content-match also does not have a worst-case efficient solution. Thus, it is accepted that content-match is an expensive task, best done as few times as possible.

Furthermore many existing publish-subscribe systems perform content-based routing in a distributed manner, such as Siena [54], Gryphon [24], Rebeca [33], Hermes [42], Jedi [16] and XNet [45]. Whenever an event is published at a router, the router forwards the event to its neighbors from whom matching subscriptions were received. This process is repeated at

each subsequent router, lying on the path established by a prior propagation of the matching subscription, to reach interested subscribers. Consequently the same event-subscription match gets repeated at many routers, which further exacerbates the efficiency of event forwarding.

In this chapter, we introduce *lookup reuse*, a novel approach to improve the efficiency of event forwarding. Lookup reuse enables faster event forwarding through reusing matching results computed by upstream routers in making forwarding decisions at downstream routers. In many cases, this lets downstream routers replace an expensive content-match operation with a much cheaper hash-table lookup. We study the integration of lookup reuse into existing content-based event forwarding algorithms. In particular, we propose two reuse strategies: *Always-Reuse* and *Partial-Reuse*. As a comparison, existing event forwarding algorithms are classified as *Never-Reuse*. Our simulations show that when combined with popular content-based event matching algorithms, Partial-Reuse is the best strategy to use.

Although lookup reuse is considered as a lightweight modification to existing event forwarding algorithms, it might create conflicts with other optimizations to a pub/sub system. For example, subscription covering is considered as an optimization to realize efficient propagation of subscriptions in a pub/sub system. Yet its presence will invalidate the Always-Reuse strategy and affects the performance of the Partial-Reuse strategy. We will analyze and measure the impact of subscription covering to lookup reuse strategies at the end of this chapter.

3.1 Introduction

In current pub-sub systems, event forwarding is performed independently from router to router. When an event is forwarded from one router to the next hop after a content-match, the next hop router starts from scratch and performs another content-match on the event using its own routing table. Our faster event forwarding strategy is based on the following observation. In many cases, the efforts spent by the routers in performing a content-match are repetitive. Information needed by downstream routers to forward an event has been partially or wholly computed by the upstream routers during their content-match procedure.

The above redundancy results from the way in which subscriptions are propagated in the

system. A subscription that originates at a router will be forwarded to neighboring routers so that matching events can flow back. Therefore, routing tables at neighboring routers share many common subscriptions, and an event-subscription match which occurs in one router will also occur in a neighboring router with high likelihood.

We propose a novel technique called *lookup reuse* to reduce this redundancy and improve forwarding efficiency. The general approach is as follows. Once a router performs a content-match and forwards an event to a next hop router, it passes along some metadata about the content-match. Informally, this metadata tells the reason why the event is being forwarded in this direction. This metadata can be used effectively by the next hop router to speed up its own event forwarding procedure.

A simple way to implement lookup reuse, called *Always-Reuse*, is as follows. The metadata is a list of all the subscriptions that match the event at the upstream router. The upstream router passes on this list while forwarding an event. Instead of sending the actual subscriptions themselves, which can be quite bulky, the upstream router can hash each subscription into a unique key and then send the key instead. At the downstream router, a content-match can be replaced by (many) inexpensive hash-table lookups on the set of keys received from the upstream router. Since a content-match is much more expensive than a hash-table lookup, this may speed up event processing at the downstream router.

To get an idea of the potential gain from replacing a content-match with a hash-table lookup, we conducted experiments to compare the cost of the following operations: (1)*key lookup*: each subscription is assigned a unique key and the set of all keys K is indexed using a hash table. Given a query key k, does $k \in K$? With a good hash table implementation, this cost is independent of the table size. (2)*content-match*: given an event e and a set of subscriptions S, does there exist a subscription $s \in S$ such that e matches s? This cost depends on the size of S, and also on the content-match algorithm being used.

We study the cost of two representative content-match algorithms: the K-d tree and the Counting algorithm (their descriptions are available in Section 3.4). Table 3.1 shows the experimental results. Our experiments show that the cost of a content-match is about 2-3

orders of magnitude greater than the cost of a key lookup for subscription sets of the same size.

Input	Counting	K-d tree	Key Lookup
1	720	144	0.73
2	391	77	0.73
3	177	42	0.73

Table 3.1: The average time taken for a content-match using two different algorithms. All times are in microseconds. The data is for 10,000 subscriptions. For inputs 1,2 and 3, the average fraction of subscriptions matching an event was 1%, 5% and 15% respectively.

A simple approach like Always-Reuse may not work well in many circumstances. For example, consider a case when a given event matches all subscriptions at a router. Finding all matching subscriptions and forwarding their keys along with the event may not be very efficient for the following reasons. First, finding *all* matching subscriptions imposes a heavy burden on a single router. It is also an "overkill", since the presence of a single matching subscription is enough to forward an event along. Second, the number of keys forwarded is huge. This may induce a significant message overhead and slow down the packet transmission over the network link. Third, the overhead of looking up all these keys at the downstream router could nullify any advantage gained by the elimination of a content-match. A better strategy would be a hybrid approach that switches dynamically between reusing lookups and performing a content-match, depending on the matching results passed by preceding routers. The hybrid approach is better able to balance the overhead of lookup reuse with the cost of content-match. The hybrid strategy we propose in this work is called *Partial-Reuse*.

In addition to the above factors, the validity of *Always-Reuse* is based on the assumption that the event source is able to find out all the matching subscriptions registered inside the network for a published event. This condition can be satisfied only if a subscription is replicated at every router. Recently researchers proposed to exploit the covering relationship among subscriptions to realize efficient subscription propagation. Under this advanced scheme, not every subscription is broadcasted over the network. Therefore *Always-Reuse* will become invalid.

On the other hand, we are going to show that *Partial-Reuse* remains correct and explore the impact of subscription covering to the performance of *Partial-Reuse*.

In summary, the contributions of this paper are as follows:

(1) We introduce *lookup reuse* as a lightweight optimization to the most critical task of event forwarding in distributed content-based pub-sub systems. It replaces a large fraction of potentially expensive content-match operations with much cheaper hash-table lookups, by strengthening collaboration among pub-sub routers. Lookup reuse is simple, and can be profitably used in conjunction with *any* content-matching scheme, as long as the cost of a content-match is significantly greater than a hash-table lookup.

(2) We explore the design space of various lookup reuse strategies. The simple Always-Reuse just relies on key lookups (after an initial content-match), while a hybrid Partial-Reuse dynamically switches between lookup reuse and content-match. In addition, we also include a strategy called *Never-Reuse*, which is an alias of the existing content-based event forwarding algorithm that does not reuse any key lookups.

(3) We conducted simulations under different scenarios to measure the performance of proposed lookup reuse strategies by integrating key lookup with two representative content-match algorithms: the Counting algorithm and the K-d tree algorithm. Our simulations showed that *Partial-Reuse* outperforms *Always-Reuse* and *Never-Reuse*, suggesting *Partial-Reuse* is the best strategy to reuse key lookups.

(4) We investigated the performance of *Partial-Reuse* in the presence of subscription covering. Though the introduction of covering increased the overhead of *Partial-Reuse*, the results showed that it is still superior to *Never-Reuse* on most inputs.

3.2 Related work on event forwarding

Content-match lies at the heart of event forwarding in a distributed pub-sub system. Recently Kale *et al.* [28] formally proved the hardness of content-match by showing its equivalence to the notoriously difficult Partial Match problem. For the problem of content-match, various forms of decision trees and subscription indexes are proposed. These efforts can be largely clas-

sified into two broad categories: the counting-based approach [12, 19, 40] and the tree-based approach [28, 7, 1].

Kulik [29] proposed a fast event forwarding algorithm, called match-structure event forwarding. In his approach, every event or subscription is prefixed with a match-structure header, which helps to speed up event-subscription matching at downstream routers. In comparison with lookup reuse, Kulik's scheme differs in the way how the header is generated and interpreted, and is somewhat similar to our *Always-Reuse* strategy.

Content-based publish-subscribe forwards an event based on its content. Existing group communication protocol like IP multicast, though very efficient, is not readily applicable to this new communication pattern. Cao *et.al.* proposed a new design called MEDYM(Match-Early with DYnamic Multicast) in [9]. MEDYM decouples the event forwarding into two functionalities: slow content-based event matching at network edge, and fast address-based event multicasting inside the network.

The above two approaches share similarity with our *Always-Reuse* strategy. For a single event, expensive content-match is executed at the entry point. Once an event entered the network, content-match is eliminated and replaced by other faster forwarding method. Meanwhile they also face the problems intrinsic to *Always-Reuse* as we exposed in Section 3.1.

It is suggested that partitions can be used to expedite event forwarding by confining the propagation of information to smaller scopes [62, 8]. There are two dual approaches to achieve this: event partitioning and subscription partitioning. Partitions enhance the relevance of subscriptions to events by relocating them to the same subnet. As a result, events traverse fewer hops and the size of routing tables is also smaller, since each router only needs to maintain a subset of subscriptions, pertaining to events that may be routed on the networks in which it participates.

The efficiency of event forwarding also depends on the subscription management. Many systems [50, 39, 30, 57, 61] exploit "covering" relationships among subscriptions to reduce the number of forwarded subscriptions and keep the size of routing tables small. The introduction of subscription covering influences the implementation of lookup reuse. Such an impact is

investigated in Section 3.6.

3.3 Lookup reuse strategies

In this section, we describe various event forwarding strategies, starting from the current forwarding algorithm, which does not reuse any lookups, and moving on to the Always-Reuse strategy, and then the Partial-Reuse strategy. Lookup Reuse algorithms tie in with the architecture of a distributed publish-subscribe system. Before describing the algorithms, we first define our system model as follows.

Interconnection topology In a pub-sub system, we want to preserve the following invariant: a router receives no more than one copy of a given event or subscription. Thus, each event or subscription must be propagated over a tree. To ease our algorithm description, we assume a simple network topology, where all pub-sub routers are organized into a single spanning tree. Each router is connected to adjacent routers on the tree through network interfaces. This is not a robust infrastructure from the perspective of fault-tolerance and network congestion. In many cases a more redundant topology with cycles is used to connect the pub-sub routers.

But such a simplification will not alter the design of Lookup Reuse algorithms. In case the topology is not a tree, different sources are going to use different trees to propagate their original subscriptions and events. The only change arising is the set of interfaces a downstream router has to examine upon receiving an incoming event. We can formulate this set as $Forwards(e) = Neighbors - NST(e)$. $Neighbors$ is the complete set of incident network links. NST (Not on any Spanning Tree) comprises all the links which are not on the computed spanning tree rooted at the original publisher of e. With acyclic topology, NST only includes the link from which e was received. For a redundant topology, NST may include more incident links. So with a redefinition of $Forwards(e)$, one can immediately apply the same Lookup Reuse algorithm to a redundant network topology.

Subscription propagation In order to retrieve useful events published at a remote site, we need to register local subscriptions at other routers as well. Subscription propagation is thus an indispensable procedure of any distributed publish-subscribe system. In this section, we consider a simple propagation scheme where every subscription is broadcasted over the entire network. Researchers also propose to take advantage of the covering relationship among subscriptions to realize efficient subscription propagation. We will investigate the impact of covering to lookup reuse strategies in Section 3.6.

A router assigns incoming subscriptions to the interfaces from which they are received. It often indexes subscriptions coming through the same interface to expedite content-based event matching. However the exact form of a subscription index is not important to our study. Lookup reuse should generate a profit as long as a content-match based on a subscription index runs significantly slower than a hash-table lookup.

3.3.1 Never-Reuse

Now we review the current event forwarding algorithm. When an event arrives at a router from an interface, the router needs to find out all the local subscribers who are interested in this event. Besides for every other incident interface, the router checks to see if there is a subscription received from that interface which matches the event. If a matching subscription exists, then the event is forwarded, otherwise it is not forwarded. The same process is repeated at all routers on the path(s) from the event source to the recipients of the event.

The pseudo code of *Never-Reuse* is presented in Algorithm 3.1. We refer "event source" to the router at which the event is first published. The routing step is very similar at any other router. The only difference is that a downstream router should exclude the interface from which the event arrived when matching the event. This is necessary to break the loops in event delivery. Since the current approach does not involve any lookup reuse, we call it the *Never-Reuse* strategy.

Algorithm 3.1 Never-Reuse (*event* e) /* event source */

```
1: for each interface I do
2:   if I is the local interface then
3:     use content-match to find all matching subscriptions
4:     deliver e to interested local subscribers
5:   else
6:     use content-match to find one matching subscription that arrived through I
7:     if a matching subscription exists then
8:       forward e through I
```

3.3.2 Always-Reuse

To implement lookup reuse, for each subscription entering the network, a hash function is applied on it to obtain an integer value. This integer is called a subscription's *key*. If two subscribers submit the same subscription, we can break the symmetry by prefixing each subscription with the subscriber's unique id before hashing it. We assume that the hash function will generate different keys for different subscriptions, and this can be easily achieved (with overwhelming probability) using any standard hash function, such as the simple modulo function, as long as the key space is large enough (32 bits will do). Thus, we assume that there is a one-one mapping between subscriptions and keys.

We use a structure called the *KeyTable* at every router. KeyTable is a hash table mapping a subscription's key to the interface from which that subscription arrived. Given the key of a subscription, KeyTable returns the associated interface by using a single hash table lookup.

In Always-Reuse, the event source is responsible for finding *all* the matching subscriptions for an event. The strategy eliminates content-match at any downstream outer. Alternatively event forwarding is achieved by performing hash table lookups on the set of matching subscription keys that are piggybacked over the event. The pseudo code of Always-Reuse is presented in Algorithm 3.2 and 3.3.

Always-Reuse can be useful when an event matches only a few subscriptions. Under this circumstance, the load of finding all matching subscriptions at the event source is moderate. Furthermore the event forwarding inside the network is realized using pure key lookups. The benefit is more pronounced if an event traverses many pub-sub routers. However it performs

Algorithm 3.2 Always-Reuse (*event* e) /* event source */

```
1: for each interface I do
2:   if I is the local interface then
3:     use content-match to find all matching subscriptions.
4:     deliver e to interested local subscribers
5:   else
6:     use content-match to find all matching subscriptions
       and store their keys in K_∞.
7:     if K_∞ ≠ φ then
8:       forward < e, K_∞ > through I
```

Algorithm 3.3 Always-Reuse (*event* e, *set<key>* K) /* downstream routers */

```
1: let K_I represent the set of subscription keys associated with interface I
2: for each interface I except the one from which e is received do
3:   if (K ∩ K_I) ≠ φ then
4:     if I is the local interface then
5:       deliver e to interested local subscribers by looking up K ∩ K_I.
6:     else
7:       forward < e, K ∩ K_I > through I
```

poorly under circumstances described in Section 3.1.

3.3.3 Partial-Reuse

In comparison with Always-Reuse, Partial-Reuse makes a few changes at both the event source and at the other routers. Its pseudo code is presented in Algorithm 3.4 and 3.5.

At the event source, it is no longer necessary to find *all* matching subscriptions before forwarding an event. Alternatively we apply a threshold of n, which specifies an upper bound on the number of matching subscriptions to search. Suppose an event matches m subscriptions on interface I. If $m \leq n$, the content-match returns m keys. If $m > n$, the content-match just returns n keys. Therefore the number of keys is bounded by n under Partial-Reuse.

We say the key list is *complete* if it contains the keys of all matching subscriptions. At a downstream router, if the received key list is complete, Partial-Reuse forwards an event only based on key lookups. Otherwise it first uses key lookups to eliminate content-match on as many interfaces as possible. Content-match is then performed only on interfaces which are not covered by any key.

Partial-Reuse achieves good load balance, as we no longer rely on the event source to find all the matching subscriptions. By imposing a threshold on the number of matching subscriptions to search, it is able to control the overhead of content-match as well as key lookups at every router.

Algorithm 3.4 Partial-Reuse (*event* e) /* event source */

```
1:  for each interface I do
2:    if I is the local interface then
3:      use content-match to find all matching subscriptions.
4:      deliver e to interested local subscribers
5:    else
6:      use content-match to find at most n matching subscriptions
        and store their keys in K_{≤n}.
7:      if K_{≤n} ≠ φ then
8:        forward < e, K_{≤n} > through I
```

Algorithm 3.5 Partial-Reuse (*event* e, *set<key>* K) /* downstream routers */

```
1:  let K_I represent the set of subscription keys associated with interface I
2:  for each interface I except the one from which e is received do
3:    if K is complete then
4:      if (K ∩ K_I) ≠ φ then
5:        if I is the local interface then
6:          deliver e to interested local subscribers by looking up K ∩ K_I
7:        else
8:          forward < e, K ∩ K_I > through I
9:    else
10:     if I is the local interface then
11:       use content-match to find all matching subscriptions
12:       deliver e to interested local subscribers
13:     else
14:       if (K ∩ K_I) ≠ φ then
15:         forward < e, K ∩ K_I > through I
16:       else
17:         use content-match to find at most n matching subscriptions
            and store their keys in K_{≤n}
18:         if K_{≤n} ≠ φ then
19:           forward < e, K_{≤n} > through I
```

3.4 Simulation methodology

We simulated the following event forwarding strategies: Never-Reuse, Always-Reuse and Partial-Reuse using the `OMNeT++`[37] discrete event simulator. All the simulations were performed on a computer with an Intel Pentium 4 2.4GHz processor, 512KB Cache and 1GB RAM. Our machine is installed with Red Hat Enterprise Linux WS release 4.

3.4.1 Configuration

We considered a system with 100 routers arranged in a balanced binary tree topology, whose depth was $\lfloor \log 100 \rfloor = 6$, and diameter was 12. Subscriptions and events are assumed w.l.o.g. to contain only numeric attributes. For instance, a subscription to textbooks might be $S = (\text{year} \in [1999, 2004], \text{price} \in [30, 60])$ and an event advertising a book might be $E = (\text{year} = 2002, \text{price} = 40.50)$. Note that lookup reuse strategy is not restricted to numeric data, and can be used with any form of events and subscriptions.

Each experiment uses a total amount of $10,000$ subscriptions and $1,000$ events. They are evenly distributed across all the routers at the beginning of a simulation. When the simulation starts, every router broadcasts its local subscriptions to populate the distributed routing tables. After all subscriptions have been registered and indexed at the routers, routers start publishing events which are delivered to interested subscribers by using the selected event forwarding algorithm.

We evaluated the performance of event forwarding algorithms under different scenarios, which were characterized by a parameter called *matching density*. Informally, the matching density is the average fraction of the subscriptions that match a published event. We consider inputs with the following matching densities: $0.1\%, 0.2\%, 0.5\%, 1\%, 5\%, 10\%, 15\%$ and 20%. For example, with a matching density of 0.5%, each event will match approximately $10,000 \times 0.5/100 = 50$ subscriptions.

We also tested Partial-Reuse algorithm by trying different thresholds: $20, 40, 60, 80$ and 100. The thresholds impose an upper bound on the number of matching subscriptions to search, when a router uses any content-match algorithm to evaluate an event.

Performance metric: We measured the *event processing overhead*, which is the total time spent by all routers in processing events. Note that this does not include the time for setting up the system, or for propagating subscriptions. An event forwarding strategy is better if the event processing overhead is smaller.

3.4.2 Data model

In the context of numeric data, an event can be thought of as a point in high-dimensional space and a subscription is a hyper rectangle in the same space. The space dimension is equal to the number of available attributes. We create 5 attributes for events and subscriptions in our simulation. For a single attribute, we consider that an event value is a random number drawn from the range $[0, 1]$ and a subscription predicate is a random interval within the same range.

One challenge in the evaluation of publish-subscribe system has been the lack of real life trace data. In the absence of this, we experiment with two distributions for events and subscriptions. Both of them are widely used in the pub-sub literature [46, 47, 8, 30].

Normal distribution Riabov *et.al.* derived this model in [46, 47] based on a systematic study of people's interests on the stock market transactions. It approximated the basic properties of real-life subscriptions, while being tuned for rectangle-based subscriptions like what we study in this paper.

For a single attribute, the event value follows a normal distribution. The center of a subscription interval also follows a normal distribution. The two distributions share the same mean, but may differ in the variance. The length of a subscription interval follows a Pareto distribution. The Pareto distribution is formulated as $\Pr(X>x)=\left(\frac{x}{a}\right)^{-b}$ for all $x \geq a$ and $b > 1$. The mean of a Pareto distribution is $\frac{ba}{b-1}$.

We experiment with two normal distributions: *norm-1* and *norm -2*. We use $(\mu_1, \sigma_1) = (0.5, 0.25)$ for events and $(\mu_2, \sigma_2) = (0.5, 0.125)$ for the center of subscription interval in norm-1. In norm-2, we use $(\mu, \sigma) = (0.5, 0.25)$ for both events and the center of subscription interval. For both distributions, we set $b = 19$ and vary a to generate subscription intervals of different

lengths. By adjusting the average interval length, we are able to control the probability that an event matches a subscription on a single attribute. This further leads to the desired matching density.

Uniform distribution Researchers used this model in [8, 30]. For a single attribute, the event value is uniformly drawn from the range $[0, 1]$. The length ℓ of a subscription interval is fixed as some constant, while one of its endpoints is chosen uniformly from the range $[0, 1 - \ell]$. If we use γ to denote the matching density and let n represent the number of attributes, then the connection between γ and ℓ is roughly $\gamma = \ell^n$.

3.4.3 Subscription indexing

Routers often build subscription index to expedite content-based event matching. Though lookup reuse intends to replace expensive content-match with cheap hash-table lookup, it can not completely eliminate them. This is true at least with the event source. So we still need some form of subscription index at every router. However the exact form of indexing is not a major concern, as long as content-match runs significantly slower than hash-table lookup.

Content-based event matching has been extensively studied in the literature. They can be mainly classified into two categories: counting algorithm [12, 19, 40] and tree-based approach [28, 7, 1, 50]. We pick one indexing out of each category and implement them in our simulation. In this section, we give a brief introduction about the selected subscription index.

Counting algorithm The counting algorithm maintains a counter for each subscription that records the number of predicates in a subscription which are satisfied by an event. Given an event, the algorithm iterates over the event's attributes. For each attribute, it finds all subscriptions whose corresponding predicate is satisfied by the event and increments the counters of these subscriptions. After going over all the attributes, the algorithm returns subscriptions whose counter values equal their number of predicates.

K-d tree algorithm This demonstrates an application of the well-known data structure "K-d Tree" to the indexing of numeric subscriptions. K-d tree corresponds to a recursive partitioning of a k-dimensional space. It is a useful data structure for multidimensional range searching and the details can be found in [2].

If subscriptions and events have k attributes, we can treat them as $2k$ dimensional points as follows. We can transform an event $e = (x_1, x_2, \ldots, x_k)$ to $e' = (-x_1, x_1, -x_2, x_2, \ldots, -x_k, x_k)$, and a subscription $s = ([l_1, r_1], [l_2, r_2], \ldots, [l_k, r_k])$ to $s' = (-l_1, r_1, -l_2, r_2, \ldots, -l_k, r_k)$.

If s matches e, we have every coordinate of s' must be no less than the corresponding coordinate of e'. In other words, for a given event $e = (x_1, x_2, \ldots, x_k)$, all the subscriptions that match e correspond to the points falling in the region $([-x_1, +\infty], [x_1, +\infty], \ldots, [-x_k, +\infty], [x_k, +\infty])$. So content-based event matching can be transformed into multidimensional range search, which in turn can be efficiently answered by K-d tree.

3.4.4 Summary

We evaluate the performance of lookup reuse by employing it in conjunction with two representative content-match algorithms: the counting and the K-d tree algorithm. Due to the absence of trace data, we experiment with three distributions of events and subscriptions: *norm-1, norm-2 and uniform*. This provides in total six combinations of [content-match algorithm, data distribution model]. For each individual pair, we measure the performance of selected strategies under different scenarios, which are characterized by the *matching density*. We select the following discrete matching densities: 0.1%, 0.2%, 0.5%, 1%, 5%, 10%, 15% and 20%. Finally we tested different thresholds for Partial-Reuse strategy. They are 20, 40, 60, 80 and 100.

3.5 Simulation results

We make at a high level the following conclusions over the simulation results:

(1) Lookup reuse provides faster event forwarding regardless of the distributions of events and subscriptions. For example, by combining lookup reuse with the counting algorithm,

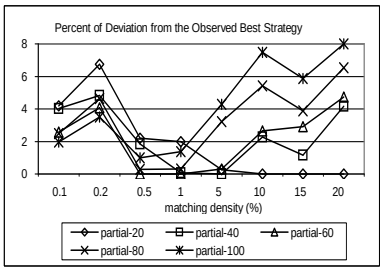
we can reduce the event processing overhead on average by 39% to 48% under the listed data distributions. The average is taken over various inputs with small and high matching densities.
(2) Lookup reuse provides faster event forwarding regardless of the underlying content-match algorithm. Although K-d tree runs much faster in content-match than the counting algorithm for numeric subscriptions, we can still achieve an average reduction of the event processing overhead by 25% - 35% under the listed data distributions.
(3) In most cases, Always-Reuse is not the best strategy for reusing lookups. In contrast, Partial-Reuse outperforms both Never-Reuse and Always-Reuse on most inputs, suggesting that Partial-Reuse is the best strategy for reusing lookups.
(4) Most benefits of Partial-Reuse can be realized by reusing a fairly small number of keys, about 40-60. Even when the matching density is very high (say 20%, where each event matches about $2,000$ subscriptions), a threshold of 60 lookups works well, and yields the best performance among the tested strategies.

The low threshold for Partial-Reuse is good news from the standpoint of message overhead. A threshold of 60 means that each event carries at most 60 keys. Since each key is a hash value taking 4 bytes, the message overhead is no more than 240 bytes, maybe much smaller. When compared with the size of an event, which is typically an XML document or database tuples, the message overhead may not be significant. Furthermore, a low threshold means that no matter how popular an event is, the load of matching an event or performing key lookups at a router does not increase too much due to lookup reuse. This leads to better load balanced event forwarding.

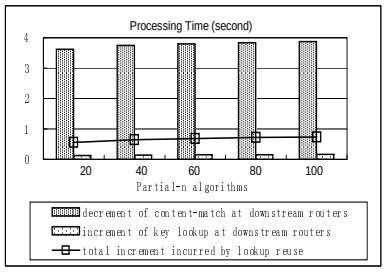
We now analyze the results of the [counting, norm-1] combination. In all the graphs, *Partial-N* denotes the Partial-Reuse strategy with a threshold of N. Figure 3.1(a) tabulates the event processing overhead of various lookup reuse strategies under different matching densities. We can see that the performance of Always-Reuse degrades very quickly as the matching density increases. In contrast, Never-Reuse is not appealing when the matching density is below 15%. Overall Partial-Reuse outperforms both Never-Reuse and Always-Reuse on most inputs.

	0.1%	0.2%	0.5%	1%	5%	10%	15%	20%
Never	4.51	5.28	5.71	6.12	5.89	5.47	5.01	4.43
Partial-20	1.95	2.21	2.47	2.80	3.46	3.79	4.10	4.38
Partial-40	1.94	2.17	2.46	2.74	3.45	3.87	4.15	4.56
Partial-60	1.92	2.16	2.41	2.74	3.46	3.89	4.22	4.59
Partial-80	1.91	2.17	2.42	2.75	3.56	3.99	4.26	4.67
Partial-100	1.90	2.14	2.44	2.78	3.60	4.07	4.34	4.73
Always	1.87	2.07	2.54	3.35	11.77	30.00	62.29	104.88

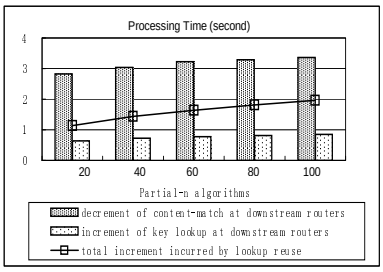
(a) Event processing overhead of lookup reuse strategies (time in seconds)



(b) Relative performance of Partial-Reuse strategies



(c) Tradeoff in Partial-Reuse strategies(0.2% matching density)

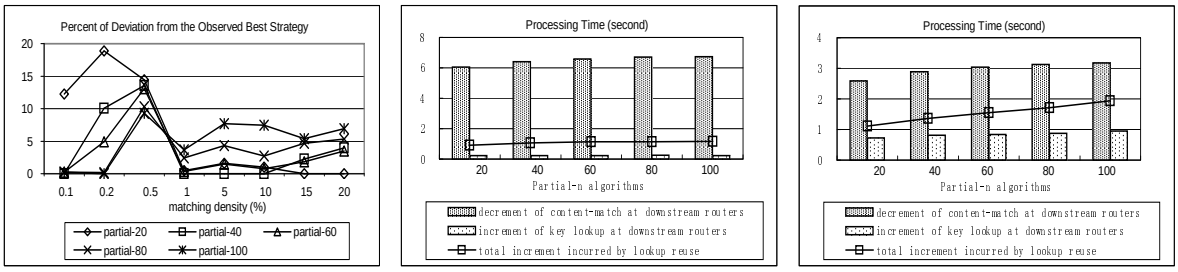


(d) Tradeoff in Partial-Reuse strategies(10% matching density)

Figure 3.1: Counting algorithm, norm-1 distribution

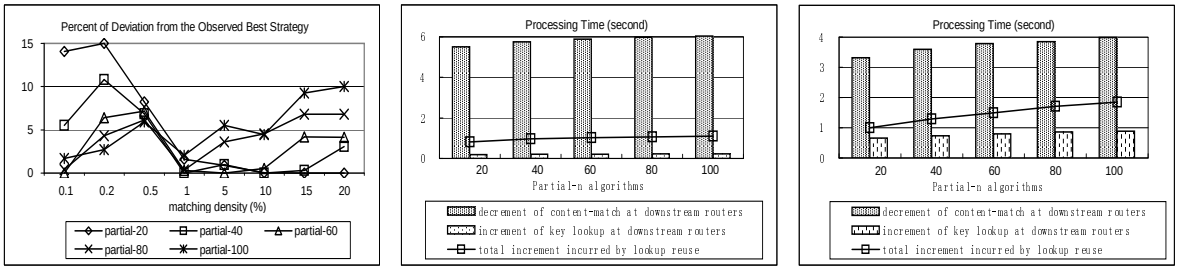
Figure 3.4(a) compares the relative performance of different Partial-Reuse strategies. The relative performance measures the deviation in time of each forwarding strategy from the best strategy observed on every input. For instance, if the overhead of Partial-40 on an input is 1.2 seconds, and the shortest time measured among all forwarding strategies for the same input is 1.0 second, then the graph shows a deviation of 25% for Partial-40. In our simulation, Never-Reuse has a deviation declining from 142% down to 1.1%. Always-Reuse has a deviation climbing from 0% up to 2294%. Both curves are not drawn in Figure 3.4(a), as they don't fit easily into the graph. It is clear, from Figure 3.4(a), that there is no absolute winner in that no single forwarding strategy is consistently the best over all inputs. However, all Partial-Reuse strategies illustrated here come close to the optimal in every case. It can be seen that on every input, their overhead is within 8% of the optimal for that specific input.

To better interpret the displayed results, we break down the event processing overhead into three parts: (1) time of content-match at the event source. (2) time of content-match at downstream routers and (3) time of key lookup at downstream routers. The reason why Lookup Reuse gains a benefit is because it replaces expensive content-match with cheap key lookup



(a) Relative performance of Partial-Reuse strategies (b) Tradeoff in Partial-Reuse strategies(0.2% matching density) (c) Tradeoff in Partial-Reuse strategies(10% matching density)

Figure 3.2: Counting algorithm, norm-2 distribution



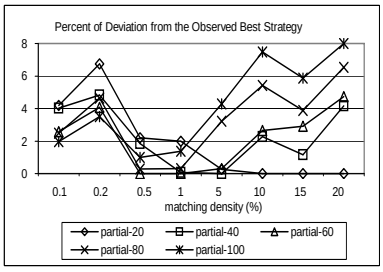
(a) Relative performance of Partial-Reuse strategies (b) Tradeoff in Partial-Reuse strategies(0.2% matching density) (c) Tradeoff in Partial-Reuse strategies(10% matching density)

Figure 3.3: Counting algorithm, uniform distribution

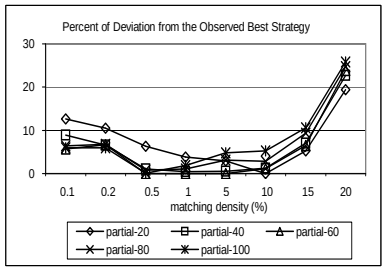
at downstream routers. Thus we expect to see a decreasing of item(2) and an increasing of item(3) at these places. The tradeoffs are illustrated in Figure 3.1(c) and 3.1(d). We infer that by investing a little time on key lookup, the cost of content-match at downstream routers has declined significantly irrespective of the matching density. The leverage effect can be clearly seen if you compare the height of adjacent bars for every Partial-N algorithm. As to the event source, it spends more time in content-match in order to generate the keys to be reused by downstream routers. The lines in Figure 3.1(c) and 3.1(d) show the total increment in time related to Lookup Reuse. It is a combination of the increment in both item(1) and item(3). We can see even with the addition of the extra cost at event source, the decrement in item(2) is still large enough to counteract the total increment. Thus Lookup Reuse helps to reduce the event processing overhead.

The results of the other two distributions, when employing lookup reuse with the counting algorithm, are shown in Figure 3.2 and 3.3. Qualitatively the trends are similar to the norm-1 distribution. For most inputs all illustrated Partial-Reuse strategies incur an overhead within 10% of the optimal. When the matching density is high, most of the benefits of lookup reuse can be achieved by reusing just a small number of keys, about 20-40. Although the same configuration yields a worse performance when the density is below 0.5%, it is noticeable that the relative performance of Never-Reuse under these inputs goes beyond 250%.

We also conducted simulations by employing lookup reuse with the K-d tree algorithm. For brevity, we compared the performance of Partial-Reuse when used in conjunction with the two content-match algorithms under the norm-1 distribution. The results are given in Figure 3.4. It shows that the benefits of lookup reuse is less pronounced when the K-d tree algorithm is in charge. This is because K-d tree seems more efficient in content-match than the counting algorithm, according to the time listed in Figure 3.4(c). But the relative performance of Partial-Reuse is still within 10% on most inputs for the K-d tree algorithm. The comparison reveals that by switching to a more efficient content-match algorithm, we didn't observe a drastic degeneration on the performance of Partial-Reuse strategies. Therefore the exact form of subscription indexing is not crucial to the performance of Lookup Reuse strategies.



(a) Relative performance of Partial-Reuse algorithms (counting)



(b) Relative performance of Partial-Reuse algorithms (K-d tree)

	counting	K-d tree
0.1%	1052	182
0.2%	1007	181
0.5%	857	173
1%	720	144
5%	391	77
10%	249	42
15%	177	42
20%	101	19

(c) Average time of evaluating a single event(time in microseconds)

Figure 3.4: Comparison of the counting and K-d tree algorithm (norm-1 distribution)

Figure 3.4(b) did show that when the matching density is between 15% and 20%, Partial-Reuse exhibited a certain kind of degeneration. If we correlate the trend reflected by Fig-

ure 3.4(b) with the numbers listed in Figure 3.4(c), it can be inferred that the advantage of Partial-Reuse starts to diminish if the average time spent by K-d tree in content-match is below 40. So we conclude that Lookup Reuse will gain a benefit as long as content-match runs *significantly* slower than key lookup. Such property is important, as it justifies why we want to forgo content-match for key lookup at downstream routers.

3.6 Impact of subscription covering

So far we assume that subscriptions are broadcasted inside the pub-sub network. Recently researchers [50, 39, 30, 57, 61] proposed to take advantage of the covering relationship among subscriptions to realize efficient subscription propagation.

Definition 1 *Let $N(s)$ denote the set of all events that match subscription s. Let s_1 and s_2 be two arbitrary subscriptions. We say that s_1 covers s_2, denoted by $s_1 \supseteq s_2$, iff $N(s_1) \supseteq N(s_2)$.*

Subscription covering (defined above) is an effective way to reduce routing table sizes and speed up event matching. A pub-sub router can exploit covering relationship among subscriptions as follows: Let I represent one interface at a pub-sub router. The router received a subscription s_2 and decided to forward it through I. We can check to see if some existing subscription s_1, which has been forwarded through I, covers s_2. If it does, then s_2 is not forwarded since all events matching s_2 will be received due to the prior propagation of s_1. Repeating this process at every intermediate router can result in a significant reduction in the number of forwarded subscriptions. As a consequence, the sizes of routing tables decrease, leading to faster content-based event matching at every router.

Efficient utilization of covering demands the support of new data structure. To be specific, we need novel indexing to organize subscriptions for fast detection of the covering relationship. Meanwhile because of unsubscribing, some subscriptions which were previously covered may no longer be covered by any other subscription. Such subscriptions should be identified and forwarded to other routers. Otherwise clients may miss interesting events in the future. To facilitate this task, we need a separate data structure to maintain the detected covering

relations. However the discussion of these techniques goes beyond the scope of this paper. Interested readers can check out the papers cited in the beginning of this section.

In this section, we want to investigate the impact of subscription covering to the proposed Lookup Reuse strategies. First of all, the presence of covering will invalidate Always-Reuse and other similar approaches [9, 29] proposed in the literature. If covering is in place, then a router just forwards a minimal set of received subscriptions to its neighboring routers. Under this circumstance, the event source can not always generate a complete list of keys of matching subscriptions, because some matching subscriptions are not propagated to the event source due to the presence of covering. If we continue to forward events only based on key lookups inside the network, we undertake the risk of missing interested subscribers.

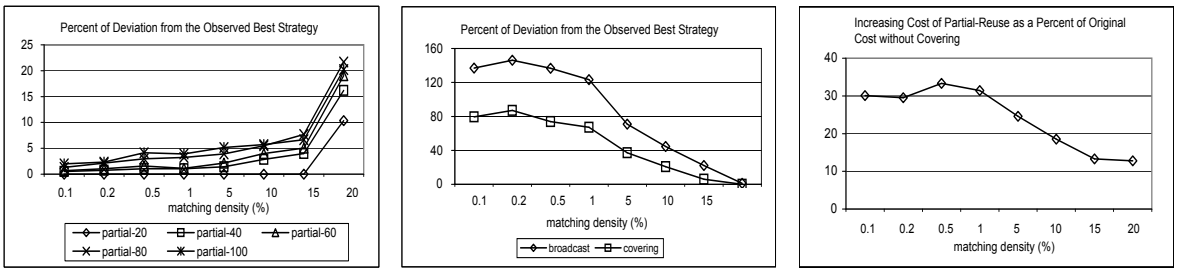
On the other hand, Partial-Reuse remains a valid solution. It already contains in Algorithm 3.5 the instructions to deal with the case when the received key list is incomplete. The modified Partial-Reuse algorithm, which works in the presence of covering, is given in Algorithm 3.6.

Algorithm 3.6 Partial-Reuse (*event* e, *set<key>* K) /* downstream routers */

```
1:  let K_I represent the set of subscription keys associated with interface I
2:  for each interface I except the one from which e is received do
3:    if I is the local interface then
4:      use content-match to find all matching subscriptions
5:      deliver e to interested local subscribers
6:    else
7:      if (K ∩ K_I) ≠ φ then
8:        forward < e, K ∩ K_I > through I
9:      else
10:       use content-match to find at most n matching subscriptions
          and store their keys in K_≤n
11:       if K_≤n ≠ φ then
12:         forward < e, K_≤n > through I
```

Our next question is how well Partial-Reuse performs with the introduction of covering? To answer this question, we conducted simulations to measure the performance of Partial-Reuse under the new model of subscription propagation for the [counting, norm-1] combination. Figure 3.5(a) shows that the relative performance of Partial-Reuse still stays within 10% of

the observed best strategy on most inputs. Furthermore with the introduction of covering, Partial-20 consistently yields the best performance among selected Partial-Reuse strategies. It reinforces the observation that most of the benefits of Partial-Reuse can be achieved through reusing a small number of keys.



(a) Relative performance of Partial-Reuse strategies

(b) Relative performance of Never-Reuse with/without covering

(c) Increasing cost of Partial-Reuse(Partial-20)

Figure 3.5: Counting algorithm, norm-1 distribution, under subscription covering

Figure 3.5(b) compared the performance advantage Partial-Reuse holds over Never-Reuse with/without covering. We notice that the advantage is roughly halved across all the matching densities with the introduction of covering. This is mainly due to the increasing cost of Partial-Reuse as indicated in Figure 3.5(c). We notice that this increasing is largely attributed to the slowed content-match at downstream routers. By comparing Algorithm 3.5 and Algorithm 3.6, we inferred that with the introduction of covering, there is no way for a router to judge whether the received key list is complete or not, even though it could be complete in reality. Consequently some part of content-match, which originally could be replaced by key lookups, now have to be performed. This directly leads to the increasing cost of Partial-Reuse under the new subscription propagation model.

CHAPTER 4. Efficient management of subscription covering

In Chapter 2, we discussed a basic model of propagating subscriptions inside a distributed publish/subscribe system. One problem with the above approach is that every router needs global knowledge of all subscriptions, which results in large routing tables. This in turn leads to slower matching of notifications and greater latency for event delivery. Clearly this is a serious problem because timely delivery of events is important in most systems. To alleviate this, multiple groups [58, 39, 50, 30, 57, 61, 10] suggested exploiting the *covering relations* among subscriptions to significantly reduce routing table sizes.

In this chapter, we will define the notion of subscription covering and explain how it can be used to optimize the propagation of subscriptions. The introduction of covering brings new problems though, i.e. fast detection and maintenance of covering relations. In the literature, the subscription POSET and its variants are proposed as solutions to this problem, combining the detection and maintenance of covering into a single data structure. We take a different approach by decoupling the detection of covering from its maintenance. We propose a generic framework consisting of a subscription index to support fast detection of covering and an easy-to-update graph dynamically maintaining detected covering relations. We formalize the interaction between the two components and show that a simple heuristic provides a 2-approximation algorithm for optimizing the number of covering queries on average. Experimental results show that the proposed framework offers a significant performance improvement over existing POSET family.

Being an indispensable part of the generic framework, the detection of covering is of paramount importance to the framework's efficiency. However it remains a hard combinatorial problem, in spite of the help of a subscription index. Its special case, covering detection

for numeric subscriptions, can be modeled as a multidimensional point dominance problem, for which no worst-case time efficient solution exists.

To alleviate the complexity, we introduce a novel approach called *approximate subscription covering*, which provides much of the benefits of covering at a fraction of its cost. By forgoing an exhaustive search for covering subscriptions in favor of an approximate search, it is shown that the time complexity of covering detection can be dramatically reduced. The tradeoff between the efficiency of covering detection and the approximation error is demonstrated through the analysis of a subscription index built for covering detection based on space filling curves.

4.1 A generic framework for the management of subscription covering

Subscription covering is an effective way to reduce the complexity of content-based routing and avoid unnecessary proliferation of subscriptions throughout the system. The notion of covering is formalized in Definition 2. For example, subscription $s_1 = \{\text{airline=ANY}, \text{price} < 350\}$ covers $s_2 = \{\text{airline=UA}, \text{price} < 300\}$, because the former has a broader range of air tickets to select.

Definition 2 *Let $N(s)$ denote the set of all notifications that match subscription s. Let s_1 and s_2 be two subscriptions. We say that s_1* covers *s_2, denoted by $s_1 \supseteq s_2$, iff $N(s_1) \supseteq N(s_2)$.*

We can take advantage of covering relations to optimize subscription propagation as follows: Let I represent an interface at a pub-sub router. The router received a subscription s_2 and decided to forward it through I. We can check if there exists some subscription s_1, which covers s_2 and was forwarded through I. If s_1 exists, then s_2 is not forwarded since all events matching s_2 will be received due to the prior propagation of s_1. Repeating this process at every router can result in a significant reduction in the number of forwarded subscriptions. The advantages are twofold: (1) The number of subscribe and unsubscribe control messages is reduced, and (2) The sizes of routing tables decrease, leading to faster notification forwarding.

While these advantages are well recognized, subscription covering by itself is a hard problem. First, the publish-subscribe system is a dynamic online messaging system. Clients can

submit and withdraw subscriptions at any time. With the introduction of covering, newly arrived subscriptions may not be forwarded to other routers if they are already covered by existing subscriptions. Such optimization can pose problem when we unsubscribe a registered subscription s. A router needs to check those subscriptions which were covered by s, as they may no longer be covered by any other subscription after the removal of s. Such subscriptions should be identified and forwarded to other routers. Otherwise clients may not receive interesting notifications in the future. In order to quickly identify these subscriptions, we need a data structure to maintain detected covering relations. We name such data structure the *relation graph*. Second, for a large distributed system, a pub-sub router can receive hundreds of subscriptions in a second. The subscription database can be populated fairly quickly. As a consequence, it gets much harder to find existing subscriptions which cover a new one. Similar to the database domain, some kind of *subscription index* is desired to expedite the covering detection. The two components are connected in a way that the construction of a relation graph relies on the subscription index to detect existing covering relations.

4.1.1 Relation graph

Relation graph is used to maintain detected covering relations. In a relation graph, nodes represent individual subscriptions. A directed edge indicates detected covering relation between a pair of subscriptions.

The first relation graph was presented in the Siena system [10]. Carzaniga *et.al.* built the subscription POSET(partially ordered set) to manage subscriptions registered at a pub-sub router. The Siena POSET is limited in terms of the scalability, because it attempts to maintain a fine-grained hierarchy of the covering relation. Tarkoma *et.al.* relaxed some constraints and proposed POSET-derived forest and other variants that perform considerably better under frequent subscription additions and removals. These data structures in the POSET family are going to be discussed in Section 4.3.

The construction of a relation graph relies on a separate mechanism whose function is to find existing subscriptions which cover the new one. For POSET structures, they use the same

data structure for the purpose of covering detection. Their algorithm is to traverse the POSET in breadth-first-order and test if a visited node covers the new subscription on the fly. This is a linear time solution. However the detection of covering can be made more efficient by building some form of subscription index. Unlike the POSET structures combining the detection of covering with its maintenance, we decoupled the two tasks and built separate data structure for either of them.

In our approach, the detection of covering is facilitated by a subscription index and the relation graph depends on the service of the index to realize fast updating. Since covering detection is a time-intensive operation, the cost of updating a relation graph is largely determined by the frequency of which we query the subscription index. We investigated the interaction between the two components and conducted a formal analysis to show that a simple structure called G_1, where at most one covering subscription is to be searched (if any) for each new subscription, provides a 2-approximation algorithm for optimizing the total number of queries on average to the underlying index. To the best of our knowledge, this is the first study to analyze the tradeoffs among possible structures of a relation graph. Our theoretical findings are corroborated by our experiments, which demonstrate that G_1 outperforms other structures over a range of system parameters.

4.1.2 Indexing for covering detection

Similar to the database domain, subscription index can be used to quickly detect covering relations. Due to frequent subscription additions and removals, the index should be a dynamic data structure supporting the following operations: (1) Fast insertion and deletion of subscriptions. (2)Given a new subscription, return a set of existing subscriptions which cover the new one. Our approach allows a user to choose existing index or implement his own. Examples of existing indexes built for covering detection can be found in [30, 50, 34]. A brief description about them is given in Section 4.5.

4.1.3 Framework composition

Our solution framework is structured in two layers:

- The upper layer is a relation graph, which stores the detected covering relations among subscriptions.
- The lower layer is a subscription index, which is used to detect new covering relations when subscriptions are added to or removed from the database.

The framework supports the following operations:

Subscribe: When a new subscription s arrives, we add it to the relation graph. The index is used to detect if there exists subscriptions covering s. If s is covered, it is not forwarded. The index can find covering subscriptions (if any) by examining only a fraction of the subscriptions in the database.

Unsubscribe: When a registered subscription s is unsubscribed, we remove s from the relation graph and locate subscriptions which lose all their coverings due to the removal of s. We query the index to determine if these subscriptions are still covered. If not, we need to forward them to other routers.

What makes the framework appealing is that it decouples the detection of covering from its maintenance. By taking a modular approach, we provide the flexibility of enhancing the performance by improving the design of individual component separately. In this paper, we are going to show that a simple graph design called G_1 enjoys faster updating than existing POSET structures. If sophisticated subscription index is in use, the performance gain will be more pronounced.

We made the following contributions:

- A generic solution framework to efficiently manage covering relations among subscriptions in a distributed publish-subscribe system and provide the maximal flexibility of customization.
- A theoretical proof that a simple relation graph tracking a single covering relation for each subscription is sufficient to derive an algorithm whose average total covering de-

tection needs are within a factor of two of the optimal. This statement is reinforced by experimental validation.

- A comparative study of the proposed solutions to the problem of subscription covering. Experiments show that our modular approach yields significant performance gain over monolithic POSET structures.

4.2 Related work on the management of subscription covering

The first data structure for maintaining the covering relation was the POSET(partially ordered set) implemented in the Siena [54] system. For each subscription, POSET stores the information of its immediate predecessors and successors. These terms are explained in Section 4.3. In short, POSET reveals a fine-grained hierarchical covering relation among all the subscriptions. Due to this extravagant feature, POSET is found not easy to update and limited in terms of the scalability [54].

Tarkoma [58] relaxed some constraints in the construction of POSET and designed POSET-derived forest and other variants that perform considerably better under frequent subscription additions and removals. In general, a pub-sub router has multiple interfaces used to connect with neighboring routers. An incoming subscription often needs to be forwarded through several interfaces. The POSET variants are incorporated with techniques optimizing those interface-specific operations.

POSET structures lack an efficient way to detect the covering relations required by its construction or updating. Researchers explored the building of subscription index to expedite the detection of covering. Examples of generic subscription index used for covering detection include MBD(modified binary decision diagrams) [30] used in the Toronto Publish/Subscribe System and the counting algorithm [34] applied in the Rebeca system.

For subscriptions containing only numeric attributes, Shen *et.al.* [50] showed the equivalency of subscription covering to the problem of multidimensional range search and used spatial data structures like K-d tree and space filling curve to organize subscriptions. In a recent work, Ouksel *et.al.* took a novel approach to use a probabilistic algorithm for covering detection.

The complexity of their algorithm is O(nm), where n is the number of subscriptions and m is the number of attributes.

So far the research on subscription covering proceeded in two parallel paths. The studies inventing new data structures to maintain covering relation lacked efficient method to detect covering. On the other hand, the researches looking for efficient solution to the detection of covering didn't mention how to effectively maintain the discovered relations as well. We are the first group which investigates the interaction between the two components and proposed a coherent framework to completely address the covering problem.

Finally subscription merging is considered as an advanced technique used to produce more compact routing tables in a publish-subscribe system. The MBD index provides good heuristics about mergeable subscriptions. Triantafillou *et al.* presented a summarization structure [61] to compress the subscriptions stored by a pub-sub router. They present novel algorithms to efficiently propagate these subscription summaries and use these summaries to route matched events.

4.3 Design of relation graph

A relation graph maintains detected covering relation among subscriptions. It is instrumental in deciding whether a received subscription is covered, thus needs not be forwarded. In this section, we explore the design space of a relation graph.

In a distributed publish-subscribe system, a router has multiple interfaces used to connect with neighboring routers. It is common for a pub-sub router to forward an incoming subscription through several interfaces. To simplify our description, we assume from now on that a relation graph is created for every interface of a router to maintain the covering relation among all the subscriptions that a router passes to the associated interface.

There exists many ways to design a relation graph. The following section reviews existing proposals. The construction of a relation graph depends on a separate mechanism whose function is to find existing covering relations. This mechanism can be built into the graph by itself or can be delegated to a separate data structure. The POSET structures take the

former approach, while our modular framework follows the latter one. In our approach, the detection of covering is facilitated by a subscription index. Thus the design of a relation graph is relatively independent. To analyze the tradeoffs, we simply assume an abstract service interface provided by the subscription index and ignore its concrete implementation.

4.3.1 Dense graph G^*

Probably the most straightforward design is a dense graph G^* defined as follows. The nodes of G^* are S, the set of subscriptions. There is an edge in G^* directed from $s_1 \in S$ to $s_2 \in S$ iff $s_1 \supseteq s_2$. In other words, G^* maintains *all* pairwise covering relations among the subscriptions in S. Clearly, we only need to forward those subscriptions which have no incoming edges in G^*. The rest are covered by at least one other subscription and do not need to be forwarded (unless this subscription was received before its covering subscription).

The problem with G^* is that it may have too many edges and may be too expensive to maintain as subscriptions are added and deleted. For example, if a new subscription s arrives, which is covered by all existing subscriptions, then edges have to be created from each existing subscription into s, and this would take time $O(|S|)$ (this is in addition to the time taken to discover all the covering relations, which is even more expensive). It might happen that s is unsubscribed immediately, so that all this work is wasted. This example indicates G^* can often be an overkill.

4.3.2 Subscription POSET

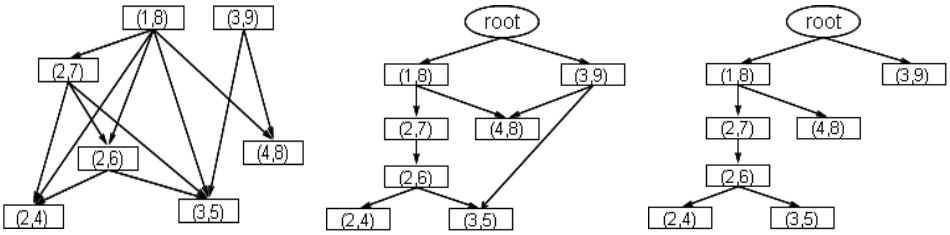
Subscription POSET represents a partially ordered set of subscriptions. It is used in the Siena publish-subscribe system [10] for maintaining covering relation among subscriptions. This partial order is defined by the covering relation among subscriptions. In particular, it introduces the new concepts of *immediate predecessors* and *successors* of a subscription.

Definition 3 *Subscription s_1 is an* **immediate predecessor** *of subscription s_2 and s_2 is an* **immediate successor** *of s_1 if and only if $s_1 \supset s_2$ and there exists no other subscription s_3 in the POSET such that $s_1 \supset s_3 \supset s_2$.*

We know that covering relations are transitive, but POSET only connects a subscription with its immediate predecessors and successors. Thus it contains less number of edges than G^* does. In fact, POSET can be considered as a variant of G^*, where redundant edges are removed from G^*, but every directed edge in G^* is implicitly contained in the graph through a directed path between the vertices.

The "top-level" subscriptions, which are referred as "roots", are not covered by any other subscription in the POSET. Thus they should be forwarded by a pub-sub router. Non-root subscriptions generally don't contribute to the forwarding traffic, unless a subscription was received before its ancestors.

The nice thing about POSET is that it exhibits a fine-grained hierarchical covering relation among all the subscriptions. However this extravagant granularity could compromise its performance. The bottleneck lies in the search of the tightest covering relations. In addition, it is required to return a *complete* set of predecessors and successors for every new subscription. In face of the limitation, Tarkoma *et.al.* [58] proposed the following POSET variant.



(a) Dense Graph G^* (b) Siena POSET (c) POSET-derived Forest

Figure 4.1: Proposed graph structures

4.3.3 POSET-derived forest

The POSET-derived forest is similar to the Siena POSET with one exception that each subscription only keeps one immediate predecessor and a subset of available immediate successors. The addition and removal algorithms are described in [58]. With fewer number of predecessors and successors to maintain, the forest is demonstrated to have better performance than the POSET under frequent subscription additions and removals.

POSET-derived forest preserves some important properties of POSET, such as *minimal cover* and *sibling purity*. The formal definitions are given in [58]. In brief, "minimal cover" says that the root nodes in the forest correspond to a minimal set of the subscriptions which are not covered, thus need to be forwarded by a router. The property of "sibling purity" claims that no two children with the same parent cover each other, or in other words form a parent-child relationship. It is easy to maintain sibling purity for the *add* operation, but more complicated for the *del* operation.

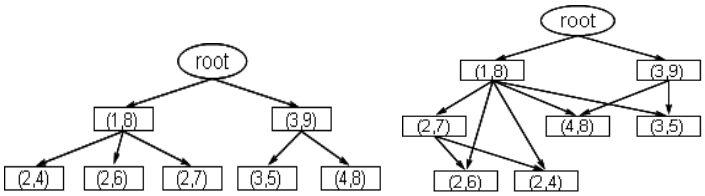
The Siena POSET and its variant share the common nature of looking for the tightest covering relation when inserting a node into the graph. This property relocates a node as far away from the roots as possible. It helps to constrain a node's degree by stretching the relation graph vertically. The del operation, which detaches predecessors and successors from the selected node and reconnects them properly, might benefit from this property. On the other hand, this attribute increases the complexity of the add operation. As we know, finding a single covering subscription is a time consuming operation. It will be more costly for discovering the tightest covering relation for each subscription. If you review the deletion operation in [58], it repeatedly invokes the add routine for every disconnected successor. Thus it is unclear whether the del operation will actually benefit from this property.

4.3.4 Flat graph G_k

In practice it is important to know whether a new subscription is covered or not. But it does not matter whether the detected covering relation is tight or loose. In other words, the fine granularity of covering relation exhibited by the POSET structures might be an overkill for the sole purpose of subscription covering. Therefore we are motivated to simplify the graph design. In short, we are interested in a directed graph G with vertex set S (the same as G^*), and edge set $E \subset E^*$. The edge set E must satisfy the following key property.

Property 1 *If $s \in S$ has a non-zero in-degree in G^*, then it should have a non-zero in-degree in G.*

This way, a new subscription which is covered by an existing one will never be forwarded, since it will have a non-zero in-degree in G^*, and hence, a non-zero in-degree in G.



(a) Example of G_1 (b) Example of G_2

Figure 4.2: Flat graph G_k

We now define a family of relation graphs G_k, which are much simpler to maintain than formerly mentioned structures. Let k be a natural number greater than or equal to 1. The vertex set of G_k is S and the edge set $E(G_k) \subseteq E^*$ has the following property: For subscription s, let $C(s)$ denote $\{t \in S,\ t \neq s \mid t \supseteq s\}$, i.e., the set of all subscriptions covering s (note that $C(s)$ might be empty). If $|C(s)| \geq k$, then there are incoming edges into s from exactly k other subscriptions belonging to $C(s)$ which cover it. If $|C(s)| < k$, then there are incoming edges into s from all subscriptions in $C(s)$. Clearly, G_k satisfies Property 1 for all $k \geq 1$.

4.4 Analysis of G_k

In this section, we analyze the cost of updating the G_k graph. First, we present the algorithms of adding to or removing from G_k a subscription. The major operation here is to search in the subscription index for covering relations. It is natural to use the number of covering queries as a metric to approximate the cost of graph updating. We will discuss the tradeoff caused by choosing different subscript k. We develop a probabilistic analysis to show that graph G_1 yields a 2-approximation algorithm for optimizing the number of covering queries on average.

4.4.1 Algorithms for addition and deletion

Let S denote the current set of received subscriptions. The algorithms to maintain G_k in the presence of additions and deletions to S follows. The add operation is described in Algorithm 4.1 and del operation in Algorithm 4.2.

Algorithm 4.1 Insertion: new subscription s arrives.

```
1: if s exists then
2:    increment its counter by one and return
3: else
4:    Create a new node for s in G_k.
5:    C ← cov_k(S, s).
6:    if |C| = 0 then
7:       forward s to other routers
8:    else
9:       update G_k by adding directed edges from every subscription in C to s
10:   add s into the index
```

We assume that the underlying subscription index provides us a function $cov_\ell(S, s)$. If s is covered by ℓ or more subscriptions in S, the function returns ℓ subscriptions which cover s. Otherwise, the function returns all the subscriptions which cover s (note that this set may be empty).

In addition, a pub-sub router can receive identical subscriptions from different clients and pass them to the same interface. So we keep a counter for every node in the relation graph to keep track of the frequency of a particular subscription.

4.4.2 Cost of maintaining G_k

For which value of k, does G_k give us the most efficient algorithm? Assuming that k is a small integer, the cost of handling subscribe is dominated by the cost of $cov_k()$. Since it is always more expensive to find more covering subscriptions, the cost of $cov_k()$ strictly increases with k. Thus, the cost of addition clearly increases with k.

Meanwhile the cost of handling unsubscribe might decrease. If k increases, the average in-degree of a node in G_k increases, making it less likely to lose all the covering relations once another subscription is deleted from G_k. However in any directed graph, the sum of the out-

Algorithm 4.2 Deletion: request to unsubscribe s arrives.

```
1: if the counter of s > 1 then
2:    decrement the counter by one and return
3: else
4:    delete s from G_k and the index
5:    let T be a set of children of s which has an in-degree of 0 in G_k.
6:    for each t ∈ T do
7:       C_t ← cov_k(S, t).
8:       if |C_t| = 0 then
9:          forward t to the neighboring router
10:      else
11:         adding directed edges from every node in C_t into t.
12:   if s was formerly forwarded then
13:      forward the unsubscription request to the neighboring router
```

degrees of all the nodes equals the sum of their in-degrees. Therefore the average number of out-neighbors of a node also increases with k. This means that we need to inspect on average more nodes about their covering status when a node in G_k is deleted.

The question is whether the increase in the cost of handling subscribes can be offset by the decreased cost of handling unsubscribes. We now present evidence to show that choosing $k = 1$ gives us a total cost which is close to the optimal, on the average. While this does not mean that $k = 1$ is always the best choice, it implies that it is usually a good choice. The above statement is also supported by our experimental results presented in Section 4.5.

4.4.3 Analysis of G_1

We consider a probabilistic model of the pattern of unsubscribes for our analysis. Consider a sequence $Q = o_1, o_2, \ldots o_n$ of n subscribe and unsubscribe operations. The subscribe operations may be for arbitrary subscriptions, and each unsubscribe operation unsubscribes to a *randomly chosen* subscription that is currently subscribed for. Let n_s denote the number of subscribe operations in Q and n_u the number of unsubscribe operations, so that $n_s + n_u = n$. We do not make any assumptions on the formats of the subscriptions, hence our analysis holds for any type of subscriptions, not just numeric ones.

The major cost in the algorithms for handling subscribes and unsubscribes is the time spent

in the routine $cov_k()$. Hence, our metric for the cost of handling Q, denoted by $cost_k(Q)$, is defined as the number of times a call is made to $cov_k()$. Note that $cost_k(Q)$ is a random variable since the unsubscriptions in Q are chosen randomly from the set of existing subscriptions.

Let q_s and q_u be the number of calls to $cov_k()$ made due to the subscribe and unsubscribe operations respectively.

$$cost_k(Q) = q_s + q_u \tag{4.1}$$

Since only one call to $cov_k()$ is made by any subscribe operation,

$$q_s = n_s \tag{4.2}$$

Lemma 1 *If the relation graph used is G_1, then*

$$E[q_u] \leq n_u$$

Proof: We label the unsubscribe operations $1, 2, \ldots n_u$. For $i = 1, 2, \ldots n_u$, let random variable X_i denote the number of calls to $cov_1()$ due to the ith unsubscription.

$$q_u = \sum_{i=1}^{i=n_u} X_i \tag{4.3}$$

By linearity of expectation, we have

$$E[q_u] = \sum_{i=1}^{i=n_u} E[X_i] \tag{4.4}$$

Suppose the ith unsubscribe operation was for subscription s. Then, X_i is equal to the out-degree of s in G_1 before the deletion of s. Since s is chosen randomly from the current set of subscriptions in G_1, $E[X_i]$ is equal to the expected out-degree of a vertex in G_1.

In any directed graph, the sum of the out-degrees of all the nodes equals the sum of the in-degrees. In graph G_1, the in-degree of every node is no more than 1, so that the sum of in-degrees is no more than $|S|$. Thus, the expected out-degree of a random node in G_1 is no

more than 1. We have $E[X_i] \leq 1$. From Equation 4.4, the lemma follows. ■

Theorem 1 *For every positive integer k,*

$$E[cost_1(Q)] \leq 2 \cdot cost_k(Q)$$

Proof: From Equation 4.1, we know $cost_1(Q) = q_s + q_u$. From Lemma 1 and Equation 4.2, $E[cost_1(Q)] = E[q_s] + E[q_u] \leq n_s + n_u$. Since the number of unsubscribe operations can never be greater than the number of subscribe operations, $n_u \leq n_s$. Thus, we have

$$E[cost_1(Q)] \leq 2n_s$$

However, for any k, the algorithm G_k has to make at least n_s calls to $cov_k()$, even in the unlikely scenario that further calls to $cov_k()$ are never made for unsubscribe operations. Suppose q^* denotes the minimum number of queries made by any G_k, $k \geq 1$.

$$q^* \geq n_s$$

Thus, we have $E[q] \leq 2q^*$, which proves the theorem. ■

The above theorem tells us that the expected number of calls to $cov_k()$ made by the G_1 algorithm is always within a factor of two from the optimal number. In addition, we know that the cost of $cov_k()$ increases with k, so that the covering queries made by G_1 are cheaper than queries made by any other G_k. This presents strong evidence that the G_1 algorithm is one of the best choices among all $G_k, k \geq 1$.

4.5 Experiments

The objectives of the experimental study are two-fold: First, we need to determine what is a good value of k for the graph family G_k. Second, we want to compare the performance of

our modular approach versus the POSET structures in the presence of frequent subscription additions and removals.

In this section, we first describe the experiment configuration. Next we briefly discuss the implementation of some subscription index used for covering detection. This makes it possible to instantiate our solution framework by combining the G_k graph with a particular subscription index. Finally we present the experimental results.

4.5.1 Configuration

The key parameters of the generated workload are: n = the total number of subscribe and unsubscribe operations; d = the number of attributes in each subscription; and p = the odds of an operation being unsubscribe.

For the sake of simplicity, we only consider subscriptions having numeric attributes. In this context, each constraint in a subscription is a range query. For example, we can have a subscription $s = \{\text{age} \in [25, 35],\ \text{weight} \in [110, 170]\}$. Furthermore we require w.l.o.g. that each constraint is an interval uniformly picked from the range $[0, 1]$. To be specific, two random numbers between 0 and 1 are generated, sorted and assigned to the ends of a constraint interval. The number of constraints available in a subscription range from 3 to 6.

The input is a sequence of mixed operations. Each operation is either a subscribe or an unsubscribe. Each subscribe adds a new subscription, and each unsubscribe removes a randomly chosen existing subscription. Each operation has a probability of p to be an unsubscribe, and $(1-p)$ to be a subscribe. Clearly $p < 0.5$ as there can't be more unsubscribes than subscribes. We vary the value of p from 0 to 0.4 at a step of 0.1. Notice that when $p = 0$, it corresponds to a scenario with only additions. We experiment with various input sizes, ranging from $10,000$ to $50,000$ operations.

Our performance metric is the total time taken by the covering data structure to process all the subscribe and unsubscribe commands. Clearly a data structure is superior if it yields a shorter running time.

4.5.2 Subscription index

In order to instantiate our generic framework, it is necessary to have a concrete implementation of the subscription index. Some examples of the index are cited in Section 4.2. We used the counting algorithm [34] and the K-d tree [50] in our experiment. A brief description follows.

An event in a publish-subscribe system is normally modeled as a set of (attribute,value) pairs. A subscription is defined to be a set of conjunctive predicates, each of which specifies a constraint on the corresponding event attribute. Under this model, we have an alternative definition for subscription covering.

Definition 4 *A subscription S_1 covers another subscription S_2 if and only if every predicate in S_1 covers the corresponding predicate in S_2.*

Based on the new definition, the counting algorithm maintains a counter for each subscription, which records the number of predicates in it that satisfy the covering relation on the associated attributes. Given a new subscription, the algorithm iterates over the subscription's attributes. For each attribute, it finds all subscriptions which cover the new one on the selected attribute and increments the counters of these subscriptions. After going over all the attributes, the algorithm returns subscriptions whose counter value equals its total number of predicates.

The counting algorithm allows a subscription to contain attributes of different data types. If subscriptions just have numeric attributes like in our study, then an index based on the K-d tree structure is deemed to be more efficient [50]. K-d tree corresponds to a recursive partitioning of a k-dimensional space. It is a useful data structure for multidimensional range searching originally described in [2].

In the context of numeric attributes, we can model subscriptions as hyper rectangles in a multidimensional space. For a subscription with k attributes, we can transform $s = ([l_1, r_1], [l_2, r_2], \ldots, [l_k, r_k])$ to $s' = (-l_1, r_1, -l_2, r_2, \ldots, -l_k, r_k)$. That is, a k-dimensional hyper rectangle is turned into a $2k$-dimensional point. It can be verified that subscriptions which cover s correspond to the points falling in the range $([-x_1, +\infty], [x_1, +\infty], \ldots, [-x_k, +\infty], [x_k, +\infty])$.

Thus the search of covering subscriptions can be reduced to multidimensional range search, which in turn can be efficiently answered by K-d tree.

The introduction of K-d tree is to illustrate the flexibility of our modular approach. We are going to show that if the counting algorithm is replaced by the faster K-d tree, the performance gain achieved by our modular approach will be much more significant.

4.5.3 Ideal structure of G_k

In Section 4.4, we defined a family of relation graphs G_k. The subscript k, which we call the "reporting mode", determines the maximum in-degree of any node in the graph. We use this name because the subscript k specifies the maximum number of covering subscriptions to be reported by a subscription index. Our experiments show that *among all the selected values of k, G_1 consistently yields the best performance.* This observation supplements our theoretical analysis given in Section 4.4.3.

In this section, the counting algorithm is used to implement the subscription index. Since all the scenarios output similar results, we only display the outcome of a set of scenarios characterized by ($n = 50000$, $d = 3$). Scenarios within that group have different frequencies of unsubscribe operations. The value of p ranges from 0 to 0.4.

It is clear from Figure 4.3(a) that G_1 consistently yields the shortest runtime. To interpret the result, we break down the processing time into two parts: (1) the time spent on subscribe and (2) the time spent on unsubscribe. For every $k(k \neq 1)$, we measure the difference between G_k and G_1 in terms of the two variables and plot them in Figure 4.3(b) and 4.3(c). The graphs show that the decreasing of item(2), as an effect of the increasing of k, is negligible in comparison with the increasing of item(1). The increasing of the processing time is thus attributed to the increasing of the subscribing time. Since the cost of adding a new subscription increases strictly with k, the processing time can be minimized by choosing $k = 1$.

But we want to know why the increasing of k brings limited benefit to the unsubscribe operation? At a first glance, it becomes harder to lose all the covering relations when some subscription is removed, if we increase a node's in-degree by using a larger k. In fact Fig-

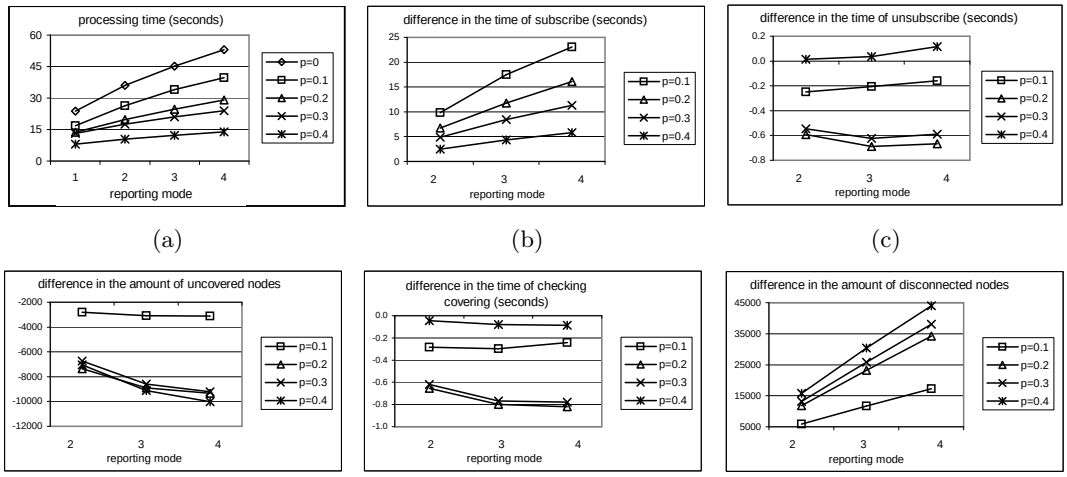


Figure 4.3: 50000 (un)subscribe operations. 3 attributes per subscription. The difference is relative to $k = 1$

ure 4.3(d) does show that the amount of nodes being left uncovered reduces drastically when k gets larger. The difference is measured by subtracting the corresponding amount induced by G_1. But it does not lead to the same level of reduction in the checking time as indicated in Figure 4.3(e). The checking time is a quantity used to measure the overhead of checking the covering status for every uncovered node during unsubscribe. It is realized by searching in the subscription index to look for more covering subscriptions. Clearly this cost increases with the value of k. Given the fact that every uncovered node was initially covered by k number of existing subscriptions, it simply gets much harder to find k number of additional covering subscriptions. Therefore the advantage brought by the reduction in the amount of uncovered nodes was impaired.

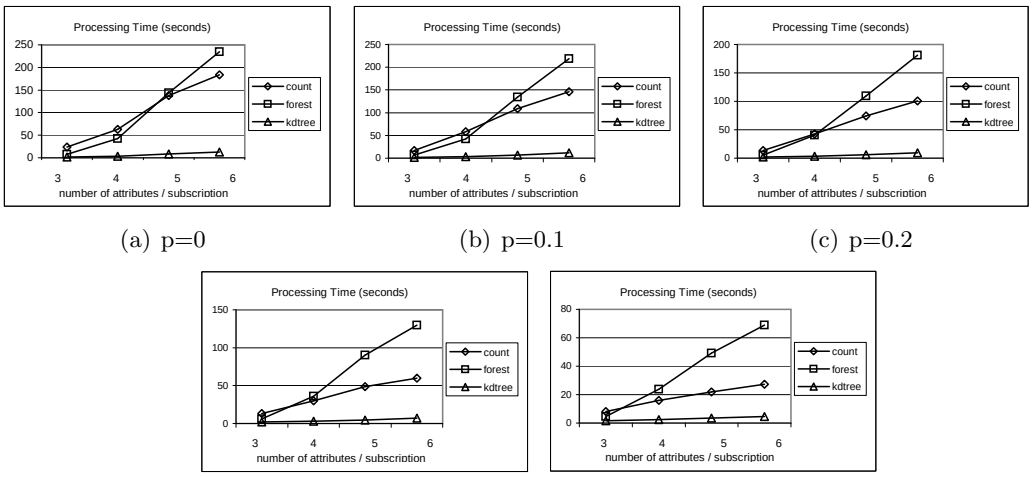
Second, a node in G_k will be disconnected from all its neighbors during unsubscribe. Section 4.4.2 shows that a larger k will increase both a node's in-degree and out-degree, which is reflected in Figure 4.3(f). The induced extra overhead further diminished the benefit gained in the checking time, as you may compare the values shown in Figure 4.3(e) and Figure 4.3(c).

In summary, by increasing the value of k, we slightly benefit the unsubscribe operation, yet severely penalize the subscribe operation. Since it turns out that the processing time is

dominated by the overhead of handling subscribes, it is beneficial to keep $k = 1$ to minimize the processing time.

4.5.4 Comparative study of covering data structures

In this section, we compare the performance of following covering data structures: the 2-layer framework, POSET-derived forest and Siena POSET. The purpose is to judge which structure yields the shortest time in processing a sequence of mixed subscribe and unsubscribe operations.



(a) p=0 (b) p=0.1 (c) p=0.2

(d) p=0.3 (e) p=0.4

Figure 4.4: 50000 (un)subscribe operations

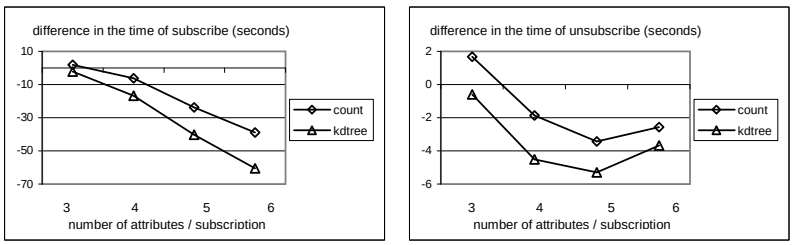
We choose an input size of 50000 operations and vary the values of $d(3 \leq d \leq 6)$ and $p(0 \leq p \leq 0.4)$ to measure the performance under different configurations. The results are plotted in Figure 4.4. The labels "count" and "kdtree" represent two separate instances of the 2-layer framework. They use the same graph G_1, but different subscription index.

The Siena POSET runs much slower than the other data structures under all the scenarios. For instance, it spent approximately 560 seconds in the scenario specified by $(n = 50000, d = 6, p = 0.4)$, while the time incurred by the other structures for the same scenario were less than 60 seconds as indicated in Figure 4.4(e). Because of the huge disparity, the POSET curves

don't fit easily into the graphs and are not drawn.

In Figure 4.4, the performance of the "counting" instance is close to that of POSET-derived forest, when the number of attributes in each subscription is no greater than 4. It outperforms POSET-derived forest if more attributes are introduced. In contrast, the results generated by the K-d tree instance show overwhelming advantage over the other two structures under all the scenarios. For example, in the scenario of $(n = 50000, d = 6, p = 0.2)$, the overhead of K-d tree is roughly 9% of counting and 5% of POSET-derived forest. In another scenario characterized by $(n = 50000, d = 6, p = 0.4)$, the cost of K-d tree is about 7% of POSET-derived forest and 17% of counting. The significant performance boost roots in the fact that K-d tree is more efficient than the counting algorithm in detecting covering for numeric subscriptions.

The results also demonstrate the flexibility of our modular approach. If regular subscription index is in use, we can improve the performance by using a simple graph G_1. If the same graph G_1 is equipped with a sophisticated index, then the processing overhead can be further reduced. Since the graph design and the building of subscription index are loosely-coupled in our framework, if someone comes up with better graph design or builds more efficient index in the future, we can easily integrate his contribution into our framework and make it a better solution to address the problem of subscription covering. However it is unclear how to realize the same type of adaptability in the monolithic POSET structures.



(a) (b)

Figure 4.5: 50000 (un)subscribe operations. Probability of unsubscribe is 0.4. The difference is relative to the POSET-derived forest

We can break down the processing time illustrated in Figure 4.4 into, (1) time spent on subscribe and (2) time spent on unsubscribe, to investigate why POSET-derived forest performs

worse on most inputs. Figure 4.5 plots the results obtained from a set of scenarios characterized by $(n = 50000, p = 0.4)$. The graphs show the difference between POSET-derived forest and the instances of the 2-layer framework in terms of the two variables.

The results indicate that POSET-derived forest runs slower in both subscribing and unsubscribing. But the difference exhibited in the processing time is largely attributed to the difference caused by the time of subscribe. Recollect the respective algorithms of adding a new subscription, the G_1 algorithm and POSET-derived forest share the common point that finding a single covering subscription suffices. However POSET-derived forest requires the returned subscription must be an *immediate predecessor* of the new subscription. It's this extra constraint that makes the insertion into POSET-derived forest more expensive. It is unclear how to find immediate predecessors quickly other than walking down the POSET-derived forest and examine nodes one-by-one. In contrast, we can easily assign the task of covering detection to a dedicated subscription index in the G_k algorithm and expedite the insertion process. The results show that our new graph design is not only efficient but also sufficient.

4.6 Approximate covering detection

Although subscription index is shown to be useful in accelerating covering detection, it remains a hard combinatorial problem. Its special case, the detection of covering for numeric subscriptions, can be formulated as multidimensional point dominance problem, for which no efficient worst-case solution exists.

On the other hand, we observe that in our publish-subscribe application, there is no need to search for covering subscriptions exhaustively every time. Covering is just an optimization. The system will continue to work correctly if covering relations go undetected. However, if routers continue to forward subscriptions that are covered, the system could soon degrade in performance. Thus, we have two extremes, neither of them very desirable – one is to ignore covering completely and the other is to follow it exactly all the time. We propose a middle ground, *approximate covering detection*. When a new subscription arrives, the set of existing subscriptions at a router is partially searched for a covering subscription, and if none is found,

then the new subscription is forwarded. We are able to precisely quantify the fraction of the space of covering subscriptions that is to be searched. Our main result is that this middle ground is quite attractive. *It is possible to search most of the solution space consisting of covering subscriptions at a fraction of the cost it takes to exhaustively search for covering subscriptions.*

In this section, we present one solution of approximate covering based on space filling curves. This method can be applied to numeric subscriptions. The roadmap is as follows: We first show the equivalency of covering detection to multidimensional point dominance in Section 4.6.1. Next we survey the related work in Section 4.6.2. Space filling curves are introduced in Section 4.6.3. In Section 4.6.4, we present some intuition as to why approximate point dominance may be cheaper than exhaustive point dominance, and then derive the upper bound on the cost of approximate point dominance. This is followed by an analysis of a lower bound on the cost of exhaustive point dominance in Section 4.7. We finally sketch our algorithm for approximate point dominance in Section 4.8.

4.6.1 Covering detection through point dominance

We consider a publish-subscribe system where each event has β numerical attributes, and each subscription is a conjunction of range constraints, with one constraint per attribute. An event can be treated as a point in β dimensional space, and a subscription can be modeled as a β dimensional rectangle that matches all messages whose corresponding points lie inside the rectangle.

Let S denote the set of subscriptions that have been registered at the router. Given a new subscription s, the problem of finding whether or not it is covered by an existing subscription in S is equivalent to the problem of finding an existing rectangle that encloses the incoming rectangle. We apply the following well-known transformation (Edelsbrunner and Overmars[18]) to convert this into an equivalent *point dominance* problem in 2β-dimensional space.

A β-dimensional rectangle (subscription) $s = ([\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_\beta, r_\beta])$ is transformed into a 2β-dimensional point $p(s) = (-\ell_1, r_1, -\ell_2, r_2, \ldots, -\ell_\beta, r_\beta)$. The following fact can be

easily verified : *For two β-dimensional subscriptions s_1 and s_2, s_1 covers s_2 iff every coordinate of $p(s_1)$ is no less than the corresponding coordinate of $p(s_2)$.* Note that the transformation goes both ways. It is shown [18] that 2β-dimensional point dominance can be reduced to the β-dimensional rectangle enclosure (or subscription covering) problem. Henceforth, we consider the following point dominance formulation of subscription covering.

Problem 1 (Point Dominance) *Index a set P of points in d dimensional space to answer the following query efficiently. Given a d dimensional point $x = (x_1, x_2, \ldots, x_d)$, report any point in P that lies in the region $([x_1, \infty], [x_2\infty], \ldots, [x_d, \infty])$. If there are no points in the region, then report "empty".*

Note that we use ∞ to denote the maximum value that can be taken by a coordinate along a dimension. This maximum may be different for different dimensions. Our formulation of approximate subscription covering is through the following relaxed version of point dominance, called *ϵ-approximate point dominance*, for a user specified ϵ.

Problem 2 (ϵ-Approximate Point Dominance) *Index a set P of points in d dimensional space to answer the following query efficiently. Given a user defined parameter $0 < \epsilon < 1$, and one d dimensional query point $x = (x_1, x_2, \ldots, x_d)$, search a subset of the region $([x_1, \infty], [x_2, \infty], \ldots, [x_d, \infty])$ whose volume is at least $(1-\epsilon)$ of the volume of the entire region. If any point was found in the search, return it, and return "empty" otherwise.*

For example, a 0.05-approximate point dominance query searches 95% of the volume of the region that contains points corresponding to covering subscriptions. The only time when it fails is the case when the query subscription is covered, but all covering subscriptions lie in the remaining 5% of the region that has not been searched. If subscriptions are well distributed over the universe, then an approximate point dominance search can be expected to find most existing covering relations between subscriptions.

For a point dominance query, let b_{max} and b_{min} denote the number of bits required to represent the longest and the shortest sides respectively, of the query rectangle. The aspect

ratio α of the query rectangle is defined as $\alpha = b_{max} - b_{min}$[1]. Informally, the aspect ratio is small (close to zero) when the sides of the query region are approximately the same length and large when they are of significantly different lengths.

We consider indexes for approximate point dominance based on the Z space filling curve. The Z curve has been used in a variety of indexing applications, including commercial data products such as Oracle [38, 22]. Other popularly used SFCs are the Hilbert curve [25] and the Gray code curve [20]. It has been observed [31] that the performance of the Z and Hilbert curves for many indexing applications are within a constant fraction of each other.

Our Contributions We introduce the notion of *approximate covering* to optimize subscription propagation in content-based publish-subscribe systems. Using the point dominance formulation of subscription covering, we show the following. *For point dominance queries where the aspect ratio of the query region is small, approximate point dominance is much cheaper than exhaustive point dominance.* More precisely,

1. The worst case time complexity of an ϵ-approximate point dominance query in d dimensions using the Z SFC is $O\left[\log \frac{d}{\epsilon} \cdot \left(2^{\alpha+1}\frac{d}{\epsilon}\right)^{d-1}\right]$

2. In contrast, the worst case time complexity of an exhaustive point dominance query using the Z SFC is $\Omega\left[\left(2^{\alpha-1}\ell\right)^{d-1}\right]$ where ℓ is the length of the shortest side of the query rectangle.

3. We present a simple algorithm for approximate covering detection based on the Z space filling curve.

Somewhat surprisingly, this shows that for a point dominance query with a small aspect ratio, the complexity of an ϵ-approximate query is *independent of the side lengths of the query region*, while the complexity of an exhaustive point dominance query increases as the $(d-1)$th power of the smallest side length of the query region. This implies that an ϵ-approximate

[1]In two dimensional space, the aspect ratio of a rectangle is traditionally defined as the ratio of the longer to the shorter side. Our definition of aspect ratio is approximately the logarithm (to base 2) of the traditional definition. This definition leads to a convenient statement of our results.

query (for a constant ϵ) is much cheaper than an exhaustive query. Further, we can expect that the benefits of approximate covering over exhaustive covering will be more pronounced as the query region gets larger. For query subscriptions with a small aspect ratio, approximate covering can yield most of the benefits of exhaustive covering at a small fraction of the cost, thus making a strong case for using approximate covering in optimizing content-based routing.

If however, the aspect ratio of the query rectangle was large, then the term 2^{α} will dominate the above expressions, and though approximate covering is still cheaper than exhaustive covering, the benefits will not be as much as the case of small aspect ratio. An extreme case in two dimensions is a $M \times 1$ rectangle, which is not efficiently handled by most popular SFCs.

4.6.2 Related work on the detection of subscription covering

From a worst-case time complexity perspective, the current best solution to point-dominance problem (see [44][Ch. 8], [63, 64]) over a set of n of points in d dimensions has a query time of $O(\log^{d-1} n)$, insertion and deletion times of $O(\log^{d} n)$. A serious limitation of this solution is the space complexity, which is $O(n \log^{d} n)$, making them impractical for use in a pub-sub system. For example, with 10^4 subscriptions each with 4 attributes, the space requirement is easily outside the capacity of the main memory.

Existing solutions to the problem of subscription covering [30, 61] do not provide any formal analysis of the performance. In a recent work, Ouksel *et.al.*[39] consider a relaxed notion of subscription covering and give a probabilistic algorithm for covering detection. The complexity is $O(nm)$, where n is the number of subscriptions and m is the number of attributes. To our knowledge, ours is the first algorithm for exact or approximate covering with a time complexity that is sublinear in the number of subscriptions being indexed.

Though there have been numerous applications of SFCs for indexing multidimensional data and corresponding experimental analysis, there has been relatively little work on a formal analysis of their performance. Moon *et al.*[31] present an analysis of the clustering properties of the Hilbert SFC. They show that given a query region which is a high dimensional rectangle, the average number of clusters of points inside the rectangle is proportional to the surface area of

the query rectangle. Their analysis considers exhaustive search while we consider approximate search. Tirthapura *et. al.*[60] show a formal analysis of space filling curves for parallel domain decomposition.

4.6.3 Space filling curves

We consider a d-dimensional universe $2^k \times 2^k \cdots \times 2^k$. Note that the number of dimensions d is twice the number of attributes in a subscription. Each element of this universe $p = (x_1, x_2, \ldots, x_d)$, where for all $i = 1 \ldots d$, $x_i \in [0, 2^k - 1]$, is called a *cell*. The SFC imposes a linear order on all 2^{kd} cells. Henceforth we use the term *cube* to refer to a cube in d dimensions and *rectangle* to refer to a rectangle in d dimensions.

Most SFCs used for indexing including the Z curve [32] and the Hilbert curve [25], utilize a recursive partitioning of the universe. The universe is first divided into 2^d cubes, each of side length 2^{k-1}, by bisecting along every dimension. Each resulting cube is recursively divided $k-1$ times until we are left with unit cubes. We use the term *standard cube* to refer to each intermediate cube resulting from this process. When the cube is recursively decomposed $\ell \leq k$ times, there are $2^{d\ell}$ standard cubes each containing $2^{d(k-\ell)}$ cells. Each such cube is referred to as a *standard cube at level* ℓ. Standard cubes at level k are the individual cells.

Lemma 2 *Let C and D be two standard cubes which are not equal to each other. Then either C contains D or D contains C or C and D are disjoint from each other.*

Proof: The recursive partitioning of the universe can be visualized as a tree whose root is the entire universe, and whose leaves are individual cells. Both C and D are nodes in this tree. Since $C \neq D$, the following are the only possibilities (1)C is an ancestor of D, or (2)D is an ancestor of C, or (3)C and D lie in different subtrees. In the first case, C contains D, in the second case D contains C and in the third case C and D are disjoint from each other. ■

Each standard cube at level $\ell \leq k$ is assigned a unique $d\ell$ bit number called its *key*, which defines its position in the total order. Different SFCs differ in the assignment of keys to different standard cubes at the same level. The input points are sorted according to the keys of the cells

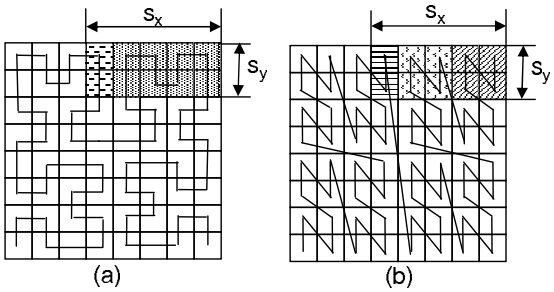


Figure 4.6: For the same $S_x \times S_y$ rectangle, there are (a) 2 runs for the Hilbert SFC and (b) 3 runs for the Z SFC

containing them, and stored in a one-dimensional data structure called the *SFC array* (note that the SFC array could be implemented using any dynamic unidimensional data structure such as a binary tree or a skip list).

A *run* is defined as a set of cells that are consecutively ordered by the SFC. For example, in Figure 4.6, for the rectangular region, there are three runs in the Z curve, and two runs in the Hilbert curve. All points belonging to a run appear as a contiguous segment in the SFC array. Accessing a run in the SFC requires two binary searches on the keys corresponding to the first and last cells in the run. Hence, an operation on a run, such as examining if a run is empty or not, is very efficient, whether the run is small or large. Hence, the performance of an SFC based query over a region depends on the minimum number of runs the region can be decomposed into. The following fact is true for the Z SFC, and for all SFCs that use a recursive partitioning of the universe. Informally, once an SFC enters a standard cube, it will leave the standard cube only after visiting every cell inside it.

Fact 1 *A standard d-cube is a single run.*

4.6.4 Upper bound for approximate point dominance

In this section, we derive an upper bound for the cost of approximate point dominance based on the Z SFC(Theorem 2). To launch a smooth progress, we first unveil the behind-the-scene intuition. Next we define the fraction of the solution space consisting of covering

subscriptions that is to be searched by approximate covering. Then we show a series of lemmas that lead to the final upper bound.

Intuition The performance of the space filling curve on a point dominance query, whether exhaustive or approximate, depends on how many runs the query region can be partitioned into. Accessing different runs costs the same, but the volume covered by different runs can be vastly different. As a result, the regions that are considered by the approximate and exhaustive point dominance queries may differ only slightly in terms of volume, but widely in terms of the number of runs required to cover them, and hence in the cost of processing.

For example, consider a universe indexed by the Z curve. Figure 4.7 shows two query regions, each corresponding to a different point dominance query. The first query region is a square of size 256×256, and the second query region is of size 257×257. For the first query, there is a single run that exactly equals the query region, and the cost of answering this query is very small. On the other hand, the number of runs required to exhaustively cover the second query region is 385, since we are forced to cover the periphery of the region using very short runs. In fact, it will be shown (Section 4.7) that the cost of exhaustively covering a d-dimensional rectangle is proportional to the surface area (the perimeter, in two dimensions) of the rectangle. However, for the second query region, one of the runs covers more than 99% of the query region, while most of the other smaller runs individually cover only 0.015% of the query region. If we only wanted a 0.01-approximate point dominance search, we would be done if we only examined the largest run, and ignored the rest. Thus an approximate point dominance search would be much faster than an exhaustive one for the second query.

Extremal rectangle The algorithm for approximate point dominance will select a subspace of the query region such that the volume of the subspace is at least $(1 - \epsilon)$ fraction of the volume of the query region, but the number of runs required to cover it is much smaller. We now describe the way in which this subspace is selected.

Given a point $p = (x_1, x_2, \ldots, x_d)$, the exhaustive point dominance asks for any point in the rectangle $([x_1, 2^k - 1], [x_2, 2^k - 1], \ldots, [x_d, 2^k - 1])$. We refer to such a rectangle as an

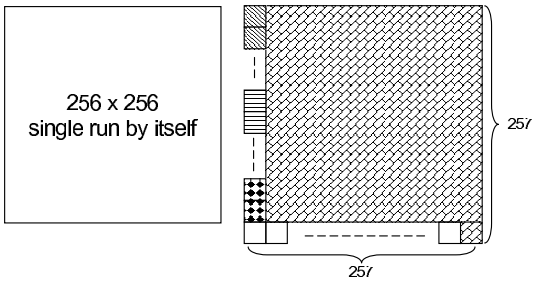


Figure 4.7: Two example point dominance queries for the Z curve. Standard cubes belonging to the same run are drawn using identical patterns

extremal rectangle, since one of its vertices is the point $(2^k - 1, 2^k - 1, \ldots, 2^k - 1)$. Note that an extremal rectangle can be completely specified through specifying its lengths along each dimension, since one of its vertices is fixed. Let $\ell = (\ell_1, \ell_2, \ldots, \ell_d)$ be a vector where for each $i = 1, \ldots, d$, $1 \leq \ell_i \leq 2^k$. We use $R(\ell)$ to denote an extremal rectangle, whose side lengths along dimensions $1, 2, \ldots, d$ are $\ell_1, \ell_2, \ldots, \ell_d$ respectively.

For a positive integer x, let $b(x)$ denote the number of bits in the binary representation of x, where the most significant bit is 1. For example, $b(9) = 4$. For positive integer x and $m < b(x)$, let $t(x, m)$ denote the integer formed by retaining the m most significant bits in x and setting the rest to zero. When the input is a vector, the operator $t()$ will be applied to each element in the vector. For example, $t(\ell, m) = (t(\ell_1, m), t(\ell_2, m), \cdots, t(\ell_d, m))$.

Given an initial query region $R(\ell)$, the approximate point dominance query considers a smaller extremal rectangle $R(t(\ell, m))$ that is completely contained within $R(\ell)$. For ease of notation, we use $R^m(\ell)$ to represent $R(t(\ell, m))$. The parameter m is chosen as a function of ϵ (the user desired coverage) such that the chosen rectangle covers at least $(1 - \epsilon)$ fraction of the volume of $R(\ell)$.

We now formally develop the upper bound. Before outlining the proof, we first explain some concepts.

Definition 5 *For rectangle T, $cubes(T)$ is defined as the minimum number of standard cubes into which T can be partitioned, and $runs(T)$ is defined as the minimum number of runs that make up all cells of T in the SFC array.*

Lemma 3 *For any rectangle T, $cubes(T) \geq runs(T)$.*

Proof: Consider a partitioning of T into $cubes(T)$ standard cubes. From Fact 1 each standard cube in such a partitioning corresponds to a single run. This immediately leads to a clustering of all cells that make up T in the SFC array into $cubes(T)$ runs. Thus it must be true that $runs(T) \leq cubes(T)$. ■

The worst-case cost of approximate point dominance is equal to $runs(R^m(\ell))$. According to Lemma 3, this is bounded by $cubes(R^m(\ell))$. So our strategy for the proof goes as follows. Lemma 5 shows that a greedy algorithm can partition an arbitrary region into a minimum number of standard cubes. Lemma 6 classifies the obtained standard cubes so that we are able to totalize $cubes(R^m(\ell))$ in Lemma 9. The final statement Theorem 2 can be easily extended from Lemma 9. Note that the proofs are very general so that they remain valid for other SFC, such as Hilbert curve, which is based on the recursive partitioning of the space.

As the first step, we decide what value of m produces the required space coverage. The following lemma shows that if we truncated each side length of $R(\ell)$ down to its $\log_2 \frac{2d}{\epsilon}$ most significant bits, then the volume of the resulting rectangle is within $(1-\epsilon)$ of the volume of $R(\ell)$. The proof is in the appendix.

Lemma 4 *Let* $0 < \epsilon < 1$. *If* $m \geq \log_2 \frac{2d}{\epsilon}$, *then* $\frac{vol(R^m(\ell))}{vol(R(\ell))} \geq 1 - \epsilon$

Proof: Let $\gamma = \frac{vol(R^m(\ell))}{vol(R(\ell))}$.

$$\gamma = \prod_{i=1}^{i=d} \frac{t(\ell_i, m)}{\ell_i}$$

$$\frac{t(\ell_i, m)}{\ell_i} \geq \frac{t(\ell_i, m)}{t(\ell_i, m) + 2^{b(\ell_i)-m}} = \frac{1}{1 + \frac{2^{b(\ell_i)-m}}{t(\ell_i, m)}} \geq \frac{1}{1 + \frac{2^{b(\ell_i)-m}}{2^{b(\ell_i)-1}}} = \frac{1}{1 + \frac{1}{2^{m-1}}}$$

Since $1 + x \leq e^x$ for all real x,

$$\frac{t(\ell_i, m)}{\ell_i} \geq e^{\frac{-1}{2^{m}-1}} = e^{\frac{-2}{2^m}}$$

Thus,

$$\gamma \geq \prod_{i=1}^{i=d} e^{\frac{-2}{2^m}} = e^{\frac{-2d}{2^m}} \geq 1 - \frac{2d}{2^m}$$

Since $m \geq \log_2 \frac{2d}{\epsilon}$, we have $\frac{2d}{2^m} \leq \epsilon$, and thus $\gamma \geq 1 - \epsilon$ ■

Next we consider the partitioning of a given extremal rectangle into standard cubes. It is always trivially possible to partition any rectangle into a set of standard cubes by breaking it up into individual cells (note that each cell is a standard cube). We now describe a greedy algorithm that outputs the partition of any region (which is not necessarily a rectangle) into a *minimum* number of standard cubes. Suppose it is required to partition a region R into standard cubes. The greedy algorithm works as follows. *If R is empty, then the algorithm outputs nothing, and exits. Otherwise, the largest standard cube that fits within R, say C, is chosen and output. The algorithm is then recursively applied on input $R - C$.*

Lemma 5 *The greedy algorithm, when applied to region R, produces an optimal partition of R into a minimum number of standard cubes.*

Proof: Proof by induction on $vol(R)$, the number of cells in R. For the base case, where $vol(R) = 0$, the algorithm is obviously correct. For the inductive case, assume that for every region R such that $vol(R) < \kappa$, the greedy algorithm decomposes R into the minimum number of standard cubes.

Consider a region R such that $vol(R) = \kappa$. Let C be the largest standard cube completely contained within R. Let D denote a partition of R into the minimum number of standard cubes. We use proof by contradiction to show that E must contain C as an element. Suppose $C \notin E$. Let $E' \subset E$ represent the set of standard cubes within E that contain cells that makeup C. Clearly E' is non-empty. Since each element of E' intersects C, and none of them can equal C, it must be true from Lemma 2 that every element of E' is fully contained within C. This yields a decomposition of R into fewer number of standard cubes than E by replacing E' with a single cube C, contradicting the optimality of E. Thus E must contain C, and the first step of the greedy algorithm is proved correct. The remaining region $(R - C)$ has fewer than κ cells, since C is non-empty. By the inductive hypothesis, the greedy algorithm generates an optimal decomposition of $(R - C)$, and the proof is complete. ■

Let D represent the set of standard cubes resulting from a decomposition of $R(\ell)$ into standard cubes based on the greedy algorithm. Let D_i represent the subset of D consisting

of standard cubes of side length 2^i. For integer x, let x_j denote the jth bit in the binary representation of x. For integer x, let $S_i(x) = \sum_{j=i}^{j=b(x)-1} x_j 2^j$, i.e. $S_i(x)$ is the result of choosing only the most significant bits in the binary representation of x starting from x_i onwards. When the input is a vector, the operator $S_i()$ will be applied to each element in the vector. Thus $S_i(\ell) = (S_i(\ell_1), S_i(\ell_2), \cdots, S_i(\ell_d))$. Let $\ell_{i,j} = \ell_{ij}$, i.e. the jth bit of ℓ_i. For $j = 1, \ldots, d$, indicator variable O_j is defined as follows. $O_j = 0$ if $\ell_{i_j} = 0$ for all $i = 1, \ldots, d$; $O_j = 1$ if $\ell_{i,j} = 1$ for some i, $1 \leq i \leq d$. Assume w.l.o.g. $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_d$. The following lemma gives a precise characterization of the type and location of the standard cubes resulting from an optimal partition of $R(\ell)$.

Lemma 6 *For $b(\ell_1) \leq i \leq b(\ell_d) - 1$, D_i is empty. For $0 \leq i \leq b(\ell_1) - 1$*

1. *D_i is non-empty if and only if $O_i = 1$.*
2. *The region occupied by $\cup_{j=i}^{j=b(\ell_1)-1} D_j$ is the extremal rectangle $R(S_i(\ell))$.*

Proof: First we prove by contradiction that D_i is empty for $b(\ell_1) \leq i \leq b(\ell_d) - 1$. Suppose D_j is non-empty for some $j, b(\ell_1) \leq j \leq b(\ell_d) - 1$. Let $C_j \in D_j$ be a standard cube of side length 2^j. The projection of C_j on the first dimension is a line segment of length $2^j \geq 2^{b(\ell_1)}$. But ℓ_1 is at most $2^{b(\ell_1)} - 1$ since it is a $b(\ell_1)$ bit number, leading to a contradiction. Thus D_i must be empty for $b(\ell_1) \leq i \leq b(\ell_d) - 1$.

For the rest of i, $b(\ell_1) - 1 \geq i \geq 0$, we use proof by reverse induction on i, starting from $i = b(\ell_1) - 1$ and going down to $i = 0$.

We first consider the base case: $i = b(\ell_1) - 1$. We have $O_{b(\ell_1)-1} = 1$, because $\ell_{1,b(\ell_1)-1} = 1$. Since $S_{b(\ell_1)-1}(\ell_j)$ is non-zero and no more than ℓ_j for $1 \leq j \leq d$, it must be true that the extremal rectangle $R(S_{b(\ell_1)-1}(\ell))$ is not empty and is fully contained inside $R(\ell)$. Moreover, $S_{b(\ell_1)-1}(\ell_j)$ is divisible by $2^{b(\ell_1)-1}$ for $1 \leq j \leq d$. So $R(S_{b(\ell_1)-1}(\ell))$ can be decomposed into the union of standard cubes of side length $2^{b(\ell_1)-1}$. We know from the above that D_i is empty for $b(\ell_d) - 1 \geq i \geq b(\ell_1)$. This indicates that the largest standard cube which can fit into $R(\ell)$ has a side length of $2^{b(\ell_1)-1}$. Such standard cubes will be chosen by the greedy algorithm if they exist. Since $R(S_{b(\ell_1)-1}(\ell))$ is not empty, we know $D_{b(\ell_1)-1}$ is also non-empty.

Finally we claim there is no standard cube of side length $2^{b(\ell_1)-1}$ in the region $R(\ell) - R(S_{b(\ell_1)-1}(\ell))$. We use proof by contradiction. Suppose the specified region contains one standard cube $C \in D_{b(\ell_1)-1}$. Since two standard cubes cannot intersect each other (Lemma 2), there can be no intersection between C and $R(S_{b(\ell_1)-1}(\ell))$. Under this condition, there must exist at least one dimension onto which the projections of C and $R(S_{b(\ell_1)-1}(\ell))$ are disjoint. Proof by contradiction. Suppose the projections of C and $R(S_{b(\ell_1)-1}(\ell))$ overlap on every dimension. Let $[l_i, r_i]$ denote the overlapping segment between the two projections along the ith dimension. As a result, $[l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_d, r_d]$ forms a rectangle which is shared by both rectangles, which contradicts the fact that they don't intersect. Therefore the projections of C and $R(S_{b(\ell_1)-1}(\ell))$ must be disjoint along some dimension.

Assume w.l.o.g. that the projections of C and $R(S_{b(\ell_1)-1}(\ell))$ are disjoint along dimension 1. The combined lengths of their projections along dimension 1 is $S_{b(\ell_1)-1}(\ell_1) + 2^{b(\ell_1)-1} = 2^{b(\ell_1)}$. However ℓ_1 can be no more than $2^{b(\ell_1)} - 1$. We reached a contradiction. Therefore the region $R(\ell) - R(S_{b(\ell_1)-1}(\ell))$ can not contain C. As a consequence, the region occupied by $D_{b(\ell_1)-1}$ corresponds to the extremal rectangle $R(S_{b(\ell_1)-1}(\ell))$. This proves the base case.

For the inductive case, assume that the theorem remains true for every i, $b(\ell_1) - 1 \geq i \geq \kappa + 1$. We now consider the case $i = \kappa$. We study the following two cases.

Case I: $O_\kappa = 0$. We prove by contradiction that D_κ must be empty. Suppose D_κ is not empty and $C_\kappa \in D_\kappa$ be a standard cube of side length 2^κ. The inductive hypothesis tells us $\cup_{j=\kappa+1}^{j=b(\ell_1)-1} D_j$ forms an extremal rectangle $R(S_{\kappa+1}(\ell))$. We are able to show, by following the analysis for the base case, the remaining region $R(\ell) - R(S_{\kappa+1}(\ell))$ can not contain C_κ. This means D_κ must be empty. Since D_κ is empty, the inductive hypothesis can be directly extended to show that standard cubes in D with a side length of 2^κ or higher form an extremal rectangle $R(S_\kappa(\ell))$.

Case II: $O_\kappa = 1$. We now consider the case $O_\kappa = 1$. Consider the extremal rectangles $R(S_\kappa(\ell))$ and $R(S_{\kappa+1}(\ell))$. Since $S_{\kappa+1}(\ell_i) \leq S_\kappa(\ell_i) \leq \ell_i$, it must be true that $R(S_{\kappa+1}(\ell))$ is fully contained in $R(S_\kappa(\ell))$, which is in turn fully contained within $R(\ell)$. Since $S_\kappa(\ell_i)$ is divisible by 2^κ for $1 \leq i \leq d$, so $R(S_\kappa(\ell))$ can be decomposed into a union of standard cubes of side

length 2^κ. Similarly, it can also be shown that $R(S_{\kappa+1}(\ell))$ can be decomposed into a union of standard cubes of side length 2^κ. Since $R(S_{\kappa+1}(\ell))$ is fully contained within $R(S_\kappa(\ell))$, the set of standard cubes in $R(S_{\kappa+1}(\ell))$ is a subset of the standard cubes in $R(S_\kappa(\ell))$. Thus the region $R(S_\kappa(\ell))$ - $R(S_{\kappa+1}(\ell))$ can also be decomposed into multiple standard cubes of side length 2^κ. Furthermore because of $O_\kappa = 1$, we must have $\ell_{i,\kappa} = 1$ for some i and in turn $S_\kappa(\ell_i) - S_{\kappa+1}(\ell_i) > 0$. So the region $R(S_\kappa(\ell))$ - $R(S_{\kappa+1}(\ell))$ is not empty. We can follow an analysis that is similar to the base case to show that the remaining region $R(\ell) - R(S_\kappa(\ell))$ does not contain any standard cube of side length 2^κ. This completes the proof that D_κ is non-empty and the union of standard cubes of side length 2^κ or higher in D form the extremal rectangle $R(S_\kappa(\ell))$. ■

Lemma 7 *For every $i = 0, \ldots, b(\ell_1) - 1$, we have the following. If $O_i = 0$, then $N_i = 0$. If $O_i = 1$, then*

$$N_i = \frac{\prod_{j=1}^{j=d} S_i(\ell_j) - \prod_{j=1}^{j=d} S_{i+1}(\ell_j)}{2^{id}}$$

Proof: From Lemma 6, we know that if $O_i = 0$, then $N_i = 0$. If $O_i = 1$, then all standard cubes in D_i lie in the region $R(S_i(\ell)) - R(S_{i+1}(\ell))$. The expression for N_i can be derived through dividing the difference in the volumes of the two rectangles by the volume of a standard cube of side length 2^i. ■

Lemma 8 *cubes$(R^m(\ell))$ is maximized iff (1) $\ell_{j,x} = 1$ for $x \in [b(\ell_j) - m, b(\ell_j) - 1]$ and $j \in [1, d]$ and (2) $b(\ell_j) = b(\ell_d)$ for $1 < j < d$.*

Proof: From Lemma 6, we have $cubes(R^k(\ell)) = \sum_{i=b(\ell_1)-k}^{i=b(\ell_1)-1} N_i$. Using Lemma 7,

$$
\begin{aligned}
N_i &= \frac{1}{(2^i)^d}\left(\prod_{j=1}^{j=d} S_i(\ell_j) - \prod_{j=1}^{j=d} S_{i+1}(\ell_j)\right) \\
&= \frac{1}{(2^i)^d}\left[\prod_{j=1}^{j=d}(S_{i+1}(\ell_j) + \ell_{j,i}) - \prod_{j=1}^{j=d} S_{i+1}(\ell_j)\right] \\
&= \frac{1}{(2^i)^d}\left[\prod_{\substack{j=1\\ j\neq 1}}^{j=d} S_{i+1}(\ell_j)\cdot \ell_{1,i} + \cdots + \prod_{\substack{j=1\\ j\neq 1,2}}^{j=d} S_{i+1}(\ell_j)\cdot \prod_{j=1}^{j=2} \ell_{j,i} + \cdots + S_{i+1}(\ell_1)\cdot \prod_{j=2}^{j=d} \ell_{j,i} + \cdots + 1\right]
\end{aligned}
$$

It can be verified that each individual component in the above expression is maximized when $(1)\ell_{j,x} = 1$ for $x \in [i, b(\ell_j) - 1]$ and $j \in [1, d]$ $(2)b(\ell_j) = b(\ell_d)$ for $1 < j < d$. Thus, N_i is maximized under the same conditions. Since this is true for all i ranging from $b(\ell_1) - k$ till $b(\ell_1) - 1$, the lemma follows. ■

Recall that $\alpha = b(\ell_d) - b(\ell_1)$ is the aspect ratio of the rectangle, and $0 < \epsilon < 1$ is a user parameter indicating the desired coverage of the approximate query.

Lemma 9 $cubes(R^m(\ell)) < m \cdot [2^\alpha(2^m - 1)]^{d-1}$

Proof: We consider two cases: $(1)k < \alpha$ and $(2)k \geq \alpha$. To find an upper bound on $cubes(R^k(\ell))$, we assume vector ℓ satisfies the conditions specified by Lemma 8.

Case 1: $k < \alpha$.

According to Lemma 7, we have for $i \in [b(\ell_1) - k, b(\ell_1) - 1]$,

$$N_i = \prod_{j=2}^{j=d} \frac{S_i(\ell_j)}{2^i} \cdot \frac{S_i(\ell_1)}{2^i} - \prod_{j=2}^{j=d} \frac{S_{i+1}(\ell_j)}{2^i} \cdot \frac{S_{i+1}(\ell_1)}{2^i}$$

According to Lemma 8, we have $\ell_{j,x} = 1$ for $x \in [b(\ell_d) - k, b(\ell_d) - 1]$ and $1 < j \leq d$.

Since $k < \alpha$, it must be true that $b(\ell_d) - k > b(\ell_1) > i$. This leads to $S_i(\ell_j) = S_{b(\ell_d)-k}(\ell_d)$ for $1 < j \leq d$.

$$N_i = \left(\frac{S_{b(\ell_d)-k}(\ell_d)}{2^i}\right)^{d-1} \cdot \left(\frac{S_i(\ell_1)}{2^i} - \frac{S_{i+1}(\ell_1)}{2^i}\right) = \left(\sum_{x=b(\ell_d)-k}^{x=b(\ell_d)-1} 2^{x-i}\right)^{d-1}$$
$$= \left(2^{b(\ell_d)-i} - 2^{b(\ell_d)-k-i}\right)^{d-1} = \left[2^{b(\ell_d)-i} \cdot \left(1 - \frac{1}{2^k}\right)\right]^{d-1}$$
$$< \left[2^{\alpha+k} \cdot \left(1 - \frac{1}{2^k}\right)\right]^{d-1} < \left[2^{\alpha}(2^k - 1)\right]^{d-1}$$

$$cubes(R^k(\ell)) = \sum_{i=b(\ell_1)-k}^{i=b(\ell_1)-1} N_i < \sum_{i=b(\ell_1)-k}^{i=b(\ell_1)-1} \left[2^{\alpha}(2^k - 1)\right]^{d-1} = k \cdot \left[2^{\alpha}(2^k - 1)\right]^{d-1}$$

Case 2: $k \geq \alpha$.

According to Lemma 7, we have for $i \in [b(\ell_1) - k, b(\ell_1) - 1]$,

$$N_i = \prod_{j=2}^{j=d} \frac{S_i(\ell_j)}{2^i} \cdot \frac{S_i(\ell_1)}{2^i} - \prod_{j=2}^{j=d} \frac{S_{i+1}(\ell_j)}{2^i} \cdot \frac{S_{i+1}(\ell_1)}{2^i}$$

We further split the derivation into two subcases.

Case 2.1 $i \in [b(\ell_d) - k, b(\ell_1) - 1]$.

According to Lemma 8, we have $\ell_{j,x} = 1$ for $x \in [b(\ell_d) - k, b(\ell_d) - 1]$ and $1 < j \leq d$.

Since $i \geq b(\ell_d) - k$, we further get $\ell_{j,x} = 1$ for $x \in [i, b(\ell_d) - 1]$ and $1 < j \leq d$. This leads to $S_i(\ell_j) = S_i(\ell_d)$ for $1 < j \leq d$.

$$N_i = \left(\frac{S_i(\ell_d)}{2^i}\right)^{d-1} \cdot \frac{S_i(\ell_1)}{2^i} - \left(\frac{S_{i+1}(\ell_d)}{2^i}\right)^{d-1} \cdot \frac{S_{i+1}(\ell_1)}{2^i}$$

Note that

$$\frac{S_i(\ell_d)}{2^i} = \frac{S_{i+1}(\ell_d)}{2^i} + 1 \ , \frac{S_i(\ell_1)}{2^i} = \frac{S_{i+1}(\ell_1)}{2^i} + 1$$

We define $X = \frac{S_i(\ell_d)}{2^i}$ and $Y = \frac{S_i(\ell_1)}{2^i}$.

$$X = \frac{S_i(\ell_d)}{2^i} = \sum_{x=i}^{x=b(\ell_d)-1} 2^{x-i} = 2^{b(\ell_d)-i} - 1$$

Similarly $Y = 2^{b(\ell_1)-i} - 1$. We have $X \approx 2^{\alpha} \cdot Y$.

$$
\begin{aligned}
N_i &= X^{d-1} \cdot Y - (X-1)^{d-1} \cdot (Y-1) \\
&= (X^{d-1} - (X-1)^{d-1}) \cdot Y + (X-1)^{d-1} \\
&= [X^{d-2} + X^{d-3} \cdot (X-1) + \cdots + (X-1)^{d-2}] \cdot Y + (X-1)^{d-1} \\
&= X^{d-2} \cdot \left[1 + \frac{X-1}{X} + \cdots + \left(\frac{X-1}{X}\right)^{d-2}\right] \cdot Y + (X-1)^{d-1} \\
&< X^{d-2} \cdot (d-1) \cdot Y + (X-1)^{d-1} \\
&< X^{d-2} \cdot [Y(d-1) + X]
\end{aligned}
$$

If we assume $2^{\alpha} > (d-1)$, we have $[Y(d-1) + X] < 2X$.

$$N_i < 2 \cdot X^{d-1} = 2 \cdot (2^{b(\ell_d)-i} - 1)^{d-1} < 2 \cdot (2^k - 1)^{d-1}$$

Case 2.2: $i \in [b(\ell_1) - k, b(\ell_d) - k)$.

The derivation is similar to Case 1. We have $N_i < \left[2^{\alpha}(2^k - 1)\right]^{d-1}$

By combining case 2.1 and case 2.2, we have

$$
\begin{aligned}
cubes(R^k(\ell)) &< \sum_{i=b(\ell_d)-k}^{i=b(\ell_1)-1} 2 \cdot (2^k - 1)^{d-1} + \sum_{i=b(\ell_1)-k}^{i=b(\ell_d)-k-1} [2^{\alpha}(2^k - 1)]^{d-1} \\
&= 2(k - \alpha) \cdot (2^k - 1)^{d-1} + \alpha \cdot [2^{\alpha}(2^k - 1)]^{d-1} \\
&< (\alpha \cdot 2^{\alpha(d-1)} + 2k) \cdot (2^k - 1)^{d-1} \\
&< (k \cdot 2^{\alpha(d-1)} + 2k) \cdot (2^k - 1)^{d-1} \\
&= k \cdot (2^{\alpha(d-1)} + 2) \cdot (2^k - 1)^{d-1} \\
&\approx k \cdot \left[2^{\alpha}(2^k - 1)\right]^{d-1}
\end{aligned}
$$

■

Theorem 2 *For any SFC that is based on a recursive partitioning of the universe (such as*

the Z curve and the Hilbert curve), the cost of an ϵ-approximate point dominance query is $O(\log \frac{d}{\epsilon} \cdot \left(2^{\alpha+1}\frac{d}{\epsilon}\right)^{d-1})$

Proof: Using Lemma 9 and since $runs(R^m(\ell)) \leq cubes(R^m(\ell))$ (Lemma 3), we have $runs(R^m(\ell)) < m \cdot [2^\alpha(2^m - 1)]^{d-1}$. From Lemma 4, if we choose $m = \log_2 \frac{2d}{\epsilon}$ we are guaranteed that $R^m(\ell)$ covers at least $(1-\epsilon)$ fraction of the volume of $R(\ell)$. Thus the number of runs that have to be accessed for an ϵ-approximate point-dominance query is no more than $\log_2 \frac{2d}{\epsilon} \cdot \left[2^\alpha \left(\frac{2d}{\epsilon} - 1\right)\right]^{d-1}$, which yields the desired upper bound. ■

4.7 Lower bound for exhaustive point dominance

In this section, we prove a lower bound on the worst-case cost of exhaustive point dominance using Z-curve (Theorem 3). The worst-case cost is equal to $runs(R(\ell))$. However, estimating the size of $runs(R(\ell))$ is much harder than estimating the size of $cubes(R(\ell))$, because there is no simple characterization for runs like the "greedy" characterization for cubes.

Our strategy for proving a lower bound is as follows. We construct an extremal rectangle $R(\ell)$ such that for a large subset of the standard cubes resulting from a greedy (optimal) partition of $R(\ell)$, no two cubes in the subset can belong to the same run in the Z curve. Thus $runs(R(\ell))$ is no less than the size of this subset.

Let γ be an integer in $(0, k-\alpha]$. For a particular α, we consider the following extremal rectangle $R(\ell)$: (1) $\ell_d = 2^\gamma - 1$ and (2) $b(\ell_i) = \gamma + \alpha$ for $i \in [1, d)$. We examine a smaller rectangle R_0 contained inside $R(\ell)$. R_0 is constructed by selecting the least significant bit from ℓ_d and the most significant bit from ℓ_i for $i \in [1, d)$. Thus, the side length of R_0 along dimension d is 1, and the side length of R_0 along all other dimensions is $2^{b(\ell_1)-1}$. When the greedy partition is applied, R_0 is filled with standard cubes with a side length of 1, since its shortest side has length 1.

The position of each standard cube can be specified by its coordinates. For example, square "a" in Figure 4.10(c) has coordinates $(010, 011)$. The Z-curve computes the key of a standard cube by interleaving the bits representing its coordinates, starting from dimension 1, and then proceeding to higher dimensions. For instance, the key of square "a" is $(001101)_2 = 13$. Based

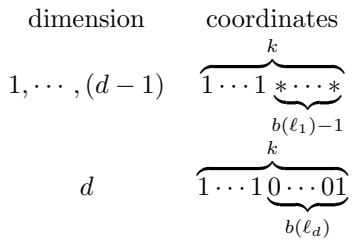


Figure 4.8: Coordinates of a cell in R_0. Symbol $*$ can be either 1 or 0. The size of the universe along each dimension is 2^k.

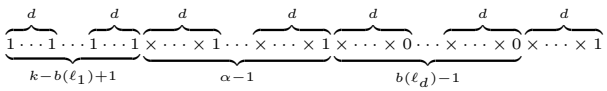


Figure 4.9: Key of a cell in R_0

on the way we construct R_0, we claim that the coordinates of each standard cube in R_0 exhibits the pattern shown in Figure 4.8. It then follows that the key of a cube in R_0 must satisfy the pattern shown in Figure 4.7. Lemma 10 uses this derived pattern to prove that no two standard cubes in R_0 belong to the same run on the Z-curve.

Lemma 10 *No two standard cubes in R_0 belong to the same run on the Z space filling curve.*

Proof: Every standard cube in rectangle R_0 must be a cell, since one side of R_0 has unit length. Let S denote the keys of all cells in R_0. We show that no pair of keys in S belong to the same run in the Z SFC. Consider two keys $s_1, s_2 \in S$. Assume w.l.o.g. $s_1 > s_2$. Since both s_1 and s_2 are odd numbers (from Figure 4.7), the difference between them is at least 2, implying that they cannot be adjacent in the SFC.

Consider the cell c corresponding to the key $s_1 - 1$. Cell c must be outside the query rectangle $R(\ell)$, for the following reason. The key of $s_1 - 1$ exhibits the same pattern as shown in Figure 4.7, except that the least significant bit is 0. By unwinding this key to retrieve the coordinates of the cell, we find that the dth dimension coordinate of c must be a k bit string consisting of $k - b(\ell_d)$ ones followed by $b(\ell_d)$ zeros. The projection of rectangle $R(\ell)$ along dimension d is a line segment of length ℓ_d whose right endpoint is $2^k - 1$ (a bit string of k ones) and left endpoint is $2^k - 1 - \ell_d$ (a bit string of $k - b(\ell_d)$ ones followed by $b(\ell_d) - 1$ zeroes

followed by a one). Clearly, the dth dimension coordinate of c cannot lie in this range, and hence c cannot belong to $R(\ell)$.

We have shown that in the linear ordering produced by the Z SFC, between any two standard cubes in R_0, there must be a cell which does not belong to $R(\ell)$. Since we are interested in partitioning only the cells of $R(\ell)$ into runs, without including any cell outside $R(\ell)$, no two standard cubes in R_0 can belong to the same run for the Z SFC. ■

Theorem 3 *For any integer $0 \leq \alpha < k$, there exists an extremal rectangle $R(\ell)$ whose aspect ratio is α and the cost of an exhaustive search of $R(\ell)$ using the Z space filling curve is $\Omega\left[\left(2^{\alpha-1} \cdot \ell_d\right)^{d-1}\right]$*

Proof: The cost of an exhaustive search is lower bounded by $runs(R_0)$, since all cells in R_0 have to be examined. From Lemma 10 we know $runs(R_0) = cubes(R_0)$. Since each standard cube in R_0 is a cell, $cubes(R_0) = vol(R_0)$

$$vol(R_0) = \prod_{j=1}^{j=d-1} 2^{b(\ell_j)-1} = \left(\frac{2^{b(\ell_d)} \cdot 2^{\alpha}}{2}\right)^{d-1} = \left(\frac{2^{\alpha} \cdot \ell_d}{2}\right)^{d-1}$$

■

4.8 Algorithm for approximate point dominance

In this section, we sketch an algorithm for approximate point dominance based on the Z SFC. The only data structure to maintain is the SFC array, which sorts input points according to their positions on the Z curve. It is easy to maintain this sorted order, while allowing frequent additions and deletions of points, by using a dynamic ordered data structure such as a balanced binary tree.

Given a point dominance query, our algorithm follows the greedy approach to partition the query region into a minimum number of standard cubes. It then searches these cubes in the SFC array for covering subscriptions, in the descending order of their volumes. Meanwhile it keeps track of the ratio of the volume searched to the volume of the query region. The search terminates when either a covering subscription is found or this ratio exceeds $1 - \epsilon$.

The major operation in the above algorithm is to compute the keys (defined in Section 4.6.3) of the standard cubes produced by the greedy decomposition – these keys are required by the search in the SFC array.

Let $R(\ell)$ denote the extremal rectangle to be searched for a point dominance query, where $\ell = (\ell_1, \ell_2, \ldots, \ell_d)$. Recall that D_i (defined in Section 4.6.4) is the set of all the standard cubes resulting from the greedy decomposition, whose side length is 2^i. It is sufficient to demonstrate how to compute the keys of all standard cubes within a particular D_i. We use the following two-stage algorithm. In the first stage, we decompose the space occupied by D_i into disjoint rectangles with the following two properties: (1) The side length of the rectangle along every dimension must be a multiple of 2^i so that the entire rectangle is a union of standard cubes in D_i. (2) There exists at least one dimension along which the side length of the rectangle is exactly 2^i. In the second stage, we identify the standard cubes within each such rectangle and compute their keys.

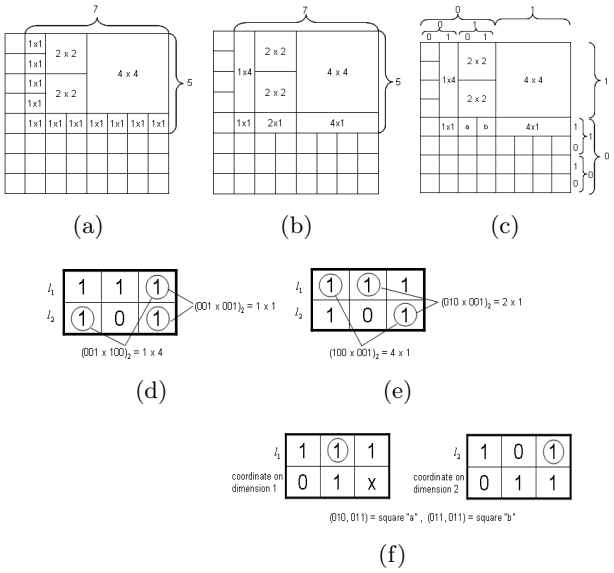


Figure 4.10: Computation of the keys of standard cubes resulting from a greedy decomposition. Symbol $\times$ can be either 1 or 0

Consider such a rectangle r in D_j that satisfies the above two properties. To decide r's side length on dimension $j (j = 1 \ldots d)$, we only need to consider the non-zero bits in ℓ_j, whose position in ℓ_j is at least i (the position of a bit b in a binary number is defined as the

number of bits to the right of b in the binary number – note that the least significant bit is at position 0). Let $P = (P_1, P_2, \ldots, P_d)$ be a vector defined as follows. For $j = 1 \ldots d$, P_j is the position of a non-zero bit in ℓ_j such that $P_j \geq i$. It can be verified that each unique instance of P corresponds to a different rectangle lying in the space occupied by D_i, which satisfies the properties listed in the previous paragraph. For example, all the 1×1 squares in Figure 4.10(a) constitute the space occupied by D_0. We can divide it into four rectangles as shown in Figure 4.10(b). The corresponding instances of P are shown in Figures 4.10(d) and 4.10(e).

Consider the rectangle represented by a vector P. The next step is to compute the keys of standard cubes contained within P. We create another vector $Q = (Q_1, Q_2, \ldots, Q_d)$ to store the coordinates of a standard cube within P, where Q_j is the coordinate of the standard cube along dimension j. We know that standard cubes in D_i result from $(k - i)$ rounds of recursive partitioning of the universe. Thus we can use $(k - i)$ bits to represent Q_j, for $j = 1 \ldots d$. Let $Q_{j,y}$ be the bit of Q_j at position y. We can compute Q_j from P_j and ℓ_j in the following way:

$$\begin{aligned} Q_{j,y-i} &= \neg \ell_{x,y}, && \text{for } y \in (P_x, k-1] \\ Q_{j,y-i} &= \ell_{x,y}, && \text{for } y = P_x \\ Q_{j,y-i} &= \text{either 0 or 1}, && \text{for } y \in [i, P_x) \end{aligned} \tag{4.5}$$

For example, the rectangle "2×1" in Figure 4.10(b) consists of two standard squares "a" and "b" as shown in Figure 4.10(c). We get the rectangle by selecting the first bit from ℓ_1 and the rightmost bit from ℓ_2. Figure 4.10(f) shows how Equation (4.5) can be applied to compute the coordinates of "a" and "b". Once the coordinates of a cube are available, we can get its key by interleaving the bits, and the standard squares can be searched in the SFC array.

Next we present the algorithm's pseudocode. Due to the typesetting problem, we split the algorithm into three pieces. Algorithm 4.3 is the main routine, which calls a recursive function *EnumRectangles*. The pseudo code for this routine is listed in Algorithm 4.5. Inside *EnumRectangles*, we call another recursive function *CompKeys*, whose pseudo code is presented in Algorithm 4.4.

Algorithm 4.3 Compute the keys of standard cubes in D_i

```
1: create an empty vector P of size d
2: for j = 1 to d do
3:    if ℓ_{j,i} == 1 then
4:       EnumRectangles(P,j,1)
```

Algorithm 4.4 CompKeys(vector P, vector Q, int t)

Synopsis: compute the keys of standard cubes contained inside the rectangle denoted by P

```
1: use Equation 4.5 to compute Q_t based on P_t and ℓ_t.
2: for every possible instance of Q_t do
3:    if t == d then
4:       generate the key of a standard cube by interleaving the bits in Q
5:    else
6:       CompKeys(P,Q,t + 1)
```

Algorithm 4.5 EnumRectangles(vector P, int s, int t)

Synopsis: enumerate the rectangles within the space occupied by D_i, whose side length along dimension s is 2^i

```
1: if t > s then
2:    for j = b(ℓ_t) − 1 downto i do
3:       if ℓ_{t,j} == 1 then
4:          P_t = j
5:          if t == d then
6:             create an empty vector Q of size d
7:             CompKeys(P,Q,1)
8:          else
9:             EnumRectangles(P,s,t + 1)
10: else if t < s then
11:    for j = b(ℓ_t) − 1 downto i + 1 do
12:       if ℓ_{t,j} == 1 then
13:          P_t = j
14:          if t == d then
15:             create an empty vector Q of size d
16:             CompKeys(P,Q,1)
17:          else
18:             EnumRectangles(P,s,t + 1)
19: else
20:    P_t = i
21:    if t == d then
22:       create an empty vector Q of size d
23:       CompKeys(P,Q,1)
24:    else
25:       EnumRectangles(P,s,t + 1)
```

CHAPTER 5. Fault tolerant publish/subscribe

Like any other large-scale distributed computing system, faults are common in a publish/subscribe network. They can take different forms. For example, network links may go down and destroy the network connectivity. Subscribers in each partition are then separated from the rest of the network. They can no longer receive interested events published outside the partition. It is also possible that a subscribe/unsubscribe message in transition could be dropped and lead to the inconsistency among routing tables. The presence of incorrect routing information will misguide the event traffic. Consequently subscribers may receive events they are not interested in or even worse lose interested events. Other transient faults include, but not limited to, arbitrary sequences of process crashes and subsequent recoveries, arbitrary perturbations of the memory etc.

In order to build a reliable publish/subscribe system, one approach is to enumerate all possible faults that could occur and take actions to correct each of them. However due to a wide variety of possible faults, implementing fault masking is at least expensive if not impossible. An alternative solution is *self-stabilization*, a notion developed by Dijkstra [17] in 1974. Informally a system is self-stabilizing, if starting from an arbitrary initial state (perhaps faulty), it is guaranteed to reach a "legal" state in a finite number of steps. The advantage of self-stabilization is that it addresses all faults through an uniform mechanism, rather than enumerate all possible faults and propose separate corrections for each of them. It models a system's capability to recover from arbitrary transient faults without any intervention from the external world.

In this chapter, we study how to apply self-stabilizing technique to maintain the consistency of distributed routing tables in a publish/subscribe system and recover from faults in the

network. In our approach, neighboring routers periodically exchange their routing table state, and take corrective actions if (and only when) necessary. We formally prove that our algorithm brings the system back to a legal global state if it starts out in a faulty state.

We further improve the efficiency of our self-stabilizing algorithm in two directions:

1. We show how to reduce the size of the stabilization messages by having the routers exchange only "sketches" of routing tables which are much smaller than the routing tables themselves. These sketches are based on Bloom filters, but we provide a new accuracy/space tradeoff.

2. The self-stabilization procedure, though correct in all cases, may not lead to the most efficient recovery from every fault. One such special case is a transient edge failure. For this important special case, we optimize the self-stabilization so that the message complexity for recovery is at most twice the cost of an "optimal" protocol to recover from such a fault.

Our simulations suggest that both the above produce significant improvements in the efficiency of the protocol.

5.1 Introduction

We propose a fault detection and recovery mechanism for distributed publish-subscribe networks based on self-stabilization. Our algorithm targets the integrity of distributed routing tables in a publish/subscribe system. Since routing information can arbitrarily be corrupted by transient faults, the self-stabilizing algorithm must ensure that incorrect routing entries are purged from the routing table and missing records are inserted into the routing table. The basic idea for keeping routing tables consistent is to let neighboring routers periodically exchange and compare their routing configurations.

The correctness of the publish-subscribe system is a global system property. No single node in the system will be able to say whether or not the system is in a correct state. However, we show that the predicate specifying a correct global state can be written as the conjunction of

many local predicates, each of which can be checked in a decentralized way using local actions. This property will help us in designing a local algorithm for the fault-tolerance. We prove that our algorithm leads the system to a legal state, and the time taken is proportional to the diameter of the network.

If implemented in a straightforward way, self-stabilization presents a large message overhead. At first glance, it seems necessary for the nodes to pass their complete routing tables to their neighbors. The routing tables are typically large (a few thousands of subscriptions), and since this exchange has to take place periodically, this could lead to significant message traffic. To reduce this overhead of checking, we propose to communicate small space "sketches" of the routing tables, instead of the whole routing tables themselves. Our sketches are based on Bloom Filters[6]. These can be used to detect inconsistencies between routing tables very accurately, and yield space improvements of two orders of magnitude.

We formally analyze the space/accuracy tradeoff of checking using our sketches. Our analysis of the false positive probability of using Bloom filters to check set equality is novel. All previous analyses of Bloom filters focused on the false positive probability for checking the set membership.

The self-stabilization algorithm, though correct in all cases, may not lead to the most efficient recovery from every fault. One such special case is a transient edge failure. In face of an edge failure, some subscriptions become obsolete, since they were generated by nodes in a partition which is now unreachable. The stabilization algorithm will remove these subscriptions, otherwise the endpoints of the broken edge will continue to receive useless events. But, if the broken edge comes back up quickly, then these subscriptions become useful again. In such a case, it is better to delay unsubscribing after a link failure.

Based on the above observation, the time to perform unsubscribing, after a link failure, is critical to the system performance. Unfortunately there is no synchronization between the time of link failure and the running of self-stabilization procedure, as the latter's period is determined by a preset timer. Consequently the next round of stabilization might kick in soon after the link failure, which may not be ideal for the system performance.

To minimize the reconfiguration overhead, we propose an *adaptive* strategy to reset the timer of self-stabilization in the presence of link failure. The timeout value is no longer fixed, but varies based on the amount of useless events received. We conduct a "competitive analysis" to show that the incurred overhead under our adaptive strategy is no more than twice the overhead of an *optimal* solution. Our simulation results reinforce the conclusion.

In summary, our contributions are as follows:

1. We present a local, self-stabilizing algorithm for adding fault-tolerance and recovery to publish-subscribe systems. We prove that irrespective of what global state the system begins in, it will reach a legal global state, and quantify the time (number of parallel steps) taken to do so.

2. We present a way to reduce the overhead of fault-tolerance by communicating "sketches" of the routing tables as opposed to the routing tables themselves. We analyze the space-accuracy tradeoffs provided by these sketches.

3. We have simulated the self-stabilization algorithm, and these simulations reinforce our theoretical analysis. We optimized the algorithm with respect to an important special case, that of a transient edge failure, and this demonstrates how the choice of a timeout for initializing the fault recovery can be very important for the overall performance.

5.2 Related work on reliable publish/subscribe systems

The closest match to our work is [36], in which Mühl *et.al.* also took the self-stabilizing approach to recover a pub/sub system from transient faults. Their main idea is that routing entries should be *leased*. Any entry which is not renewed before the expiration of its leasing period will be purged from the system. One nice thing about this approach is that it allows a publish/subscribe system to recover from both internal and external faults. For instance, if a client crashes, his subscriptions are automatically deleted after the leases are expired.

However this benefit also comes at a great cost. In order to keep an entry alive, it is necessary to renew it on time. To do that, "resubscribing" messages issued by clients are

periodically propagated inside the system. Since we must renew every subscription, this incurs significant message overhead as the amount of subscriptions registered in a system could be huge.

In [4], the authors described a guaranteed delivery service for the Gryphon system. This service depends on an acknowledgement-based scheme and requires stable storage at the event publishing sites. This approach does not work in dynamic scenarios where routers may crash and network links can go down.

In face of the topological changes, such as tree partitioning or grafting/pruning of the branches, Siena [10] suggests the usage of system primitives, including subscribe/unsubscribe, to allow subtrees to be merged or to be split. In [41], the authors argue that subscribing and unsubscribing should be treated asymmetrically and propose an optimization over the Siena approach. In another paper [15], they optimize the special case of a link failure and a link formation occurring in parallel and push the reconfiguration overhead for this special case to a minimum. In addition, epidemic algorithms [14] are also applied in unreliable and highly dynamic scenarios to provide reliable information dissemination to a group of recipients. Epidemic algorithms are lightweight, scalable and robust, but they provide guarantees only in probabilistic terms.

The reliability of a pub/sub system can also be strengthened by building the system on top of a robust overlay network. Some systems like Scribe[48] and Hermes [42] use distributed hash tables, which take advantage of the peer-to-peer routing substrate to achieve fault-tolerance.

5.3 System model

We deal with a publish-subscribe network whose nodes are organized into a single spanning tree. We assume that all communication links are FIFO. Each node holds a routing table. Many data structures have been proposed for fast matching and forwarding of events[54, 24]; we will not be concerned with the exact form of this data structure. For our purposes, the routing table is a set of tuples of the form (sub, R) where sub is a subscription, and R is a set of neighboring nodes from which the subscription was received. If any event arrives that

matches *sub*, then it is forwarded to all nodes in R, except for the node from which the event arrived.

Self-stabilizing algorithms can be built in a modular fashion[23]. Our algorithm stabilizes the state of the routing tables on a tree-based publish-subscribe system. This can be layered on top of another algorithm which stabilizes the spanning tree itself.

5.4 Self-stabilizing algorithm

The self-stabilizing algorithm is concerned with the consistency between the routing tables held by the nodes. Neighboring nodes periodically exchange the states of their routing tables. A node takes corrective actions if its routing information is inconsistent with some neighbor. Each node makes local corrections independently and asynchronously. Through a sequence of local corrections, we restore the consistency among the distributed routing tables.

5.4.1 Local legality implies global legality

The definitions of legal local/global states lie at the core of our algorithm. Before defining them, we first introduce some notations and concepts.

An undirected edge connecting nodes a and b is denoted by $\{a, b\}$. It is composed of two directed edges, denoted by (a, b) and (b, a). For node v, let $N(v)$ denote the neighbor set of v. If we remove edge $\{a, b\}$ from the tree T, the whole tree is divided into two parts. The subtree rooted at a is denoted by T_a^b and the subtree rooted at b is denoted by T_b^a.

For directed edge (a, b), the *filter* $F_{a\to b}$ is the union of all subscriptions registered at node a, which sends matching events to node b; in other words, it is the set of all subscriptions that a has so far received from b. The set S_b^a is the union of all subscriptions that are generated by nodes in the subtree T_b^a.

Definition 6 *A* message transition *can be one of the following:*
(1)generation: A node produces a new (un)subscription.
(2)propagation: A (un)subscription is being transmitted over the link.
(3)consumption: The receiver processes an incoming (un)subscription.

Definition 7 *A system is* quiescent *if there are no subscribe/unsubscribe messages in transit.*

Suppose the neighbor set of b, $N(b) = \{n_1, n_2, \ldots n_k, a\}$. Let $X_{b\to a}$ denote the subscriptions in transit from b to a and $Y_{b\to a}$ denote the unsubscriptions in transit from b to a. Let L_b denote the local subscriptions issued by node b. We first define what it means for an edge to be locally legal.

Definition 8 *The* directed edge (a, b) *is* locally legal*, iff* $F_{a\to b} \cup X_{b\to a} - Y_{b\to a} = \cup_{i=1}^{k} F_{b\to n_i} \cup L_b$.

Definition 9 *The* undirected edge $\{a, b\}$ *is* locally legal*, iff both* (a, b) *and* (b, a) *are locally legal.*

We now define what it means for an edge to be globally legal. We assume that every received subscription will be forwarded unless it is a duplicate.

Definition 10 *If the system is quiescent, then edge* $\{a, b\}$ *is globally legal, iff* $F_{a\to b} = S_b^a$ *and* $F_{b\to a} = S_a^b$.

Definition 11 *A publish-subscribe system is in a legal state if one of the two conditions holds:*
(1)it is quiescent and all edges are globally legal or
(2)it can be reached from a legal quiescent state by a finite sequence of transitions.

The global legality of edges is hard to check directly, since it is a predicate that involves the state of the whole system. However, the local legality of an edge can be (more) easily checked. We now state a theorem which shows that the predicate defining the global legality of the system can be written as the conjunction of many local predicates, one for each edge.

Theorem 4 *The publish-subscribe system is legal iff every edge is locally legal.*

Proof: A legal system can either be quiescent or have messages in transit.

(1) The system is quiescent

The Necessity: We want to prove that inside a legal system, every edge is locally legal. Since this is a legal system, we know every edge inside is globally legal. For an arbitrary

edge (a, b),that means $F_{a\to b} = S_b^a$ (Definition 10).Assume $N(b) = \{n_1, n_2, \ldots n_k, a\}$. We have $S_b^a = \cup_{i=1}^k S_{n_i}^b \cup L_b$. We also have $F_{b\to n_i} = S_{n_i}^b$ (every edge is globally legal). By combining the two, we get $F_{a\to b} = \cup_{i=1}^k F_{b\to n_i} \cup L_b$. This concludes the proof that edge (a, b) is locally legal.

The Sufficiency: For an arbitrary edge (a, b), if it is locally legal,it satisfies $F_{a\to b} = \cup_{i=1}^k F_{b\to n_i} \cup L_b$ in a quiescent system. We need to show $F_{a\to b} = S_b^a$ (edge (a, b) is globally legal). We prove this by induction on the height of the subtree T_b^a.

Base Case: Height is equal to 1 and T_b^a only contains b. We have the following equations: $\cup_{i=1}^k F_{b\to n_i} = \phi, L_b = S_b^a$. This fact infers $F_{a\to b} = S_b^a$. We reach edge (a, b) is globally legal.

Inductive Hypothesis: Assume edge (a, b) is globally legal when the height of T_b^a is no greater than $(k - 1)$.

Inductive Step: Height is k. The subtree $T_{n_i}^b$ has its height $\leq (k - 1)$. According to the inductive hypothesis, edge (b, n_i) is globally legal. This means $F_{b\to n_i} = S_{n_i}^b$. In turn we have

$$F_{a\to b} = \cup_{i=1}^k F_{b\to n_i} \cup L_b = \cup_{i=1}^k S_{n_i}^b \cup L_b = S_b^a$$

Again edge (a, b) is globally legal. Given the fact that every edge in the tree is globally legal, we conclude that the global state is legal.

(2) The system has messages in transit

The Necessity: According to Definition 10, a legal system which is not quiescent is derived from a quiescent one by a finite sequence of message transits. We examine the first transit in that sequence. Suppose w.l.o.g the transit happens along edge (b, a). Before it happens, we know edge (a, b) is locally legal (This is proved by part (1)). We have $F_{a\to b} = \cup_{i=1}^k F_{b\to n_i} \cup L_b$. After this transit, the union of subscriptions registered at b becomes $\cup_{i=1}^k F'_{b\to n_i} \cup L'_b$. The subscriptions held by a and b are different. But the message in transit captures the difference. We have the following equation

$$F_{a\to b} \cup X_{b\to a} - Y_{b\to a} = \cup_{i=1}^k F'_{b\to n_i} \cup L'_b$$

Edge (a,b) remains locally legal (Definition 8). Therefore a single message transit preserves the local legality of each edge. Hence through a finite sequence of message transits, a non-quiescent legal system still guarantees that each edge of the network is locally legal.

The Sufficiency: Let L be a system state where every edge is locally legal and network links hold messages.We need to show that there exists some quiescent legal state Q and a finite sequence of message transits *seq* that brings Q to L. We prove this by induction on the number of messages which are in transit in L. We use k to denote the number.

Base Case: There's no message in transit. We have $k = 0$. In part (1), we've proved that in a quiescent system, the fact that every edge is locally legal ensures a legal system.Thus *seq* is the null sequence. In the base case, Q=L.

Inductive Hypothesis: We assume that L can be reached from Q through *seq* as long as the number of messages in transit in L is no greater than $(k-1)$ and every edge is locally legal.

Inductive Step: We define L to be a system state where every edge is locally legal and the number of messages in transit is k. We define L' to be another system state where every edge is locally legal and the number of messages in transit is no greater than $(k-1)$. According to the inductive hypothesis, we know that L' is a legal system. In the remaining part, we want to prove that L can be reached from L' by a sequence of message transits *seq'*.

We choose an arbitrary message which is in transit along edge (u, v). We reverse the progress by letting u withdraw the message and cancel the local modification. If each edge incident on u is locally legal, this action won't disturb its local legality. If node u is the source of the message,u will remove the message from all incident edges. Since the degree of u is no less than one, the number of messages inside the system decreases at least by one. We conclude L can be reached from L'.

On the other hand, if the message was received from some neighbor n_j by u, u will withdraw the message from other incident edges but return it back to n_j along edge (u, n_j). This action still maintains the local legality of each edge. If the degree of u is greater than two, the number of messages in transit decreases at least by one. L can be reached from L'. Otherwise we repeat the same action at node n_j. It is clear that such a recursive process will terminate after a finite

steps of moves. The worst case is that the tree topology is a chain, in which the recursive reversal will end up at the exact source of the message. Furthermore such a reversal decreases the number of messages in transit at least by one. Once again L can be reached from L'.

In summary, we conclude L can be reached from L' by a finite step of message transits *seq'*. L' itself can be reached from a quiescent legal state Q by a finite step of message transits *seq*. By following *seq* and *seq'* in order, we can bring Q to L. This completes the proof. ■

5.4.2 The edge stabilization algorithm

Given the above theorem, we only need to stabilize each directed edge into a legal state, and the system will reach a globally legal state. It is easy to set a (faulty) directed edge to a legal local state using appropriate subscriptions/unsubscriptions. However, stabilizing a faulty edge might "disturb" a neighboring edge, and cause it to move from a legal to an illegal state, so that the global state is still illegal. Informally speaking, such "disturbances" can flow only along a simple path in the tree, and have to eventually stop at a leaf. Thus, eventually the system will reach a globally legal state.

A timer is assigned to each directed edge in the network, and the (directed) edge stabilization procedure is initiated upon expiry of the timer. The period of the timer controls the frequency of stabilization, and hence the message overhead (more discussion of the timer appears in Section 5.6). The source node of a directed edge is responsible for the stabilization. A single round of the procedure consists of two phases: an observe phase followed by a correction phase. We describe the algorithm for directed edge (a, b). All the directed edges are being stabilized in parallel in this manner. Table 5.1 summarizes the appearing variables.

$N(a)$	$\{n_1, n_2, \ldots, n_k, b\}$
$S(a)$	$\cup_{i=1}^{k} F_{a \to n_i} \cup L_a$
C_1	$S(a) - F_{b \to a}$
C_2	$F_{b \to a} - S(a)$

Table 5.1: List of variables

Algorithm 5.1 Actions at node a

Event: timeout at t_1 (observe phase)

1: compute $S(a)$ at time t_1
2: send an "observer" to b
3: reset the timer for the next round

Event: get the reply from b (correction phase)
// b's reply is $F_{b\to a}$

1: **if** $S(a) \neq F_{b\to a}$ **then**
2: compute C_1, C_2
3: send C_1, C_2 to b

It is important to note that the correction phase at node b is initiated only if edge (a, b), and hence the whole system was not in a legal state. Thus, if the system is in a legal state, then the self-stabilization will not add any additional subscriptions/unsubscriptions to the system.

Algorithm 5.2 Actions at node b

Event: receive an "observer" from a (observe phase)

1: compute $F_{b\to a}$
2: return $F_{b\to a}$ to a

Event: get the reply from a (correction phase)
// a's reply are C_1 and C_2

1: subscribe to each record contained in C_1
2: unsubscribe to each record contained in C_2

Theorem 5 *If every directed edge in the tree executes the above stabilization process, then the system will reach a legal global state irrespective of which state it starts in.*

Proof: We first verify that the above procedure can set edge (a, b) to a locally legal state. According to the algorithm, $S(a)$ was computed just before node a sent out the observer. Since the network edges are FIFO, $F_{b\to a}$ was a cumulative result at b after processing all the messages sent by a before receiving the observer. If edge (a, b) was locally legal, we must have $F_{b\to a} = S(a)$ (Definition 8). Otherwise a informed b of the difference by sending C_1 and C_2. It is clear that edge (a, b) can be restored to a locally legal state in this manner.

However the correction of edge (a, b) might disturb other edges emanating from b and cause them to move from a legal state to an illegal one, so that the global state is still illegal. If

there is any such, node b will stabilize it. But the correction of an outgoing edge at b will not re-disturb edge (a, b), as the local legality of edge (a, b) is totally determined by incoming edges (except (b, a)) at a. As a result, such a ripple effect can only propagate in one direction, starting from the root of the subtree T_b^a and going down towards the leaves. Since the tree has a finite diameter, this procedure must terminate in the end. Till then every edge is locally legal. Based on Theorem 4, we know the entire system reaches a global legal state.

As to the time taken till the system reaches a legal state, it is no more than d parallel time steps, where d is the diameter of the tree, and a parallel time step is the time required for an edge stabilization procedure (observe+correct phases) at all edges in parallel. ■

5.5 Reducing the message overhead

An important component of the local stabilization algorithm is the checking of the equality between the two tables S_a^b and $F_{b\to a}$. One way to do this is to send the entire table $F_{b\to a}$ across from node b to node a, but this would result in a large message overhead for the following reasons:

1. The objects being sent across and compared are large sets of subscriptions. These routing tables might contain thousands of subscriptions, and if each subscription takes a few tens of bytes, then these messages would be of the order of a few hundred kilobytes or more. In addition, comparing these large sets would be significant computational overhead.

2. Self-stabilization is a periodic system behavior, which further exacerbates the above problem.

Our approach to reducing this overhead is as follows. Instead of sending the entire routing tables across, we send only a sketch of the table to the neighboring node. This sketch takes much smaller space than the table itself.If the sketches don't match, then the two sets are clearly unequal. If they do match, then the sets are equal with high probability. In other words,there is some probability that the routing tables are actually inconsistent, but the sketches do not reveal

it. If the probability of false negative is low, this approach leads to a significant improvement in the efficiency.

We summarize the desirable properties of a sketch:

1. The size of the sketch should be small compared to the original routing table size
2. It should be able to detect inconsistencies with high probability, and with low computational overhead
3. The cost of maintenance should be low, i.e. every time a subscription or unsubscription is received, we should be able to update the sketch quickly.
4. Since we need to compute the union of routing tables while checking, we need to be able to (quickly) compute the sketch of the union of sets given the sketches of the sets.

We suggest using the Bloom Filter as the sketch of each set.There are many advantages associated with bloom filter. First of all, it is a compact data structure. Suppose a routing table contains 1000 subscriptions. As an example, let's assume the average size per entry is 50 bytes. Then the set size is 5×10^4 bytes. In comparison if a bloom filter allocates 4 bits to represent an entry, then we have $m/n = 4$(bits/entry). We will show later that this ratio guarantees a low error rate. The filter size is only 4×10^3 bytes,which is a tenfold saving in storage.

To support dynamic sets, we use a variant of bloom filter, called "counting bloom filter". In counting bloom filter, each bit is associated with a small counter. When an item is inserted, the corresponding counters are incremented; when an item is deleted, the corresponding counters are decremented. We reset a bit to 0 if the associated counter decreases down to 0. The counters are maintained locally. We only exchange and compare the filters. The updating requires $O(1)$ time.

Since each bloom filter is a bit vector of the same size, it is straightforward to union separate filters, maintained for each F, to get S. Furthermore for every new (un)subscription, we only need to update a single filter. And the updating complexity is constant time.

The last merit is related to the low error rate. At the core of bloom filter design, there is a clear tradeoff among the bloom filter size(m), the number of hash functions used(k) and the probability of a false positive (p). Below we analyze the various tradeoffs associated with using a Bloom filter for the purpose of testing equality between sets.

5.5.1 Bloom filter for testing set membership

A Bloom filter is a method for representing a set $A = \{a_1, a_2, \ldots, a_n\}$ of n elements to support membership queries. It was invented by Burton Bloom[6] in 1970. The construction of Bloom filter is as follows. First allocate a vector v of m bits, initially all set to 0. Choose k independent hash functions,$h_1, h_2, \ldots, h_k$, each with range $\{1, \ldots, m\}$. For each element $a \in A$, the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$ in v are set to 1. A particular bit might be set to 1 multiple times. Given a query that asks if $b \in A$, we check the bits at positions $h_1(b), h_2(b), \ldots, h_k(b)$. If any of them is 0, then certainly $b \notin A$. Otherwise we reply that $b \in A$ even though there exists a small probability that this isn't the case. This probability is termed as "false positive" probability for a membership query. The parameters k and m should be chosen such that the rate of a false positive is acceptable. We refer to [6] for further details.

As described in [21], there exists a tradeoff between m, k and p_{err}, the probability of a false positive for a membership query. Observe that after inserting n keys into a filter of size m, the probability that a particular bit is still 0 is exactly

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}} \tag{5.1}$$

Hence the probability of a false positive in this situation is

$$p_{err} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{5.2}$$

5.5.2 Bloom filter for testing set equality

In self-stabilization, we do not use a Bloom filter to test for set membership, but to compare if two sets are equal. More precisely, we want to check if the union of a few sets ($S_a^b =$

$\cup_{i=1}^{k} F_{a \to n_i} \cup L_a$) equals another set ($F_{b \to a}$). This calls for a new analysis of the tradeoffs between the false positives and the parameters of the Bloom filter. We now sketch our analysis of the false positive probability for the context of set equality, and graph the resulting tradeoffs obtained.

Let B_S denote the Bloom filter of set S. Let m denote the size (in bits) of the Bloom filter, and k the number of hash functions. We want to compute the probability that B_A and B_B are equal, though A and B are unequal. Let p denote the false positive probability. We have the following theorem about p.

Theorem 6

$$p = (1 - p_1 - p_2)^m$$

where

$$p_1 = e^{-\frac{k \cdot |B|}{m}} \cdot [1 - e^{-\frac{k \cdot |A-B|}{m}}]$$

and

$$p_2 = e^{-\frac{k \cdot |A|}{m}} \cdot [1 - e^{-\frac{k \cdot |B-A|}{m}}]$$

Proof: If $B_A \neq B_B$, it means that some bits in B_A which are set to 1 remain 0 in B_B or the other way round. For an arbitrary bit in the bloom filter, we use E_1 to represent the event that the bit is 1 in B_A yet remains 0 in B_B and use E_2 to denote the event that the bit is 1 in B_B yet remains 0 in B_A. Let p_1 and p_2 stand for their probability respectively.

$$\begin{aligned} p_1 &= Pr[(\text{the bit is 1 in } B_A) \cap (\text{the bit is 0 in } B_B)] \\ &= Pr[(\text{the bit is 1 in } B_A) | (\text{the bit is 0 in } B_B)] \cdot Pr[\text{the bit is 0 in } B_B] \end{aligned} \tag{5.3}$$

As we know, after inserting n keys into a bloom filter of size m, the probability that a particular bit remains 0 is exactly

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$

$$\therefore Pr[\text{the bit is 0 in } B_B] = e^{-\frac{k \cdot |B|}{m}}$$

Given the condition that the bit remains 0 in B_B, if it becomes 1 in B_A, it can only be set by

some element belonging to $(A - B)$.

$$\therefore Pr[(\text{the bit is 1 in } B_A)|(\text{the bit is 0 in } B_B)] = 1 - e^{-\frac{k \cdot |A-B|}{m}}$$

$$\therefore p_1 = e^{-\frac{k \cdot |B|}{m}} \cdot [1 - e^{-\frac{k \cdot |A-B|}{m}}]$$

Due to the symmetry, we have

$$p_2 = e^{-\frac{k \cdot |A|}{m}} \cdot [1 - e^{-\frac{k \cdot |B-A|}{m}}]$$

Now the fact of $B_A = B_B$ indicates that none of the bits in the bloom filter can satisfy either E_1 or E_2. By taking the complement, we have

$$p = (1 - p_1 - p_2)^m$$

■

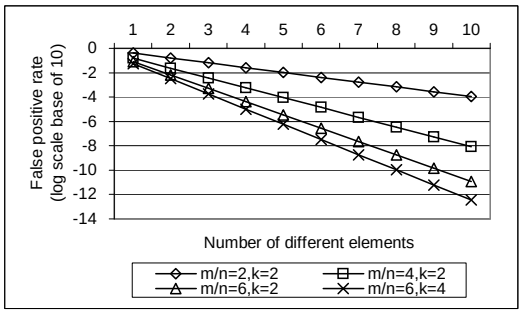


Figure 5.1: The probability of false positive under various m/n and k combinations. The y-axis is the error probability in a log-scale. The x-axis is the size of the difference $|A - B|$.

We now sketch the value of p with respect to various values of $k, m/n$. For simplicity, we only consider the case where $(A - B) \neq \emptyset$ but $B_A \subseteq B_B$. It is important to note that this probability(p_A) is a loose upper bound of p, whose value will be even smaller. In Figure 5.1, the probability decreases (leading to a more accurate test for equality) when the difference between the sets is large. As k increases, the computation overhead for maintaining the Bloom filter increases. The parameter m/n is the number of bits used per element: as it increases,

the space overhead also increases, but the false positive probability decreases. Thus, by using 4 bits per element and 2 hash functions, the false positive rate for sets differing by 3 elements is only about 0.001, and this can be further decreased by increasing k or m/n.

5.6 Optimization of transient edge failure

The self-stabilization algorithm, though correct in all cases, may not lead to the most efficient recovery from every fault. One such special case is a transient edge failure. Transient edge failure and the accompanied reconfiguration are common issues in publish-subscribe systems. In the literature, [41, 15] have specifically studied this problem.

When a link is removed, each of its endpoints is no longer able to route matching events to the other partition. Hence the stabilization procedure will remove the obsolete subscriptions inside each partition. But if the link comes back up quickly, these removed subscriptions will be re-added. It is better to delay unsubscribing in such a case.

However it is always possible that the edge never re-forms. If we delay unsubscribing, the endpoints will keep receiving useless events due to the presence of obsolete subscriptions. This also exacerbates the system performance.

Based on the above observation, the time to perform unsubscribing, after a link failure, is critical to the system performance. Unfortunately there is no synchronization between the time of link failure and the running of self-stabilization procedure, as the latter's period is determined by a preset timer. Consequently the next round of stabilization might kick in soon after the link failure, which may not be ideal for the system performance.

In this section, we propose an *adaptive* algorithm to reset the timer of the self-stabilization procedure after a link failure. The timeout value is no longer fixed, but varies based on the amount of useless events received. We derive a "competitive analysis" to prove that the incurred overhead under our adaptive strategy is no more than twice the overhead of an *optimal* solution. Our simulation results reinforce the conclusion.

5.6.1 The adaptive algorithm

In order to quantify the incurred overhead under an edge failure, we first give a definition of the message cost inside the system. Here a message can be an event, a subscription or an unsubscription.

Definition 12 *The cost of a single message is measured as the total number of hops traversed by that message inside the system.*

As we have seen, an edge failure/restoration induces two types of costs: the cost of unsubscribing/re-subscribing and the cost of useless events.

In terms of the first cost, we consider the extreme where each subscription inside the system is only shared by one subscriber. Under this condition, any reconfiguration message generated by an endpoint of the broken edge will propagate throughout the entire subtree rooted at itself. This gives us a coarse estimation of the cost, which is upper bounded by the size of the subtree.

With respect to the second cost, it depends on the system load and the duration of link failure. If the load is high (a high event publishing rate) and the duration is long, two endpoints of the broken edge will receive more useless events. The cost becomes significant, otherwise it is moderate.

The total reconfiguration overhead under an edge failure is a summation of the two costs. According to our analysis, one of them is bounded and the other is boundless. Furthermore the two costs are correlated, as unsubscribing can suppress the growth of the cost of useless events. In order to minimize the total overhead, we have to make a decision about when to unsubscribe (when to invoke the next round of stabilization). An optimal decision is based on the total amount of useless events that will be generated, which is not apriori. Thus we propose an approximate online algorithm called the *adaptive* algorithm.

Algorithm description According to Section 5.3, we assume the application interacts with some underlying topology maintenance algorithm. That algorithm notifies the application about the event of link failure. The application then cancels the scheduling of the next round of stabilization. Meanwhile each endpoint of the broken edge initializes a threshold to monitor

the cost of useless events received. The threshold is set to be the size of the subtree rooted at the endpoint. The next round of stabilization is invoked either the cost of useless events exceeds the threshold or the link re-forms, whichever happens first.

We have the following theorem w.r.t the overhead of the adaptive algorithm.

Theorem 7 *Under the adaptive strategy, the resulting cost is never more than twice the cost of the optimal solution.*

Proof: Let T be the size of the spanning tree, E be the cost generated by useless events and δ be the duration of the link failure. We use γ to denote the ratio of the cost of the adaptive strategy and the cost of the optimal solution. If within the time of δ, we have $E \leq T$. Under the adaptive strategy, the cost is equal to E. The cost of the optimal solution is also E. We get $\gamma = 1$. If within the period of δ, we have $E > T$. Under the adaptive strategy, the cost becomes $E + 2T$ $(\delta \neq \infty)$. The optimal cost is almost $2T$ $(\delta \neq \infty)$. We have $1.5 < \gamma < 2$ $\left(\frac{E}{2T} > 0.5\right)$. If $\delta = \infty$, which indicates that the link goes down permanently, the optimal solution produces a cost of T. The cost of the adaptive strategy amounts to $E + T$. We have $\gamma \approx 2$ $(E \approx T)$. ■

We introduced the framework of the algorithm. But two questions remain unresolved: (1)How to identify useless events and keep track of the corresponding cost? (2)How to estimate the size of the subtree in a distributed manner?

To keep track of the number of hops traversed by an useless event,say e, we add a counter to e. The counter is initialized to zero at the source of the event. As e goes through each hop, the receiver node will either increment the counter by one or reset it to zero. If the receiver has in its routing table two or more subscriptions matching e, this indicates e is a useful event and we reset the counter to zero. Otherwise e is potentially an useless event and we increment the counter by one. When e finally reaches an endpoint,if that node can't find any subscriber to receive e, it decides that e is an useless event. The endpoint updates the cost of unwanted events received by adding the value of the counter.

As to the computation of the size of the subtree rooted at each node, we can start from the leaves and calculate the subtree size recursively in a bottom-up fashion. This is the

implementation of a centralized algorithm and the transformation to a distributed version is straightforward.

5.6.2 Simulation

We simulated our algorithm using the discrete event simulator OMNET++[37]. The reconfiguration overhead is chosen to be our performance metric. We studied the following three strategies under a single link failure:

1. The "strawman algorithm". We borrowed the name from [41]. It resets the timer upon a link failure. It also invokes the stabilization without delay.

2. The "static algorithm". It does not change the preset timer. The stabilization runs at regular intervals.

3. Our "adaptive algorithm". It resets the timer upon a link failure. The stabilization is re-invoked either after a sufficient number of useless events are received or when the link re-forms, whichever happens first.

Configuration In our simulations, both an event and a subscription are chosen to be 3-character random strings, where each character can be any of the 26 lowercase letters and 26 capital letters. An event matches a subscription if the two strings are identical. For example, an event can be "Ace", a subscription can be "ace", but they don't match. The topology is a single spanning tree, consisting of 100 nodes.

Each scenario is uniquely identified by a combination of the following parameters:

- *Publish Rate*: The publish rate regulates the system load. A high rate indicates a heavy load, while a low rate corresponds to a light load. We simulate two scenarios: a *light system load* using a publish interval of 5.0 seconds and a *heavy system load* using a publish interval of 0.1 seconds.

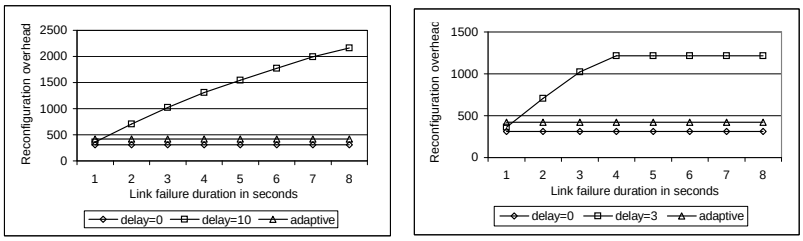
- *Subscribe Rate*: The subscribe rate controls the density of the subscriptions. A high density indicates a big extent to which a subscription is shared by many subscribers. In

turn it means a shorter path for an (un)subscription to propagate. This reduces the cost of unsubscribing/re-subscribing. In reality, a subscription unlikely has a big degree of sharing. Thus we set the subscribe interval to be 2.0 seconds. In addition, each router can subscribe to at most 20 event patterns.

- *Fixed Timer*: Only the "static" algorithm sticks to the fixed timer and invokes the stabilization at regular intervals. We choose two timeout values: a longer one of 10 seconds and a shorter one to be 3 seconds.

For each run, we plot the overhead against the duration of link failure under different strategies.

Result analysis Under a heavy load(Figure 5.2), the endpoints expect to receive more useless events. It's ideal to unsubscribe early to limit the increasing cost of useless events. In Figure 5.2, the strawman curve has the lowest overhead, as it unsubscribes without delay. The adaptive curve has a slight increase, for it delays unsubscribing a little bit. The static algorithm has a poor performance, but a shorter timeout brings down the overhead by 50%.

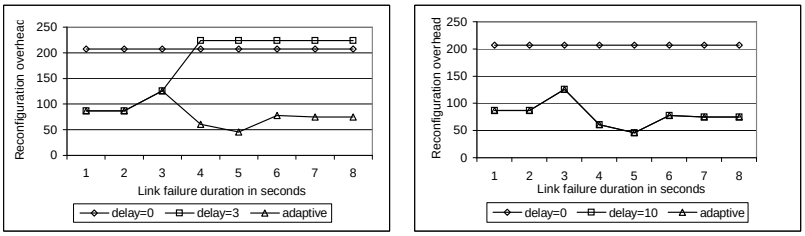


(a) (b)

Figure 5.2: Reconfiguration overhead under heavy system load

Under a light load(Figure 5.3), the cost of unwanted events is negligible due to the rare occurrences of unwanted events. It is better to delay unsubscribing. Therefore any static algorithm with large timeout performs well under this condition. Meanwhile the adaptive algorithm also yields the same amount of cost. This observation is proved in Theorem 7. This time both the strawman and the static strategy with short timeout value generate huge

overhead. As a comparison, our adaptive algorithm saves two thirds of the cost of the strawman approach.



(a) (b)

Figure 5.3: Reconfiguration overhead under light system load

In summary, neither the *static* nor the *strawman* does well for both the lightly loaded and heavily loaded cases. However, the *adaptive* algorithm for triggering the reconfiguration shows a good (though not optimal) performance in both cases.

CHAPTER 6. Conclusion

Publish/subscribe is a generic paradigm for event dissemination, where *events* generated by publishers are delivered to subscribers, who express their interests in certain types of events via *subscriptions*. The strength of the publish/subscribe model lies in the anonymity between publishers and subscribers, which makes it well suited for building modern loosely coupled distributed applications. Among different variants of publish/subscribe, *content-based* publish/subscribe is especially expressive, since subscriptions consist of user defined predicates on the complete content of an event. A content-based system is usually constructed as a distributed network of routers, so that it can scale with the number of users, and avoid a single point of failure.

However, the flexibility of content-based systems currently comes at a significant cost. Event forwarding is a complex task, since each forwarding decision is now based on the content of an event, rather than on a fixed destination address, as in IP routing, or on a fixed group name, as in traditional multicast. Meanwhile in order to receive events published at remote sites, it is necessary for each router to propagate locally registered subscriptions. But disseminating subscriptions in the network can also be expensive both space-wise and time-wise. Furthermore since it is built as a distributed network of multiple routers, the system needs to be tolerant of various faults such as link failure, message loss and data corruption.

In this thesis, we develop techniques for building a scalable and reliable distributed content-based publish/subscribe system. Our contributions are as follows.

First, we expedite distributed event forwarding through a strategy called "lookup reuse" which replaces a large fraction of expensive content-match with much cheaper hash-table lookups. This is made possible by letting routers share and reuse computed matching re-

sults. In contrast, current event routing strategy performs content-match independently from router to router. Simulations show that lookup reuse decreases the event processing overhead on average by 40%.

Next, we exploit an optimization called *subscription covering* to decrease the number of subscriptions forwarded in the system. To efficiently manage subscription covering, we build a two-layer framework decoupling the detection of covering from its maintenance. Simulations show that our modular structure is faster than previously known solution, which combines the two functions into a single data structure.

We also propose a new concept called *approximate covering* that provably obtains much of the benefits of exact covering at a fraction of its cost. A concrete solution based on Z space filling curve is presented. Our approach is novel and valuable, as covering detection is shown to be a hard problem. Its special case can be formulated as the problem of multidimensional point dominance, for which no worst-case efficient solution is known.

Further, we design a scalable fault-tolerance scheme to fortify content-based systems through *self-stabilization*, where a corrupted component of the distributed system can automatically revert to a "correct" state through local corrections. This scheme has the advantage of addressing all the faults through a uniform mechanism.

BIBLIOGRAPHY

[1] Aguilera, M., Strom, R., Sturman, D., Astley, M., and Chandra, T. (1999). Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*.

[2] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

[3] Bharambe, A. R., Rao, S., and Seshan, S. (2002). Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the 1st Workshop on Network and System Support for Game*.

[4] Bhola, S., Strom, R., Bagchi, S., Zhao, Y., and Auerbach, J. (2002). Exact-once delivery in a content-based publish-subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks*.

[5] Birman, K. P. (1993). The process group approach to reliable distributed computing. *Communications of the ACM*, 12(36):36–53.

[6] Bloom, B. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7).

[7] Campailla, A., Chaki, S., Clarke, E., Jha, S., and Veith, H. (2001). Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 19th International Conference on Software Engineering*.

[8] Cao, F. and Singh, J. P. (2004). Efficient event routing in content-based publish/subscribe

service network. In *Proceedings of INFOCOM 2004. the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies.*

[9] Cao, F. and Singh, J. P. (2005). MEDYM: Match-early with dynamic multicast for content-based publish-subscribe networks. In *Proceedings of the 6th International Middleware Conference.*

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

[11] Carzaniga, A., Rutherford, M., and Wolf, A. L. (2004). A routing scheme for content-based networking. In *Proceedings of the 23rd Conference of the IEEE Communications Society.*

[12] Carzaniga, A. and Wolf, A. L. (2003). Forwarding in a content-based network. In *Proceedings of the 2003 Conference on Applications,Technologies, Architectures, and Protocols for Computer Communications.*

[13] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. (2002). Exploiting network proximity in distributed hash tables. In *Proceedings of the International Workshop on Future Directions in Distributed Computing.*

[14] Costa, P., Migliavacca, M., Picco, G. P., and Cugola, G. (2003). Epidemic algorithms for reliable content-based publish-subscribe: an evaluation. In *Proceedings of the 24th International Conference on Distributed Computing Systems.*

[15] Cugola, G., Frey, D., Murphy, A. L., and Picco, G. P. (2004). Minimizing the reconfiguration overhead in content based publish subscribe. In *Proceedings of the 19th ACM Symposium on Applied Computing.*

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

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

[18] Edelsbrunner, H. and Overmars, M. H. (1982). On the equivalence of some rectangle problems. *Information Processing Letters*, 14(3):124–127.

[19] Fabret, F., Jacobsen, A., Llirbat, F., Pereira, J., Ross, K., and Shasha, D. (2001). Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of the 20th International Conference on Management of Data*.

[20] Faloutsos, C. (1986). Multiattribute hashing using gray codes. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*.

[21] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293.

[22] Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, 30:170–231.

[23] Gouda, M. G. and Multari, N. (1991). Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458.

[24] Gryphon IBM Research (1997-2007). `http://www.research.ibm.com/distributedmessaging/gryphon.html`.

[25] Hilbert, D. (1891). Uber die stegie abbildung einer linie auf flachenstuck. *Mathematische Annalen*, 38:459–460.

[26] IBM WebSphere MQ (2005). `http://www-306.ibm.com/software/integration/wmq/`.

[27] Java Message Service (1994-2006). `http://java.sun.com/products/jms/`.

[28] Kale, S., Hazan, E., Cao, F., and Singh, J. P. (2005). Analysis and algorithms for content-based event matching. In *Proceedings of the 4th International Workshop on Distributed Event-Based Systems*.

[29] Kulik, J. (2003). Fast and flexible forwarding for internet subscription systems. In *In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems*.

[30] Li, G., Hou, S., and Jacobsen, H.-A. (2005). A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *Proceedings of the 25th International Conference on Distributed Computing Systems*.

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

[32] Morton, G. (1966). A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM.

[33] Mühl, G. (2002). *Large-scale content-based publish/subscribe systems*. PhD dissertation, Darmstadt University of Technology.

[34] Mühl, G., Fiege, L., and Buchmann, A. (2001). Generic constraints for content-based publish-subscribe systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems*.

[35] Mühl, G., Fiege, L., Gärtner, F. C., and Buchmann, A. (2002). Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*.

[36] Mühl, G., Jaeger, M. A., Herrmann, K., Weis, T., Fiege, L., and Ulbrich, A. (2005). Self-stabilizing publish/subscribe systems: algorithms and evaluation. In *Proceedings of the 11th European Conference on Parallel Processing*.

[37] OMNeT++ documentation and tutorials (1996). `http://www.omnetpp.org`.

[38] Oracle spatial (2001). `http://www.orafaq.com/faq/spatial`.

[39] Ouksel, A., Jurca, O., Podnar, I., and Aberer, K. (2006). Efficient probabilistic subsumption checking for content-based publish/subscribe systems. In *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference.*

[40] Pereira, J., Fabret, F., Llirbat, F., and Shasha, D. (2000). Efficient matching for web-based publish/subscribe systems. In *Proceedings of the International Conference on Cooperative Information Systems.*

[41] Picco, G. P., Cugola, G., and Murphy, A. L. (2003). Efficient content-based event dispatching in the presence of topological reconfiguration. In *Proceedings of the 23rd International Conference on Distributed Computing Systems.*

[42] Pietzuch, P. (2004). *Hermes: a scalable event-based middleware.* PhD dissertation, University of Cambridge.

[43] Pietzuch, P. R. and Bhola, S. (2003). Congestion control in a reliable scalable message-oriented middleware. In *Proceedings of the 4th ACM/IFIP/USENIX international conference on Middleware.*

[44] Preparata, F. P. and Shamos, M. I. (1985). *Computational geometry: an introduction.* Springer-Verlag.

[45] R.Chand and Felber, P. (2004). Xnet: a reliable content-based publish/subscribe system. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems.*

[46] Riabov, A., Zhen, L., Wolf, J., Yu, P., and Zhang, L. (2002). Clustering algorithms for content-based publish-subscribe systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems.*

[47] Riabov, A., Zhen, L., Wolf, J., Yu, P., and Zhang, L. (2003). New algorithms for content-based publication-subscription systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems.*

[48] Rowstron, A., Kermarrec, A.-M., Castro, M., and Druschel, P. (2001). SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third COST264 International Workshop on Networked Group Communication*.

[49] RSS wikipedia article (2007). `http://en.wikipedia.org/wiki/RSS_(file_format)`.

[50] Shen, Z., Aluru, S., and Tirthapura, S. (2005). Indexing for subscription covering in publish-subscribe systems. In *Proceedings of the 18th ISCA International Conference on Parallel and Distributed Computing Systems*.

[51] Shen, Z. and Tirthapura, S. (2004). Self-stabilizing routing in publish-subscribe systems. In *Proceedings of the 4th International Workshop on Distributed Event-based Systems*.

[52] Shen, Z. and Tirthapura, S. (2006). Faster event forwarding in a content-based publish-subscribe system through lookup reuse. In *Proceedings of the IEEE International Symposium on Network and Computing Applications*.

[53] Shen, Z. and Tirthapura, S. (2007). Approximate covering detection in content-based publish-subscribe using space filling curves. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*.

[54] Siena publish/subscribe (2006). `http://serl.cs.colorado.edu/siena/index.html`.

[55] Skeen, D. (1992). An information bus architecture for large-scale, decision-support environments. In *Unix Conference Proceedings*.

[56] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. (2003). Chord : a scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*.

[57] Tarkoma, S. and Kangasharju, J. (2005). Filter merging for efficient information dissemination. In *Proceedings of the 2005 CoopIS,DOA and ODBASE Confederated International Conferences*.

[58] Tarkoma, S. and Kangasharju, J. (2006). Optimizing content-based routers: posets and forests. *Distributed Computing*, 19(1):62–77.

[59] TIBCO Rendezvous (2000-2007). `http://www.tibco.com/software/messaging/rendezvous/`.

[60] Tirthapura, S., Seal, S., and Aluru, S. (2006). A formal analysis of space filling curves for parallel domain decomposition. In *Proceedings of IEEE International Conference on Parallel Processing*.

[61] Triantafillou, P. and Economides, A. (2004). Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems*.

[62] Wang, Y., Qiu, L., Achlioptas, D., Das, G., Larson, P., and Wang, H. J. (2002). Subscription partitioning and routing in content-based publish/subscribe networks. In *Proceedings of the 16th International Symposium on Distributed Computing*.

[63] Willard, D. E. (1984). New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394.

[64] Willard, D. E. and Lueker, G. S. (1985). Adding range restriction capability to dynamic data structures. *Journal of ACM*, 32(3):597–617.

[65] Zhao, Y., Sturman, D., and Bhola, S. (1999). Matching events in a content-based subscription system. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*.

[66] Zhao, Y., Sturman, D., and Bhola, S. (2004). Subscription propagation in highly-available publish/subscribe middleware. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*.