# Distribution of Route-Impacting Control Information in a Publish/Subscribe System with Delivery Guarantees

by

Yuanyuan Zhao

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2006

Zvi M. Kedem ______________________________

Daniel C. Sturman ______________________________

© Yuanyuan Zhao

All Rights Reserved, 2006

*To my son, my husband and my parents.*

*For their love and support.*

# Acknowledgment

This thesis tells the technical work I have done for my Ph.D. study. What it does not tell is the journey behind, a journey that I search for the meaning of life and work. The journey is still continuing but I have never been more determined that I will travel down the road I have found and be confident of what I do. This determination has gotten me through the most challenging times before the completion of this thesis. In all these times, my family have been supporting me, encouraging me and motivating me. They are the source of my strength.

I am extremely lucky that I have known two other people besides my family. Professor Zvi Kedem, my advisor at NYU, has been very supportive and patient in letting me find what I want to do and how I want to do it while making sure I stay on course. He is a fatherly figure I respect and love and someone who I enjoy receiving guidance from. Dr. Daniel Sturman, my advisor at IBM, has practically directed me through every phase of the research. Dan has made opportunities for me to work on interesting and challenging problems and has trusted my capabilities and encouraged me. Even in his busiest times, Dan has been available to me whenever it was needed. He is the reason that many good things have happened to me. If I have become a better writer, it is because of him. If I have known better how to achieve, it is because of him. He has shown me by example.

A person I must mention is Dr. Alan Bivens. Except for the many ping pong

games he has beaten me, Alan and I have been on friendly terms. Indeed, Alan is a true friend, a trustworthy friend. I thank him greatly for his psycological support through the hardest times. His friendship I will always value.

The members in the Gryphon team have also helped greatly. Dr. Sumeer Bhola, with whom I have worked together on many interesting problems, has taught me the meaning of technical maturity. His technical impact on me is positive and long lasting. Michael Ward made it possible for me to complete the bulk of the technical work. Dr. Rob Strom has infected me with his passion for research and in many cases demonstrated an approach. I would like to thank Dr. Tushar Chandra for opening many opportunities to me, and Dr. Marc Kaplan for teaching me many skills to solve system problems. Dr. Mark Astley has always been there whenever I needed his help. So have other members of the team.

IBM T. J. Watson Research, as a whole community, is a place I feel lucky to be with. Dr. Guerney Hunt, although I do not have interactions with him on a daily basis, is a person I really look up to. Jim Norris and Mike Frissora have always been generous in offering me their help. Jim has maintained the platform on which I conducted the experiments in this thesis as well as many others for my work. The members of Watson Table Tennis Club continue to add a touch of fun to my time in IBM.

I shall never forget that Dr. Yuanyuan Zhao is a product of New York University - a great school in a great city. Not only have many of the faculty members taught and enlightened me, so have my peers. Dr. Fangzhe Chang, a long time friend from the Capital of China to the Capital of the World, has always lent a helping hand. So have Dr. Arash Baratloo, Chenlei Chang, Min Ding, Dr. Congcun He, Dr. Ayal Itzkovitz, Dr. Hoger Karl, Dr. Bin Li, Hua Wang, Xin Zhang and Tao Zhao. Many thanks to them for being part of my happy time at NYU.

Finally, I would like to thank Professor Arthur Goldberg and Ernest Davis, for being on my committee.

# Abstract

Event-driven middleware is a popular infrastructure for building large-scale asynchronous distributed systems. Content-based publish/subscribe systems are a type of event-driven middleware that provides service flexibility and specification expressiveness, creating opportunities for improving reliability and efficiency of the system.

The use of route-impacting control information, such as subscription filters and access control rules, has the potential to enable efficient routing for applications that require selective and regional distribution of events. Such applications range from financial information systems to sensor networks to service-oriented architectures. However, it has been a great challenge to design correct and efficient protocols for distributing control information and exploiting it to achieve efficient and highly available message routing.

In this dissertation, we study the problem of distributing and utilizing route-impacting control information. We present an abstract model of content-based routing and reliable delivery in redundant broker networks. Based on this model, we design a generic algorithm that propagates control information and performs content-based routing and delivers events reliably. The algorithm is efficient and light-weight in that it does not require heavy-weight consensus protocols between redundant brokers. We extend this generic algorithm to support consolidation and merging of control information. Existing protocols can be viewed as particular encodings and optimizations of the generic algorithm. We show an encoding using virtual time vectors that supports reliable delivery and deterministic dynamic

access control in redundant broker networks. In our system, the semantics of reliable delivery is clearly defined even if subscription information and access control policy can dynamically change. That is, one or more subscribers of same principal will receive exactly the same sequence of messages (modulo subscription filter differences) regardless of where they are connected and the network latency and failure conditions in their parts of the network.

We have implemented these protocols in a fully-functioning content-based publish/subscribe system - Gryphon. We evaluate its efficiency, scalability and high availability.

# Contents

**Dedication** iii

**Acknowledgment** iv

**Abstract** vi

**List of Figures** xiii

**List of Tables** xv

**1 Introduction** 1

1.1 Large-Scale Content-based Publish/Subscribe Systems . . . . . . 2

1.2 Application Scenarios . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2.1 Financial Information Systems . . . . . . . . . . . . . . . . . 4

1.2.2 World-wide Sporting Event Systems . . . . . . . . . . . . . 5

1.2.3 The Smart Energy Network . . . . . . . . . . . . . . . . . . 5

1.2.4 Service-Oriented Architecture . . . . . . . . . . . . . . . . . 6

1.3 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

1.5 Dissertation Outline . . . . . . . . . . . . . . . . . . . . . . . . . . 12

**2 Background and Related Work** **14**

2.1 Distributed Messaging Systems . . . . . 14

2.1.1 Group Communication Systems . . . . . 15

2.1.2 Group/Topic-based Publish/Subscribe Systems . . . . . 17

2.1.3 Centralized Event-based Systems . . . . . 20

2.1.4 Distributed Content-based Publish/Subscribe Systems . . 21

2.2 Peer-to-peer Systems . . . . . 23

2.2.1 Classification of Peer-to-Peer Systems . . . . . 24

2.2.2 Publish/Subscribe over Peer-to-Peer Networks . . . . . 25

2.2.3 Summary . . . . . 26

2.3 Access Control in Distributed Messaging Systems . . . . . 26

**3 A General Algorithmic Model** **30**

3.1 Topology, Content-based Routing and Reliable Delivery . . . . . 30

3.1.1 Routing Topology . . . . . 30

3.1.2 Content-based Routing and Reliable Delivery . . . . . 35

3.2 General Model for Subscription Propagation and Content-based Routing . . . . . 37

3.2.1 Notation . . . . . 38

3.2.2 Consistency of Broker Subscription State . . . . . 40

3.2.3 Content-based Routing Algorithm . . . . . 41

3.2.4 Providing Reliable Delivery . . . . . 44

3.2.5 Summary . . . . . 47

3.3 Generic Subscription Propagation Algorithm . . . . . 48

3.3.1 Subscription Information Maintenance and Exchange . . . 49

3.3.2 Computing Subscription Change for an Upstream Broker . 51
3.4 Subscription Propagation with Aggregation . . . . . . . . . . . . . 52
3.4.1 Aggregating Conjunctions . . . . . . . . . . . . . . . . . . . 53
3.4.2 Operation and Constraint Properties . . . . . . . . . . . . 56
3.5 Subscription Propagation with Filter/Conjunction Merge . . . . . 59
3.5.1 Merging Brokers . . . . . . . . . . . . . . . . . . . . . . . 61
3.5.2 Algorithm Sketch . . . . . . . . . . . . . . . . . . . . . . . 61
3.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

**4 Subscription Propagation Using Virtual Time Vectors 64**

4.1 Protocol Description . . . . . . . . . . . . . . . . . . . . . . . 65
4.1.1 Virtual Times . . . . . . . . . . . . . . . . . . . . . . . . 65
4.1.2 The Conjunction DAG . . . . . . . . . . . . . . . . . . . . 66
4.1.3 Handling New Subscriptions and Assigning Virtual Start Times . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.1.4 Handling Unsubscription . . . . . . . . . . . . . . . . . . . 73
4.1.5 Computing an Incremental Subscription Change Message . 74
4.1.6 Propagating Subscription Changes at An Intermediate Routing Broker . . . . . . . . . . . . . . . . . . . . . . . . 75
4.1.7 Data Message Routing . . . . . . . . . . . . . . . . . . . . 82
4.1.8 Detecting the Subscription Delivery Starting Point . . . . 85
4.2 Mapping the Generic Algorithm . . . . . . . . . . . . . . . . . . 86
4.2.1 Encoding the Subscription Sets $S$ and $S_m$ . . . . . . . . . 86
4.2.2 Encoding the Constraints . . . . . . . . . . . . . . . . . . . 88
4.2.3 Implementing the Sufficiency Test . . . . . . . . . . . . . . 89

4.3 Liveness and Failure Handling . . . . . . . . . . . . . . . . . . . . . 90
4.3.1 Recovery from Broker Crashes . . . . . . . . . . . . . . . . . 91
4.3.2 Recovery from Subscription Message Losses . . . . . . . . . 95
4.3.3 Recovery from Subscription Message Arriving Out-of-Order 97
4.3.4 Recovery from Subscription Message Duplication . . . . . 98
4.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

**5 Dynamic Access Control 99**

5.1 Deterministic Service Model . . . . . . . . . . . . . . . . . . . . . 100
5.1.1 System Entities & Content-based Rules . . . . . . . . . . . 100
5.1.2 Clear Starting Points of Access Control Changes . . . . . . 103
5.1.3 Subscription Propagation, Content-based Routing and Reliable Delivery . . . . . . . . . . . . . . . . . . . . . . . . . 105
5.2 Protocol Description . . . . . . . . . . . . . . . . . . . . . . . . 107
5.2.1 Protocol Overview . . . . . . . . . . . . . . . . . . . . . . 107
5.2.2 Distributing Access Control Information . . . . . . . . . . 110
5.2.3 Handling Client Subscriptions . . . . . . . . . . . . . . . . 112
5.2.4 Propagating Restricted Client Subscriptions . . . . . . . . 114
5.2.5 Handling Dynamic Access Control Changes . . . . . . . . 115
5.2.6 Routing Data Messages . . . . . . . . . . . . . . . . . . . . 117
5.2.7 Enforcing Access Control . . . . . . . . . . . . . . . . . . . 118
5.2.8 Sunsetting of Access Control Rules/Versions . . . . . . . . 119
5.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120

**6 Implementation and Evaluation 121**

6.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
6.1.1 Covering Test . . . . . . . . . . . . . . . . . . . . . . . . 122
6.1.2 Multiple Publisher Hosting Brokers . . . . . . . . . . . . . 124
6.1.3 Sufficiency Test Result Cache . . . . . . . . . . . . . . . . 125
6.1.4 Improving System Concurrency Level . . . . . . . . . . . . 130
6.2 Analytical Model and Results of System Scalability . . . . . . . . 131
6.2.1 Analytical Model . . . . . . . . . . . . . . . . . . . . . . . 132
6.2.2 Communication Overhead Scalability . . . . . . . . . . . . 133
6.2.3 Sufficiency Test Computation Scalability . . . . . . . . . . 134
6.2.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
6.3 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . 136
6.3.1 The Flooding Scheme . . . . . . . . . . . . . . . . . . . . . 136
6.3.2 Micro Benchmark: Content Matching Overhead . . . . . . 137
6.3.3 System Load Comparison in Selective Subscription Tests . 139
6.3.4 CPU Overhead with Access Control Enforcement . . . . . 144
6.3.5 Latency Measurements . . . . . . . . . . . . . . . . . . . . 145
6.3.6 Latency Measurement with Dynamic Access Control . . . 150
6.3.7 Scalability Measurements . . . . . . . . . . . . . . . . . . . 154
6.3.8 Failure Test . . . . . . . . . . . . . . . . . . . . . . . . . . 157
6.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160

**7 Conclusions and Future Work 162**

**Bibliography 165**

# List of Figures

1.1 Publish/Subscribe System . . . . . . . . . . . . . . . . . . . . . . 3

3.1 Redundant Routing Networks . . . . . . . . . . . . . . . . . . . . . 33

3.2 Transforming a Physical Broker Network . . . . . . . . . . . . . . 34

3.3 Message Streams . . . . . . . . . . . . . . . . . . . . . . . . . . 44

3.4 Binary Redundant Spanning Tree . . . . . . . . . . . . . . . . . . 48

3.5 Applying Incremental Changes $\Delta\Lambda_m$ in $b_2$ . . . . . . . . . . . . . 54

3.6 Redundant Routing Network Supporting Subscription Merge . . . 62

4.1 A Redundant Broker Network . . . . . . . . . . . . . . . . . . . . . 69

4.2 State at Broker $SB_1$ After Inserting Subscriptions $s_1$ and $s_2$ . . . . 70

4.3 State at Broker $SB_2$ after Inserting Subscription $s_4$ . . . . . . . . 71

4.4 State at Broker $SB_3$ after Inserting Subscription $s_5$ . . . . . . . . 72

4.5 Broker $B_1$ Subscription DAGs . . . . . . . . . . . . . . . . . . . . 79

4.6 State of Broker $B_3$ after Receiving Subscription Incremental Change from $SHB_3$ . . . . . . . . . . . . . . . . . . . . . . . . . 80

4.7 State of Broker $PB_1$ after Receiving 3 Changes from $N_{21}$ and 1 Change from $N_{22}$ . . . . . . . . . . . . . . . . . . . . . . . . . 81

4.8 Binary Redundant Spanning Tree . . . . . . . . . . . . . . . . . . . 88

5.1 Service Model of Dynamic Access Control . . . . . . . . . . . . . 104

5.2 Redundant Routing Networks . . . . . . . . . . . . . . . . . . . . . 111

5.3 Retrieve Initial Access Control for a New Client/Principal . . . . 113

6.1 Matching Overhead on Subscriptions on Cached Attributes . . . . 139

6.2 Matching Overhead on Subscriptions on Non-Cached Attributes . 140

6.3 Network Topology for System Load (CPU) Comparisons . . . . . 141

6.4 PHB System Load (CPU) Comparison . . . . . . . . . . . . . . . 142

6.5 SHB System Load (CPU) Comparison . . . . . . . . . . . . . . . 143

6.6 Topology Network for CPU Measurements of Access Control . . . 144

6.7 CPU Utilization at Brokers . . . . . . . . . . . . . . . . . . . . . 145

6.8 A Linear Topology Network for Latency Measurements . . . . . . 147

6.9 Delivery Start Latencies and Message Delivery Latency . . . . . . 148

6.10 Topology Network for Latency Metrics with Dynamic Access Control151

6.11 Latency Metrics with Dynamic Access Control . . . . . . . . . . . 154

6.12 A Fan-out Topology Network for Scalability Measurements . . . . 155

6.13 Scalability: PHB CPU Utilization Comparison . . . . . . . . . . . 156

6.14 Scalability: Intermediate Broker CPU Utilization Comparison . . 157

6.15 Scalability: SHB CPU Utilization Comparison . . . . . . . . . . . 158

6.16 Topology Network for Fault Tolerance Test . . . . . . . . . . . . . 159

6.17 Client Message Rate and Intermediate Broker CPU Utilization with Crash Failure . . . . . . . . . . . . . . . . . . . . . . . . . . . 161

# List of Tables

2.1 Features of Distributed Content-based Publish/Subscribe Systems 22

6.1 Sufficiency Test Cache . . . . . . . . . . . . . . . . . . . . . . . . . . 130

# Chapter 1

# Introduction

Event-driven middleware is a popular infrastructure for building large-scale asynchronous distributed systems. The application domains include software systems that are reactive in nature, such as security monitoring/alerting systems and industrial control systems. The paradigm also applies to systems that are not inherently reactive but nevertheless can benefit from the loosely-coupled nature of the event-based communication. As a result, these systems are easier to extend, more flexible to integrate, and of higher reliability.

Compared with traditional event-driven infrastructure such as group communication and topic-based publish/subscribe systems, content-based publish/subscribe messaging offers the most flexibility and expressiveness, but also presents the greatest challeges. Recent research development on stateful or stream-based systems further generalize on the basis of content-based systems to provide subscriptions that can capture even richer application semantics. In these systems, the techniques in content-based systems are usually building blocks and are often generalized to solve problems that are specific in the domain

of stateful systems. This thesis addresses problems arising in content-based publish/subscribe systems, such as scalability, performance, security and availability that are often encountered in the wide-area deployment of large-scale distributed systems. In particular, we examine how protocols using route-impacting control information can enable the building of systems that can address these problems.

## 1.1 Large-Scale Content-based Publish/Subscribe Systems

A content-based publish/subscribe system (Figure 1.1) typically consists of *publishers* that produce messages and *subscribers* that register interest in receiving messages. The interest is usually expressed through content filters in the form of Boolean expressions. The system ensures timely delivery of published messages to all interested subscribers, and typically contains *routing brokers* for this purpose. Publishers and subscribers obtain service by connecting to brokers and are decoupled from each other, since publishers need not be aware of which subscribers receive their messages, and subscribers need not be aware of the sources of the messages they receive.

In a content-based publish/subscribe system, message routing and delivery are usually regulated by control information. This control information includes

- *Subscription:* specifies the kind of data messages that are of interest to certain subscribers in certain parts of the routing network.
- *Advertisement:* specifies the kind of data messages that will be published by a certain publisher or publishers in certain parts of the routing network.

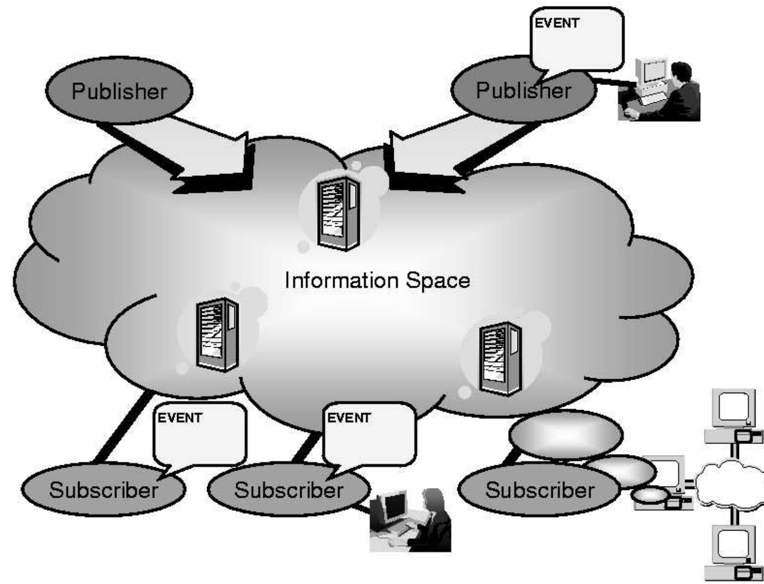



Figure 1.1: Publish/Subscribe System

- *Access control:* specifies the kind of data messages that are allowed to be published or subscribed to by clients of certain principals.

In a content-based system, one or more types of control information are distributed and maintained in order to more efficiently route messages. However, the way in which this control information is handled impacts the scale, performance and even correctness of a system.

## 1.2 Application Scenarios

In order to understand the challenges in content-based publish/subscribe systems, it is necessary to consider the typical application scenarios where traditional methods would be prohibitively expensive in terms of efficiency or usability. We describe in this section several example applications that have motivated our re-

search.[1] These applications typically require stringent service guarantees such as reliable delivery, efficient and scalable message delivery through content-filtering, security mechanisms to prevent unauthorized accesses and high availability toward 24-hour continuous operation while allowing dynamic administrative changes to security policies. Traditional methods usually fail to meet one or more of these requirements.

### 1.2.1 Financial Information Systems

The financial markets, such as the Chicago Mercantile Exchange (CME) and New York Stock Exchange (NYSE), have experienced immense growth in electronic trading over the last few years. The trend is toward 24 hour global electronic trading in a converged marketplace, combining cash, futures and options markets, driving demand for highly scalable and reliable electronic trading capabilities.

As a result, their current trading systems, based largely on proprietary technology, are getting stressed. Several exchanges have been experimenting with open systems based technologies and commodity processing platforms to build new trading engines. The challenge is to show that these engines, far more adaptable and flexible, are just as dependable as the proprietary ones.

The messaging engine, as part of the new enabling technology, is expected

[1] The financial information system example is derived from an internal IBM news report "IBM Research wows the Chicago Mercantile Exchange" published on June 8, 2005. The world-wide sporting event systems example comes from previous deployments of the Gryphon systems in four Grandslam Tennis Tournament and 2000 Olympics in Sydney Australia etc. The smart energy network example comes from related publications [45] at Climate Solutions by Patrick Mazza. Information on Service Oriented Architectures (SOA) and Enterprise Service Bus (ESB) can be found online from IBM and Microsoft etc.

to be able to handle high message rate, deliver messages reliably (in order and without gaps), and provide high availability of close to zero percent of system downtime. The messaging engine is also required to be able to enforce access control with regard to who are allowed to publish information on trades and who (such as paid investors) are allowed to receive financial news. It is also required that the messaging engine support changes in access control policy and provide a well-defined semantics of the change with regard to message delivery.

### 1.2.2 World-wide Sporting Event Systems

Messaging middleware has been used increasingly in major sporting events such as the Olympics and the tennis opens for tasks such as web-based scoreboard and news dissemination. Many of these sporting events are multi-game and are interested by audience from across the world. The amount of information generated in this type of environment is typically great and requires the messaging engine to support large numbers of users. In particular, some games are of regional interest due to time or cultural differences. The messaging engine, as a result, is expected to adapt by not flooding every piece of information to everywhere in the world.

In addition, security requires the messaging engine to fend off unauthorized users from publishing bogus information or from receiving information without paying the required subscription fees.

### 1.2.3 The Smart Energy Network

The recent emergence of cheap computing power and low-cost bandwidth is transforming the traditional power grid into a smart energy network. In such a network, smart appliances and sensors will enable precision power management that coor-

dinates electrical demand down to the residential level by communicating their operating status, collecting information on grid conditions and responding in ways that most benefit their owners and grid. The vast number of sensors and software agents distributed in the field will enable the power grid to automatically check failures and prevent them from cascading through the system. The direct result is a sharp increase of energy efficiency, lower energy bills, fewer blackouts and brownouts and greater resiliency against terrorist attack and natural disaster.

Communication and control is at the forefront of enabling smart energy networks. Distributed messaging middleware provides a promising method to enable the vast amount of appliances, sensors, actuators and controllers to communicate efficiently. The amount of information can be very large, hence in order to avoid overwhelming the network, it is essential to direct information flow to only where it is interested.

### 1.2.4 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is an architectural style of integrating loosely-coupled distributed applications. At the center of this architecture is a collection of services. These services communicate with each other by sending messages through an Enterprise Service Bus (ESB). The communication pattern can be one request to exactly one service provider or one request to many service providers. The one-to-many communication is especially useful in providing high availability of services. The communication can be provided by point-to-point or publish/subscribe messaging.

In an enterprise computing environment, reliability is often a key considera-

tion. The messaging layer of the integration service is usually expected to provide reliable delivery so that service requesters and providers need not worry about the loss of their communications.

Another characteristic of such an environment is the potential large number of service requesters/clients and service providers. To ensure the efficiency of enterprise systems, It is hence important to route the service requests only to the relevant service providers.

## 1.3 Problem Statement

Various types of applications can benefit from the loosely coupled nature of content-based publish/subscribe middleware and its expressiveness and flexibility. In addition, some of these applications require the content-based publish/subscribe middleware to provide more stringent service guarantees such as reliable in-order gapless delivery of messages. Some require the middleware to efficiently deliver among the large amount of message traffic, a selective set of messages that match their interest. Some have very high availability requirement such as 24-hour operation with nearly zero percent downtime. And some require stringent access control. Furthermore, the access control policy should be allowed to change (even though the changes are usually not frequent) without having to disrupt system operation. As a result, the following requirements are usually imposed on content-based publish/subscribe systems:

- Perform content-based filtering and routing throughout the network and as close as possible to the publishers in order to utilize bandwidth efficiently. Flooding, that is, sending every message everywhere, can be very inefficient

in certain environment such as sensor networks or in the case that a new subscription or a disconnected subscription wants to receive message that are published in the past.

- Provide strong service guarantees such as reliable gapless delivery.
- Provide high availability and good load balancing by utilizing redundant routes and easy switching of routes if needed.
- Provide flexible and dynamic access control and message protection.

In order to satisfy these requirements, content-based publish/subscribe systems need to distribute control information across the broker network. Such control information includes client subscription information, access control information and message protection levels (such as message integrity and authenticity). Among these, subscription and access control information can impact content-based filtering and routing. For example, if a subscriber (or subnet of subscribers) has no subscription that matches a message or the principal of a subscriber (or subnet of subscribers) has no right to access the message, there is hence no need to route the message to that subscriber (or subnet). As a result, the message is filtered.

The distribution of control information is essential for implementing content-based routing. However, if this is not done properly, system performance and correctness of reliability could be compromised.

First, in order to perform filtering and routing, a broker stores, for each of its neighboring parts of the network, the information on what messages are needed and can be accessed by subscribers from that part. When a new message is published and routed, a broker filters out and does not send the message to parts of

the network where no subscriber is interested or no subscriber is authorized to access. However, the amount of subscription and access control information could get very large as it approaches the publishers. This is due to that a routing broker is responsible for routing messages to all subscribers in its downstream (defined as the direction away from a publisher). The amount of subscription and access control information maintained for this purpose increases as the number of subscribers increases for brokers in upstream. As a result, a scalable publish/subscribe system should aggregate and only maintain a subset of this information for each routing direction of a broker as long as the subset of information is sufficient to match all messages that are needed and can be accessed by subscribers from that routing direction.

Second, the combination of content-based routing and reliable delivery provides some unique challenges. Unlike in topic or group-based publish/subscribe systems, reliability cannot be based only on detecting gaps in publisher-assigned sequence numbers as each content subscriber may request a completely unique set of messages to be delivered. Reliable delivery protocols typically rely on brokers on the routing path to assist on detecting gaps. A routing broker with incorrect subscription and access control information may decide not to forward on a message. Given that gaps cannot be detected by checking publisher-assigned sequence numbers, an end subscriber may never discover that a message was missed.

Third, loss of connectivity is common in wide-area networks, due to hardware and software failures and network misconfigurations. Hence, publish/subscribe systems should be built on networks with redundant links. This further complicates control information distribution as alternative routes with different subscrip-

tion and access control information may filter out messages matching subscriptions that are unknown to a route.

Fourth, the control information itself can change dynamically as subscribers come and go and administrators grant or revoke access control to/from clients. It is not clear how the dynamics of control information should affect content delivery, especically in the presence of reliable delivery. Due to the dynamic nature, it is also not sufficient to only use access control to restrict client subscriptions upon their entry into the system. Access control information hence should be treated separately from subscription information even though a number of techniques in the latter domain can be re-used and seamless integrated into a coherent solution.

As a result of these challenges, there had not been a solution that guarantees the correctness of content-based routing and hence existing solutions were capable of supporting reliable delivery in the presence of failures and redundant paths. To address this issue, this thesis aims to improve the understanding of the fundamentals of control information distribution (especially subscription propagation) and content-based routing. We address the lack of a coherent theory of how subscription propagation and access control information distribution and content-based routing should work in general. As a result, designing these algorithms typically became disconnected activities each dealing with different situations.

## 1.4 Contributions

This thesis investigates the problem of distributing control information that can impact the routing of events in content-based publish/subscribe systems. In particular, distributing and maintaining this control information enables the pub-

lish/subscribe system to function more efficiently by withholding from sending extra events to parts of the system where they are not needed. This efficiency is provided in conjunction with reliable delivery and high availabilty through redundant routing paths. Hence, it presents a very powerful infrastructure for building commercial applications. The main contributions of this thesis are:

- The investigation of the fundamentals of subscription propagation, content-based routing and reliable delivery. The result is a generic algorithmic model and a super-asynchronous generic algorithm that can utilize redundant paths to achieve high availability. This generic algorithm can serve as the basis to understand existing algorithms and develop future new algorithms. Those algorithms can be interpreted as different encodings and optimizations of the generic algorithm under different circumstances.

- The design of a subscription propagation protocol that utilizes clever instruments to efficiently represent subscription state and event routing requirements. We also provide an interpretation to map this protocol to the generic algorithm and hence the correctnss of the protocol can be derived from the generic algorithm.

- The development of a deterministic service model of dynamic access control with regard to reliable delivery. This deterministic feature is independent from issues like where clients are connected, the message latency and failures in the publish/subscribe system. Under this model, a publish/subscribe system has precise control over event confidentiality: if information was compromised because of incorrect access control policies, the administrator can revoke access to a principal and be sure about precisely what messages got

compromised. The semantics of reliable delivery under this deterministic model is very clear. Concretely, when access control rules for the principal are changed, two or more subscribers of the same principal will receive the same set of messages regardless of which broker the subscribers are connected to.

- The development of an efficient protocol that provides deterministic dynamic access control. Using this protocol, a publish/subscribe system does not have to perform consensus operations across many brokers, which could compromise efficiency of the system and timeliness of enacting the change. In addition, the protocol enables the system to be highly available and fault tolerant.

- The implementation of the above work in a fully-functioning content-based publish/subscribe system - Gryphon. We evaluate its efficiency, scalability and high availability analytically and experimentally.

## 1.5 Dissertation Outline

The rest of this dissertation is organized as follows:

Chapter 2 provides background information on distributed communication middleware, peer-to-peer systems and distributed access control. This information is necessary to understand the differences among the various systems and the distinct characteristics of content-based publish/subscribe systems and the challenges we faced in our work.

Chapter 3 presents a general model that is the theoretical foundation of our

work. It first describes an abstract routing topology and the concept of content-based routing and defines reliable delivery. It then presents the correctness criteria of content-based routing and correctness of subscription propagation as an optimization supporting content-based routing. Based on these correctness definitions, we describe a generic subscription propagation algorithm and an extension of the algorithm utilizing subscription aggregation. The last part of this chapter describes how the algorithm can support subscription merging.

Chapter 4 describes a concrete subscription propagation protocol that can be interpreted as a special encoding of the generic algorithm. It also presents a livenss and failure handling mechanism. The protocol is implemented in the context of the Gryphon system and the implementation is described in detail.

Chapter 5 describes the issue of dynamic access control in content-based publish/subscribe middleware supporting reliable delivery. This chapter includes a model that defines dynamic access control and our deterministic service guarantee. We then describe a protocol that can efficiently support this service guarantees and in steady state, delivers no extra message than that is needed by the clients. Using this protocol, the system provides light failover without compromising the deterministic service guarantee.

Chapter 6 describes the implementation of this work in a fully-functioning publish/subscribe system Gryphon. We describe some of the system challenges we encountered and the solutions we applied. We then present both analytical and experimental evaluation of the system.

Chapter 7 summarizes the work described in this thesis and outlines future directions of research.

# Chapter 2

# Background and Related Work

In this chapter, we present background information on event-based infrastructure. In particular, we describe the different types of event communication systems and their evolution. Presenting background information on these systems is necessary for understanding their characteristics and the challenges one faces in implementing such systems. The overview of peer-to-peer systems describes recent trends and development in network organization and adaptation in case of failures, which is an essential layer for building any distributed system with components communicating with each other.

## 2.1 Distributed Messaging Systems

The evolution of event-based systems has gone from group communication systems to topic-based systems, and then to content-based publish/subscribe systems.

### 2.1.1 Group Communication Systems

Group communication systems are the earliest application-level multicast systems. The model of Virtual Synchrony [16] was implemented in ISIS [15], and later in Horus [74]. Other examples of such software systems include Totem [47] and Transis [25]. These systems typically provide a rich set of reliable delivery primitives such as atomic delivery, causal order [40] or total order.

The core concept in group communication systems is the *group of processes*. In human society, groups have been used as a convenient means for referring to or addressing some part of the population as if it were a single entity. Computation groups were first introduced (in V Kernel [20]) as a syntactic convenience to express one-to-many communication structures and have subsequently been seen as a convenient addressing mechanism. Such facilities are offered in many local-area networks such as IP multicasting and Internet news groups or mailing lists. However, to support mission-critical tasks, further requirements must be satisfied such as the quality of message delivery, and the full benefits of the group concept can be reaped only if we know how to set up and coordinate groups of processes that work together to fulfill a common purpose. Subsequently, the group concept has been extended to include strong guarantees in the presence of failures.

Under the group communication framework, a member in a process group can send a message to the group, and processes interested in certain type of messages can join the corresponding group. The group communication primitives guarantee the delivery of the messages to group members in accordance with the quality of service levels required, for example, causal order or total order.

One essential task of the group communication systems is to maintain group

memberships, that is, maintain the list of processes that are currently in the group. This approach is different from today's publish/subscribe systems as publishers and subscribers are usually maintained anonymously.

Group communication systems are suitable for supporting applications (such as replica maintainence) that require strong consistency guarantees at a small scale (tens to hundreds of members). Group communication systems usually provide a rich set of reliability guarantees such as atomic delivery, causal order and total order. This is different from the application scenarios which publish/subscribe systems aim to support. In these scenarios, the challenge is to efficiently transmit large numbers of messages to as many subscribers as possible and the requirement for consistency is usually weaker. The group communication primitives that provide stringent consistency guarantees would be an overkill and incur unnecessary overhead.

The reliability support in group communication systems is provided to a *known* group of processes. This group of processes, however, is adjustable should a process fail or the network become partitioned. On the other hand, one kind of reliability in publish/subscribe systems guarantees eventual delivery for a client even if the client has temporarily failed or the network become partitioned but the client is not unsubscribed. Messages published while a client is disconnected or failed must be delivered to the client after it reconnects.

Furthermore, the scale of deployment of group communication systems is small compared with that of the publish/subscribe systems, which can be in the range of tens of thousands of clients or even more. The group communication systems thus have limitations in serving the needs of distributed applications in wide area

networks with large number of clients and high rate of generated events. In particular, although hierachies of groups can be used, the group communication systems do not scale to large groups. In addition, messages published are distributed to either all members of the group or none (modulo failure scenarios). This lack of expressiveness and coarse grain specification has caused large overhead in the system as very often, clients are interested in multiple groups and only subsets of messages sent to each group. In this case, additional filtering must be done at the application level in the client side.

### 2.1.2 Group/Topic-based Publish/Subscribe Systems

Early publish/subscribe systems are greatly influenced by group communications. These kinds of systems allow one to many communications and anonymity between communication parties. Topic-based publish/subscribe systems use topics to identify virtual groups and further allow a hierachy of topics and/or regular expressions in topic specifications. Examples of such systems include NDDS(Network Data Delivery Service) [31] from Real-Time Innovations, Inc., TIBCO Enterprise Message Service and Tibco Rendezvous [33], and Scribe [61] as well as standardization activities such as WS-Notification [69, 70, 71] and WS-Eventing [43].

**NDDS** NDDS is a commercial product for distributed, real-time application development that uses the real-time publish/subscribe model. It provides real-time message delivery service by allowing applications to specify how long a message should stay valid, how important a message is with regard to messages from other publishers of the same topic and how long a subscriber is willing to wait for the next published message. It provides reliable delivery within real-time constraints.

**SCRIBE** Scribe [61] is a large-scale and fully decentralized event notification system built on top of Pastry - a peer-to-peer object location and routing substrate overlayed on the Internet. It leverages the scalability, locality, fault-resilience and self-organization properties of Pastry. However, Scribe does not support content-based routing and wild card topic subscriptions, as the creation and subscription of a topic are explicitly associated with a rendezvous point - a special node with a nodeId numerically closest to the topicId of the topic. The system builds separate multicast trees for individual topics using a scheme similar to reverse path forwarding and inverts the subscription message path for later event distribution. This makes it impossible to add a node to the multicast tree for load sharing. The system recovers from multicast node failures by building new trees. It does not support reliable delivery, and unsubscription has to be delayed until the first event is received.

**TIBCO Rendezvous & Enterprise Message Service** TIBCO Rendezvous [33] is a topic-based publish/subscribe system that utilizes the multicasting capability of the underlying infrastructure. TIBCO Enterprise Message Service [33] employes a store-and-forward architecture and supports publish/subscribe messaging as well as queue-based messaging. It provides scalability and fault-tolerance via server-clustering and routing.

**WS-Notification & WS-Eventing** WS-Notification [70, 71] and WS-Eventing [43] are standardization activities on topic-based publish/subscribe messaging systems. Their goal is to standardize the roles, terminology, concepts, message exchanges and the Web Services Description Language (WSDL) [26]

needed to express the notification pattern, and to provide a language to describe Topics. Through standardization, topic-based publish/subscribe systems can achieve

- interoperation between NotificationProducers and NotificationConsumers;
- interoperation between middleware providers;
- standardized mechanism to develop Topic taxonomies;
- standarized concepts and terminology.

As standardization activities, WS-Notification and WS-Eventing address general issues related to interoperability of publish/subscribe systems without getting into details of specific system implementation details.

**CORBA Event Service** The standard CORBA operation invocation model supports two way, one way and deferred synchronous interactions between clients and servers. To facilitate asynchronous message exchange, CORBA [1] designed its Event Service [30]. The Event Service defines supplier and consumer participants, and allows asynchronous event delivery from suppliers to consumers in a many-to-many fashion. In addition, CORBA also defines event channels which mediate the delivery of events from suppliers to consumers without requiring them to explicitly know each other. There are two models of supplier-consumer collaboration through the event channel, push and pull, in which the supplier and the consumer take the active role respectively. The push model is more commonly used because of its efficient and predictable execution of operations.

The CORBA Event Service can also be extended to support fault tolerant computing by allowing replication of suppliers or consumers and distributed group

communication [44]. This is largely simplified by the fact that the Event Channel decouples the suppliers from the consumers so that the adding or removing of a supplier/consumer does not concern the other side.

**Summary** In summary, group/topic-based publish/subscribe systems have limited expressiveness in subscription description, even though many such systems enable wild cards and more complicated patterns in topic specification in subscriptions, message selectivity is restricted to the level of topics. A published message is always sent to all members of the topic specified in the message even though only a subset of the members are interested in the message. This can result in scalability and performance issues. As with group communication systems, the coarse grain specification often requires additional filtering at the application level in the client side. In addition, some systems do not provide reliability guarantees.

### 2.1.3 Centralized Event-based Systems

Elvin [63] and Yeast [37] are the representatives of centralized event-based systems.

Yeast is an event-action system. It allows clients to define an event-pattern and actions to perform when the pattern is matched. It defines a rich language that allows detailed event patterns, including temporal expressions. The *action* is specified in any language the centralized server can execute. Although the language is rich, the system does not address how to efficiently find matched event patterns.

Elvin is mainly designed around a single server that filters and forwards producer messages directly to consumers. It supports a quenching functionality where if there is no subscriber interested in messages published by a specific publisher,

publication at the publisher is supressed by sending a quenching request to the publisher. Functionality-wise, this is analogous to pushing subscription information and content-based filtering and routing beyond the edge brokers and all the way to the publishers.

Centralized infrastructure has major drawbacks in that it cannot offer a scalable solution for large deployments and the centralized server is often a bottleneck and a single point of failure.

### 2.1.4 Distributed Content-based Publish/Subscribe Systems

Distributed publish/subscribe systems usually consist of a network of brokers. In these systems, publishers and subscribers connect to a broker of choice to receive service and messages are delivered by transmitting through one or more brokers. Examples in this category include Siena [17], XNet [18], REBECA [48, 49] and Hermes [53].

In order to perform content-based filtering in a distributed environment, these systems usually propagate subscription information throughout the brokers. To achieve scalability, subscription aggregation has been used such as in [17], [18], [49]. Some systems also utilize topologies with redundant routes between servers to provide better availability such as Siena [17], XNet [18], SonicMQ [32]. Reliability is sometimes provided in these system as well such as in JEDI [23] and SonicMQ [32]. Table 2.1 summarizes the features of these systems.

There are some recent works on utilizing the subscription pattern locality to improve system performance. Wang et al [77] have proposed an approach to partitioning existing subscriptions and routing new subscriptions among multiple

Table 2.1: Features of Distributed Content-based Publish/Subscribe Systems

| Systems | Reliable delivery | Redundant Path | Subscription Propagation |
|---:|---:|---:|---:|
| Siena | No | No | Yes |
| XNet | No | No | Yes |
| Rebeca | No | No | Yes |
| JEDI | Yes | No | No |
| SonicMQ | Yes | Yes | No |
| Snoeren et al. | No | Yes | Yes |

servers to optimize various performance metrics including total network traffic, load balancing, and system throughput. They propose two approaches, one based on partitioning the event space and the other based on partitioning the subscription set. However, their work is in a static environment where subscription patterns and distributions are known in advance. In addition, they do not take into account the effect of event rate distribution. That is, different events may happen at different rate. This has limited the scope of their work.

Virgillito [75] described an approach to dynamically reconfigure a publish/subscribe broker network according to the similiarity of subscriptions they host. However, they did not address how similarity is monitored and captured in a distributed and dynamic environment. Their reconfiguration algorithm is also limited in that they do not consider network resource restrictions.

**Summary** Existing content-based publish/subscribe systems either do not address the issues of subscription propagation and organization, or do not operate

in networks with redundant links. Our work in this area complements their works and can be applied in these systems.

## 2.2 Peer-to-peer Systems

Like many other distributed systems, content-based publish/subscribe are built on a network substrate. The network layer provides routing and connectivity support and may be self-adapting and self-organizing. Content-based publish/subscribe utilizes the abstraction provided by the network layer. Optionally, the network layer can provide call-back support so that upper layers can get notified in case a topology changes. We introduce here a brief discussion of a recent trend of application-level network - peer-to-peer networks.

Contrary to the traditional client-server architecture, a peer-to-peer (P2P) computer network is a network that relies on the computing power and bandwidth of the participants in the network rather than concentrating it on a relatively few servers. P2P networks are typically used for connecting nodes via largely ad hoc connections. Such networks are useful for many purposes such as sharing content, sharing storage [39] and sharing computing resources to solve large-scale computation problems [64].

In the center of P2P systems is the concept of equal peers self-organized into network topologies. Every node in the system is a client and at the same time, a server. They consume as well as contribute resources of/to the system. The peers/nodes in P2P systems must be able to deal with instability, transient populations, fault tolerance and self adaptation. There is sometimes a claim that since there is no distinguished node in P2P systems, these systems are intrinsi-

cally highly fault tolerant and possess good performance and scaling properties. This has some truth to it, although there are a lot of P2P systems with more distinguished nodes (super-nodes) than others and fault tolerance and scalability do not come without significant effort [22].

Even though the concept of peers can be traced back to an earlier time in distributed computing and networking, such as NetNews NNTP protocol and IP protocol, the new wave of P2P systems are built on the application layer overlay networks. Some of these overlay networks provide fault tolerance by routing around the faulty nodes [5].

### 2.2.1 Classification of Peer-to-Peer Systems

Androutsellis-Theotokis and Spinellis [6] classify P2P systems in terms of their centralization and structure.

- Purely Decentralized Architectures, in which all nodes perform exactly the same tasks, acting both as servers and clients, and there is no central coordination of their activities.
- Partially Decentralized Architectures, in which some nodes are designated as *super-nodes*, serving central indexes for files shared by local peers. This designation is dynamic and super-nodes can be easily replaced if they fail.
- Hybrid Decentralized Architectures, in which there is a central server facilitating the interaction between peers by maintaining directories of metadata, describing the shared files stored by the peer nodes. End-to-end interaction usually occurs directly between peer nodes, without the intermediation of the central server.

- Unstructured network, in which the placement of content is completely unrelated to the overlay topology.

- Structured network, in which the overlay topology is tightly controlled and files are placed at precisely specified locations.

- Loosely-structured networks. A network sits in between of structured and unstructured networks.

Napster is an unstructured network using a centralized directory service to facilitate information lookup. The peers then establish direct connections without centralized server involvement. The central directory server is hence a reliability and performance bottleneck. It is also a single point for security, political, legal and economic attacks.

Gnutella [60] and Freenet [21] are decentralized unstructured systems. Gnutella dispensed with the central directory and replaced it with a flood based search. To mitigate the bad network load effect, systems like Freenet cache the lookup results and route similar future requests according to the cache.

Pastry [62], Chord [66], CAN [57] and Tapestry [81] are structured decentralized systems that use Distributed Hashtable (DHT) to store the directory. They are capable of dealing with the poor resilience and network load caused by super-nodes and flooding.

### 2.2.2 Publish/Subscribe over Peer-to-Peer Networks

Peer-to-peer systems have been used as routing substrates for building application-level multicast systems [58, 9, 59], publish/subscribe systems such as Scribe [61],

Hermes [54], and works described in [50, 68, 24, 72, 79, 80] and information retrieval systems [73, 42].

Most of these systems are group/topic-based (such as Scribe [61]) with groups/topics mapped to rendevous nodes in the underlying peer-to-peer systems. Some (such as [68]) support content-based routing by creating indices along the most popular attributes and simulate content-based routing through groups on value of index digests. Information dissemination/retrieval in these systems is thus transformed into straightforward peer-to-peer routing with reverse-path forwarding or building of multicast trees.

### 2.2.3 Summary

Peer-to-peer systems provide mechanisms for robust and adaptive routing. Systems built on top of peer-to-peer networks can leverage the fault tolerance and self-organization properties of the routing substrate. Content-based publish/subscribe systems are a higher-level abstraction that can be built on peer-to-peer networks. As will be presented later in this thesis, our work on control information dissemination with support of reliable delivery is described on an abstract routing layer whose topology can change due to failures or system load changes. The organization and adaptation of our routing layer can be provided by either peer-to-peer networks or any other methods.

## 2.3 Access Control in Distributed Messaging Systems

Security is indispensible in the wide deployment of applications built on distributed messaging infrastructure. It is vital to protect the system from various

attacks such as malicious publishers flooding the infrastructure with bogus events (Denial-of-Service attack), malicious subscribers inserting very general subscriptions to cause large amount of events being transmitted through the network but only to be discarded (also Denial-of-Service attack) and various man-in-the-middle attacks (event confidentiality, authenticity and integrity).

The greatest volume of work on security issues in distributed messaging systems is in the secure group communication area [35, 38]). In secure group communication systems, access control is usually provided by using a shared key among group members. To deal with group member joining and leaving, and to protect information from the leaving members, keys must be changed. The focus of the works in this area is on group key management ([56]), such as utilizing hierarchical groups [14, 67]).

As pointed out by Opyrchal et al. [52], the dynamic nature of a content-based system makes the secure group communication approach infeasible for enforcing access control in a content-based system. That is, when there are $n$ subscribers, events in a content-based system can potentially go to $2^n$ different groups. Managing keys for these $2^n$ groups is expensive. In addition, the matching groups can change for each event, constantly changing encryption keys significantly slows down the throughput of common encryption algorithms such as DES [55]. Instead, Opyrchal et al. tackles the problem as a group communication problem with very dynamic groups and utilizes group clustering and key caching to achieve better encryption throughput from brokers to subscribers.

Wang et al. [76] analyze security issues and requirements in Internet-scale publish/subscribe systems and presents directions to possible solutions to the various

problems. They presented novel security problems of information and subscription confidentiality in an untrusted publish/subscribe system and pointed out that methods on *computing with encrypted data* [3] and *secure circuit evaluation* [2] can be adapted to solve these problems. In their work, there is no discussion on how access to particular events can be controlled and enforced.

Belokosztolszki [11] presented a role-based model for access control [27] in content-based publish/subscribe systems. They integrate the OASIS [8] role-based access control system into the Hermes publish/subscribe middleware framework and point out that access control can be enforced as restrictions on the subscription filters. By leveraging the existing publish/subscribe platform, access control rules can dynamically change and be distributed to brokers that host clients. Bacon [7] extends the work to multiple trusted domains.

Miklos [46] devotes significant attention to specifying maximum and minimum security restrictions by ways of covering relations between filters, advertisement and events. The intuition is to use maximum security to restrict clients from accessing events they are not authorized and use minimum security to limit the overhead of doing too much content matching against too specific subscriptions.

Srivatsa and Liu [65] propose using keys, signatures and security guards to provide information confidentiality, integrity and authenticity and to fend off denial-of-service attacks.

Yan et al. [78] proposes a security framework for distributed brokering systems. The framework provides secure communications over insecure links, and ensure that only authorized entities are allowed to view entity interactions. They also presented details of their prototype implementation.

**Summary** Existing work on access control in distributed messaging systems has focused on secure distribution of events. There has not been work on the semantics of dynamic access control with regard to reliable delivery. Our work in this area addresses this issue. Existing work on the secure distribution of events according to their access control rules are complementary to our work.

# Chapter 3

# A General Algorithmic Model

In this chapter, we describe the theoretical foundation of our work. This includes an abstract routing topology model; a general model of subscription propagation, content-based routing and their correctness in terms of supporting reliable delivery; and a generic algorithm of propagating subscription information to be used by content-based routing.

## 3.1 Topology, Content-based Routing and Reliable Delivery

We first describe a topology model of redundant routing networks and content-based routing and reliable delivery in such networks.

### 3.1.1 Routing Topology

The architecture and organization of the underlying network on which a publish/subscribe system is deployed affects the correctness, high availability and

effectiveness of message delivery of the system. For example, it is common for the network to contain more than one route between any two communication end points. As a result, the system can switch from one route to another in case one route fails or becomes overloaded. However, complex network topology presents great challenges to routing such as how to avoid infinite loops and which link to use for sending a message and whether the link choice will not break the correctness of reliable delivery. It is hence very important to provide an abstract topology model that preserves the redundant routes and at the same time enables simple routing choices.

A typical physical publish/subscribe network contains one or more interconnected software processes called *brokers*. The brokers cooperate in relaying the messages through the network. In a network configured for high availability, it is common to have more than one route between two brokers and the physical network can be an arbitrary graph.

We adopt an abstract topology model of spanning trees of *nodes* where each node may include multiple *brokers* that are highly connected.[1] By *highly connected*, we refer to the existence of many intra-node broker-to-broker links such that intra-node communication can be done cheaply and easily. Hence, the brokers in the same node may be considered redundant and can work interchangeably. A local area network of brokers is such an example of a highly connected tree node of brokers.

In this topology, any pair of virtual brokers each from an adjacent tree node are connected. Furthermore, the clients may only connect at the leaf nodes and a broker may either accept clients or route broker-to-broker traffic. Each leaf

[1] A tree node is a concept same as the cellules [36] in the Gryphon routing topology.

node may contain only one broker. As a result, we use the term *broker* and *node* interchangeably when we discuss an edge broker and node.

By abstracting a complex network topology into a simpler noncyclic structure like a tree, loop-free routing can be easily achieved by considering only the tree nodes and edges. At the same time, by allowing *multiple redundant* brokers in each tree node, the abstract topology preserves the high availability feature of the original routing network.

The mapping from a complex physical routing network to this abstract redundant tree is assisted by the concept of *virtual brokers*. A virtual broker is a physical broker's presence in a tree node in the abstract routing model. If a physical broker appears in a tree node, the physical broker is said to be implementing a virtual broker in that tree node. In particular, if a physical broker both accepts clients and routes traffic to another physical broker, without loss of generality, the physical broker can be represented by two virtual brokers, one in a client-connecting leaf node and one in a non-leaf node.

The use of virtual broker allows the same physical broker to appear in more than one tree node while preserving an efficient routing implementation. For example, if a physical broker $b$ appears in two adjacent tree nodes $p(arent)$ and $c(hild)$, routing a message from $p$ to $c$ is immediate without the need to send the message through a physical link if the message is in the virtual broker of $b$ in node $p$.

The decisions of what nodes a tree should contain, how the nodes are connected, and which physical brokers belong to each tree node are typically made in an administrative process. The set of virtual brokers of each tree node can be

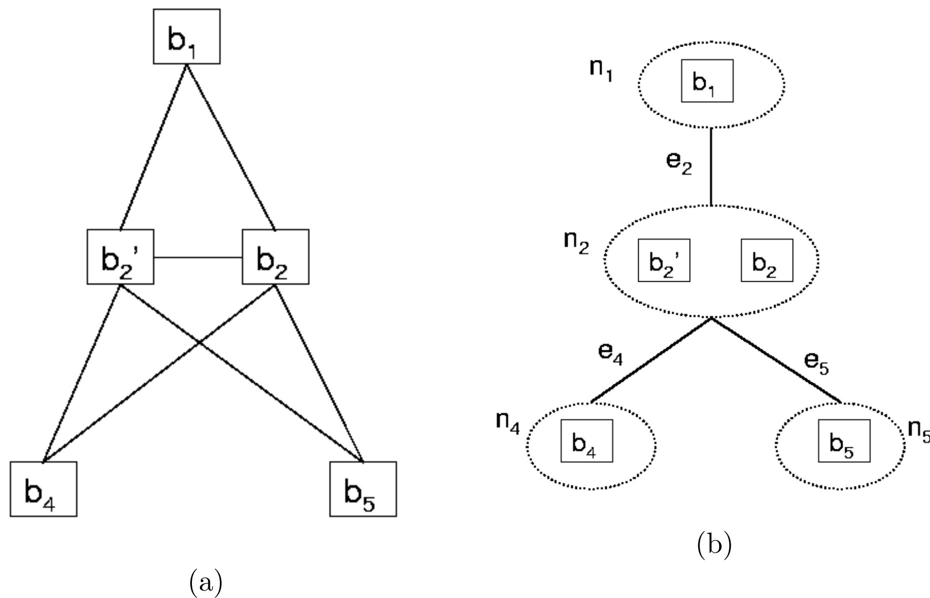


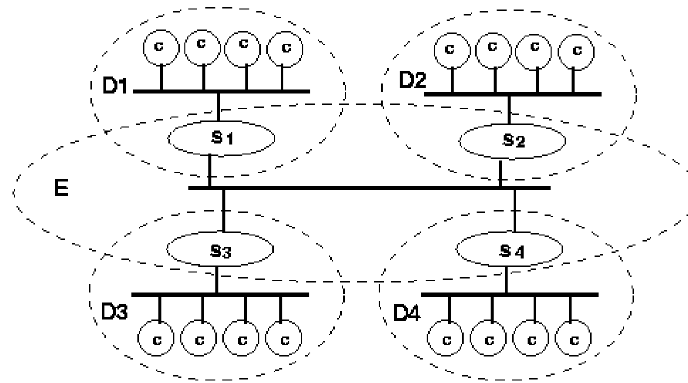
Figure 3.1: Redundant Routing Networks

computed using this administrative information. An algorithm performing this computation is shown in [36].

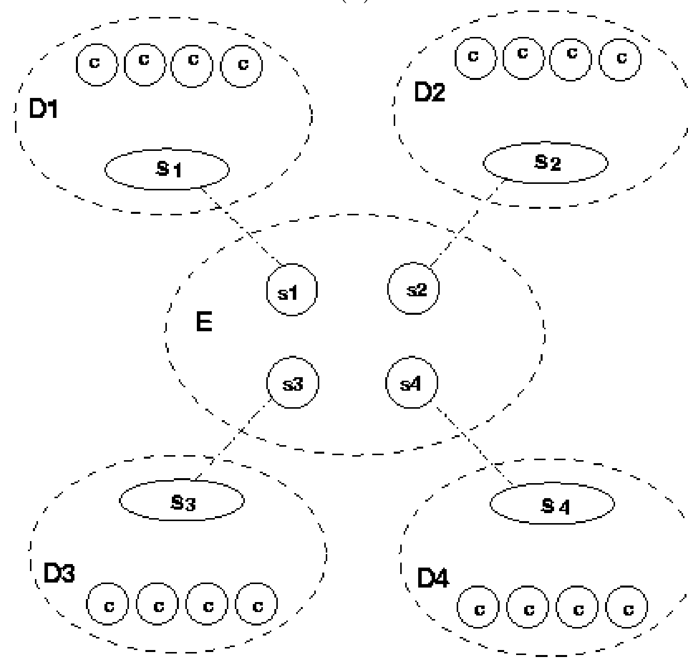
For the purpose of our thesis, it is sufficient to show that our abstract topology model can accomodate a large set of physical broker networks and thus is of practical use. For example, one can transform a graph (e.g., Figure 3.1(a)) with redundant paths into a topology (Figure 3.1(b)) under this model by grouping collections of highly-connected brokers into nodes and inter-broker links into edges between the nodes the brokers reside. A spanning tree can then be defined on such a graph of fat nodes and edges. Figure 3.2 describes another example where each physical broker $s_1$, $s_2$, $s_3$, $s_4$ appears in two tree nodes and hence is represented by two virtual brokers in the transformed abstract topology.[2]

As a result of the foregoing discussion, for the rest of this thesis, we will only

[2]This example is per courtesy of Marc A. Kaplan




(a)




(b)

Figure 3.2: Transforming a Physical Broker Network

use network topologies under this abstract model without distinguishing virtual from physical. Content-based routing thus only need to be executed at the tree node level so to make decisions on whether a message should be forwarded to a "next hop" tree node. We assume the existence of a routing substrate that finds

for the next hop tree node a virtual broker and its physical implementing broker as the destination. As the next hop tree node may contain more than one broker and its incoming edge may contain more than one link leading to these brokers, high availability can be provided by allowing the routing substrate to choose any of these virtual brokers.

We refer to a virtual broker where publishers connect as a publisher hosting broker (PHB) and a virtual broker where subscribers connect as a subscriber hosting broker (SHB). A publishing client may connect at any of the leaf brokers. However, for simplicity, we discuss our work from the standpoint of one PHB. As a result, we can assume, without loss of generality, that SHBs reside in the leaf nodes of the tree; and there is only one PHB and the node it resides in is designated as the root of the tree. We refer to where the PHB resides as *upstream* and direct the edges to point from upstream to *downstream*. We also use the term *edges* of a broker to refer to the edges of its residing tree node.

### 3.1.2 Content-based Routing and Reliable Delivery

A valid implementation of content-based subscription can be one in which the PHB and intermediate brokers forward all published messages to the SHBs, and only the SHBs apply filtering. Such a solution will be a perfectly correct implementation, but it may waste considerable bandwidth sending messages that will be later discarded. Subscription propagation is an optimization which may result in fewer wasted messages being sent to the SHBs in exchange for requiring the PHB and the intermediate brokers to perform filtering and to acquire knowledge about subscription predicates. Since subscription propagation is an optimization, it

should preserve the correctness properties of the original specification of content-based routing.

In our topology mode, data messages enter a publish/subscribe system through the PHB at the root of the redundant routing tree. The PHB maintains a *stream* for data messages published by a publisher. The messages enter the stream in the order they are published. A data message flows downstream toward the SHBs in the leaf nodes. When the message arrives at a broker, the broker performs content matching for each of its outgoing edges. If the data message does not match any subscription in the downstream of the edge, the data message can be filtered out for the downstream of the edge. Otherwise, the broker sends the data message to a broker in the child node on the other end of the edge. As previously mentioned, this step is assisted by asking the routing substrate to choose a broker from the redundant peers in the child node. When the data message arrives at a SHB, the SHB finds the matching subscriptions and delivers the data message to the corresponding subscribers.

The data messages routed by a publish/subscribe system may be subject to various levels of service guarantees such as best effort or reliable delivery. In this thesis, we define reliable delivery as a service level that guarantees for a subscription $s$ and a published message stream $PubStr$, the publish/subscribe system finds a starting message and an ending message (upon the unsubscription request of $s$), and delivers all messages in the range of [starting point, ending point] that match $s$ in an order conforming to the original stream $PubStr$. We require that the starting point be chosen within a finite amount of time if the system has bounded latency and runs without failure for sufficient amount of time. This

excludes trivial algorithms such as one that delays choosing the starting point until it sees the unsubscription request, and delivers no message.

Providing reliable delivery is a challenging task in a content-based system deployed over a network with redundant paths. Due to content-based routing, gaps can not be detected by traditional methods such as using publisher-assigned sequence numbers because each subscriber may have a subscription filter requiring a completely unique sequence of messages to be delivered. Reliability in a content-based system hence requires brokers on the routing path to assist in gap detection ([12]).

Multiple paths, communication asynchrony and failures complicate subscription propagation as redundant brokers on alternative routes may have subscription information that is different from each other's. If messages from the same published stream are routed through those brokers, they are matched to different sets of subscriptions and gaps can appear in the message sequences delivered to subscribers.

## 3.2 General Model for Subscription Propagation and Content-based Routing

In this section, we introduce a general model for subscription propagation and content-based routing. We present the correctness criteria of subscription propagation and a set of sufficient conditions for correct content-based routing and reliable delivery.

### 3.2.1 Notation

In a publish/subscribe system, the brokers route messages for subscriptions that are submitted by subscribers at the leaf nodes and can stop routing messages for subscriptions that are unsubscribed. Hence, there are two sets of subscriptions in the systems: the set of subscriptions that have been subscribed, denoted as $\mathcal{S}$, and the set of subscriptions that have been unsubscribed, denoted as $\overline{\mathcal{S}}$. These two sets of subscriptions represent the global state in the whole system and are always monotonically increasing as time passes. As is common in asynchronous distributed systems, this global state is distributed among the leaf node brokers and the whole state can only be known to an *oracle*. We introduce these concepts for correctness reasoning. They are not maintained by brokers.

We assume a subscription has a unique identity and a filter in the form of a disjunction/set of conjunctions. This does not limit the power of our model as every Boolean expression can be transformed into disjunctive normal form (DNF). We use the mathematical symbol $\lambda$ to denote an individual conjunction and $\Lambda$ to denote a conjunction set. We use operator $conj(s)$ to denote the conjunction set of a subscription $s$. Even though multiple subscriptions can have the same set of conjunctions, the identities are different.

As previously discussed (Section 3.1.1), without loss of generality, we can assume that a subscription only enters the system through a leaf node in the spanning tree. In addition, there is exactly one such leaf node for any given subscription. This assumption is made so that in our system, there is always a deterministic answer to where the home/hosting broker of a subscription is. If it is decided that a message matches this subscription, there is a deterministic

answer as to where the message should be routed to.

An implication of this assumption is that if a subscription moves from one broker/leaf node to another, it is deemed as two different subscriptions. Mobility of subscriptions is supported by letting the second subscription have a delivery starting point that is exactly where the first subscription has left. This can be achieved cooperatively by the messaging system and the application. For example, the JMS specification provides message delivery and acknowledgement modes that allow the applications to receive messages with one or more duplicates in case of a failure or subscriber disconnects or reconnects. This can be extended to have an extra layer that checks (for example, by timestamps) and filters out the duplicate messages and thus guarantees a second subscription receives messages from exactly where the first subscription has left. Furthermore, higher level service can hide the details of this technique of two subscriptions and provide to the application the abstraction of a single but mobile subscription.

We have pointed out that sets $\mathcal{S}$ and $\overline{\mathcal{S}}$ represent global state in an asynchronous distributed system, hence, the brokers can only maintain local and partial knowledge of $\mathcal{S}$ and $\overline{\mathcal{S}}$. These are maintained in the form of subscription identities. In addition, the brokers maintain a set of conjunctions for purpose of content matching and routing for these subscriptions. This local knowledge is accumulated as subscription changes are propagated. In an asynchronous system, this local knowledge usually lags behind the real global values to various degrees in different brokers. As $\mathcal{S}$ and $\overline{\mathcal{S}}$ are monotonically increasing, these local sets are subsets of $\mathcal{S}$ and $\overline{\mathcal{S}}$, respectively. We define these local sets in the following.

**Definition 3.1.** *For broker $b$ and one of its outgoing edges $e$, $b$'s* **matching set**

**of subscriptions** *at e, denoted as* $S(e)$*, represents its knowledge of the set of client subscriptions that are downstream of edge e. Similarly, b's* **matching set of unsubscriptions** *at e, denoted as* $\overline{S}(e)$*, represents its knowledge of the set of client unsubscriptions in the downstream of e.*

**Definition 3.2.** *For broker b and one of its outgoing edges e, b's* **matching set of conjunctions** *at e, denoted as* $\Lambda(e)$*, are the set of conjunctions b maintains to match and route data messages to the downstream of e.*

Although $S(e)$ and $\overline{S}(e)$ are maintained as the list of unique identities of the subscriptions, as shown in Chapter 4, a system may choose a concise representation versus enumerating every subscription.

### 3.2.2 Consistency of Broker Subscription State

Intuitively, a broker's state is consistent if the set of conjunctions (i.e., $\Lambda(e)$) it uses for matching and routing is "sufficient" for the set of subscriptions (i.e., $S(e) - \overline{S}(e)$) it routes message for. We now formally define this sufficiency requirement after introducing the concepts of *covering* of conjunctions, conjunction sets and subscription sets.

**Definition 3.3.** *Conjunction* $\lambda_2$ *covers* $\lambda_1$*, denoted as* $\lambda_2 \succeq_c \lambda_1$*, if and only if* $M(\lambda_2) \supseteq M(\lambda_1)$*, where* $M(\lambda)$ *is the set of all messages matching* $\lambda$*. Conversely,* $\lambda_1$ *is covered by* $\lambda_2$*, denoted as* $\lambda_1 \preceq_c \lambda_2$*.*

**Definition 3.4.** *Conjunction set* $\Lambda_2$ *covers* $\Lambda_1$*, denoted as* $\Lambda_2 \sqsupseteq_c \Lambda_1$*, if and only if* $\bigcup_{\lambda_2 \in \Lambda_2} M(\lambda_2) \supseteq \bigcup_{\lambda_1 \in \Lambda_1} M(\lambda_1)$

We define the covering relationship between a conjunction set $\Lambda$ and a subscription set $S$ in terms of the covering relationship between $\Lambda$ and the set of

conjunctions of all subscriptions in $S$. We use the same operator $\sqsupseteq_c$ as the conjunction set covering relation for this covering relationship between a conjunction set and a subscription set.

**Definition 3.5.** *Conjunction set* $\Lambda$ *covers subscription set* $S$*, denoted as* $\Lambda \sqsupseteq_c S$*, if and only if* $\Lambda \sqsupseteq_c \bigcup_{s_i \in S} conj(s_i)$*.*

The consistency of a broker's subscription state is hereby defined as an invariant each broker should maintain.

**Invariant 1.** *For every broker* $b$ *and its outgoing edge* $e$*, the* $\Lambda(e)$ *set of* $b$ *covers* $S(e) - \overline{S}(e)$*, that is,* $\Lambda(e) \sqsupseteq_c S(e) - \overline{S}(e)$*. We call this the* $S$-$\Lambda$ *invariant.*

Intuitively, this invariant requires that if a broker claims to route messages for a set of subscriptions, the filters (conjunction set) it uses must be sufficient to match all the messages for the subscriptions.

### 3.2.3 Content-based Routing Algorithm

We describe in this section a content-based routing algorithm under the framework we have setup earlier in this chapter.

In the abstract topology model, each broker network is a redundant routing tree. A data message $m$ flows from the root to the leaves in the tree through the connecting edges. The data message carries a *subscription header* that is initially set by the PHB to be equal to $S(e_i)$ the PHB maintains for each outgoing edge $e_i$ of its residing node. This header identifies the set of subscriptions the message is required to be matched against. We call this the *matching set of subscriptions* of $m$, denoted as $S(m)$. The value of $S(m)$ does not have to stay the same through

the whole routing process. A broker $b$ on the routing path of $m$ can modify $S(m)$ by adding subscriptions (in $b$'s matching set of subscriptions) to $S(m)$. Adding a subscription $s$ to $S(m)$ is subject to the constraint that $conj(s)$ must be covered by the conjunctions of subscriptions in the original $S(m)$.

The data message is routed by travelling from one node to its child nodes through the connecting edges. For simplicity of discussion, we assign a pseudo incoming edge to a root node where the PHB resides. The messages enter the broker network through this pseudo edge. Thus, each tree node has one incoming edge $e$, and one or more outgoing edges $e_i$. When a broker $b$ within a node receives $m$ through its incoming edge $e$, it separates out the subscriptions in $S(m)$ that are relevant to each of its outgoing edges. This can be achieved by associating each subscription entity with information on where it subscribes. We call this a *projection* of the $S(m)$ set on the outgoing edge, and denote it as $S(m, e_i)$, where $e_i$ is an outgoing edge.

The broker then checks whether it has knowledge of all the *active* subscriptions in $S(m, e_i)$. An *active* subscription is one that has not unsubscribed, i.e., not in $\overline{\mathbb{S}}$. Hence a *sufficiency test* is performed for $S(m, e_i) - \overline{\mathbb{S}}$.

**Definition 3.6.** *A broker $b$ is sufficient for message $m$ on outgoing edge $e_i$ if $S(m, e_i) - \overline{\mathbb{S}} \subseteq S(e_i) - \overline{S}(e_i)$.*

As mentioned before, $\overline{\mathbb{S}}$ may only be known to an oracle. However, since $\overline{S}(e_i)$ is always a subset of $\overline{\mathbb{S}}$, $S(m, e_i) - \overline{\mathbb{S}} \subseteq S(m, e_i) - \overline{S}(e_i)$. The sufficiency test can be replaced by a stricter test at broker $b$: $S(m, e_i) - \overline{S}(e_i) \subseteq S(e_i) - \overline{S}(e_i)$.

If there is a subscription $s$ such that $s \in S(m, e_i)$ but $s \notin S(e_i)$ and $s \notin \overline{S}(e_i)$, the sufficiency test fails for edge $e_i$. This indicates that broker $b$ lacks information

on $s$. In this case, as a conservation approach, $b$ may forward $m$ to a broker at the other end of edge $e_i$.

If the sufficiency test passes for edge $e_i$, broker $b$ matches $m$ against $\Lambda(e_i)$. If there is a match, $m$ is forwarded to a broker at the other end of $e_i$. If there is no match, $b$ may filter $m$ out. As we mentioned before, providing reliable delivery in a content-based system requires brokers on the routing path to assist in gap detection, hence broker $b$ needs to forward information on data message $m$ to indicate that the message is filtered out to distinguish it from a gap caused by message reordering or losses. We refer to a filtered out message as a *silence* message. For the purpose of the relevant downstream, this silence due to filtering is the same as a silence resulted from the elapse of time between two consecutively published messages. A silence message usually incurs less communication overhead than the original data message $m$ and may be combined with adjacent silence or data messages such as described in [12]. However, for the purpose of this thesis, we do not discuss such optimizations.

We conclude that a correct content-based routing algorithm is one where each broker only filters out a message for an outgoing edge if the sufficiency test passes and there is no match. In support of reliable delivery, such a content-based routing algorithm forwards a *silence* message in place of a filtered message. We call such an algorithm *sufficiency-directed content-based routing*.

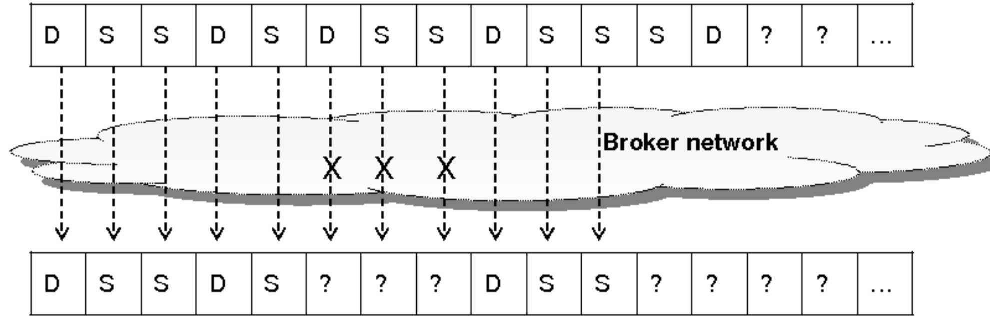



Figure 3.3: Message Streams

### 3.2.4 Providing Reliable Delivery

**Message Streams**

As previously mentioned, data messages enter a publish/subscribe system through the PHB at the root of the redundant routing tree. The PHB maintains a stream for data messages published by a publisher. These data messages are assigned monotonically increasing timestamps and occupy the message stream in increasing timestamp order.

In gapless delivery, in order to deliver a new data message at an SHB, a no-gap notification must be provided to the SHB that indicates no information is missing between the last delivered data message and the new message. Such a no-gap may result from no publication activity (that is, the elapse of time between publishing two consecutive messages) or a data message being filtered. In our system, we model the no-gap notifications as silence messages. Thus on the conceptual level, a message stream has the following structure:

The conceptual message stream at the PHB contains information for every value of timestamp: a data tick if there is a data message assigned with the

timestamp, a silence tick if there is not a data message with the timestamp, or an unknown tick if the timestamp is in the future.

The routing brokers for the message stream maintain a message stream which mirrors the one at the PHB. However, the silence tick may result from a data tick being filtered out and the unknown tick may be intertwined with other ticks due to message loss and delay of asynchronous communication and due to the fact that there are multiple brokers in a logical node and message traffic is shared among them.

**Requirement on Content-based Routing**

There are two issues in providing support for reliable delivery: First, choose a delivery starting and ending point for a subscription $s$; and second, deliver matching messages without a gap between the starting and ending points. We first address the issue of selecting delivery starting/ending points of a subscription $s$.

Let $b$ be the SHB of $s$. We define the delivery starting point of $s$ as the first message $m_1$ that $b$ receives such that $s \in S(m_1)$. We define the delivery ending point of $s$ as the last message $m_n$ with $s \in S(m_n)$ that $b$ receives before it receives the unsubscription request of $s$.

Due to PHB failures, a message $m$ (or a silence for $m$ if filtered) after $m_1$ may be forwarded with $S(m)$ such that $s \notin S(m)$. This can also occur if a broker on the routing path fails or a message takes a different routing path as a routing broker can change $S(m)$ (Section 3.2.3). In this case, $b$ discards $m$ (or the silence) and waits for a retransmission of the message with the right $S(m)$.

We hence define the *eventual monotonicity* on $S(m)$ of messages.

**Definition 3.7.** *The $S(m)$ set of messages of a published stream in range $[m_1, m_2]$ is eventually monotonic with regard to subscription $s$ if the SHB of $s$ eventually receives those messages (or corresponding silences) with $S(m)$ such that $s \in S(m)$.*

We then present a theorem of sufficient conditions for correct reliable delivery and an informal reasoning of the theorem.

**Theorem 3.1.** *Reliable delivery of a subscription $s$ can be guaranteed if the following conditions are satisfied:*

*1. (Correctness of subscription propagation) Every broker in the publish/subscribe system maintains $S$-$\Lambda$ invariant.*

*2. (Correctness of content-based routing) Routing brokers use sufficiency-directed content-based routing.*

*3. If there is a sufficiently long period of time for which the system runs without failure, newly published messages (or silences) start to arrive at the SHB of $s$ with $S(m)$ such that $s \in S(m)$ unless the system fails again. Delivery starting and ending points of $s$ can be chosen in the aforementioned way.*

*4. The system guarantees eventual monotonicity of $S(m)$ for messages in [starting point, ending point] and the SHB only accepts the transmission of messages with monotonic $S(m)$.*

*Proof.* Condition 1 guarantees that if a message matches a subscription in $S(e) - \overline{S}(e)$ of a broker, then the broker will not filter out this message because it maintains conjunctions that covers the conjunctions in $S(e) - \overline{S}(e)$.

Condition 2 guarantees that if $s$ has not been unsubscribed and a message $m$ has $S(m, e)$ such that $s \in S(m, e)$, then $m$ will not be filtered out by a broker for an outgoing edge $e$ if $m \in M(conj(s))$. This is because sufficiency-directed

content-based routing will only filter out a message if the sufficiency test passes, that is, $S(m,e) - \overline{S}(e) \subset S(e) - \overline{S}(e)$. Because $\Lambda(e) \sqsupseteq_c S(e) - \overline{S}(e)$, we have $\Lambda(e) \sqsupseteq_c S(m,e) - \overline{S}(e)$. Because $s \in S(m,e)$, then $\Lambda(e) \sqsupseteq_c conj(s)$. By definition of $\sqsupseteq_c$, $M(conj(s)) \subseteq M(\Lambda(e))$. Hence, if $m \in M(conj(s))$, $m \in M(\Lambda(e))$, that is, $m$ will not be filtered out.

Condition 3 guarantees that if the system runs sufficiently long without failures, eventually some data message will arrive with $S(m)$ such that $s \in S(m)$. This way we can pick a delivery starting point.

Condition 4 guarantees that all messages (or their corresponding silences if filtered) after the delivery starting point of $s$ arrives at the SHB of $s$ with $s \in S(m)$. Combined with Condition 2, those data messages matching $s$ will not be filtered out because they have the right $S(m)$. In addition, the SHB guarantees that the unsubscription of $s$ will not be propagated and as a result, $s$ will not be included in the $\overline{S}(e)$ set of any upstream broker $b$ until after the delivery ending point. Thus, the system guarantees a gapless sequence of messages for $s$ after its delivery starting point.

□

### 3.2.5 Summary

In summary, we have addressed the issues of what is correct content-based routing and how to select delivery starting/ending points of a subscription. We also note here that eventual monotonicity can be guaranteed using a liveness scheme such as negative acknowledgement (indicating a subscription $s$ is needed to be in $S(m)$) and retransmission. In next two sections, we discuss the issue of maintaining $S$-$\Lambda$

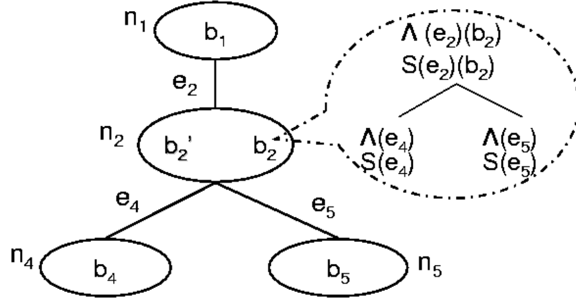



Figure 3.4: Binary Redundant Spanning Tree

invariant in the routing brokers.

## 3.3 Generic Subscription Propagation Algorithm

Publish/subscribe systems are usually dynamic with subscriptions entering and leaving. As a result, the members in the $S(e)$, $\overline{S}(e)$ and $\Lambda(e)$ sets change. In this section, we describe a generic subscription propagation algorithm. We focus on the safety aspect, that is, the maintainence of $S$-$\Lambda$ invariant in such a dynamic environment.

We first describe the information brokers maintain and exchange during the subscription propagation process and then discuss how this information is computed.

Without loss of generality, we consider a directed redundant spanning tree where every non-leaf node has exactly two children. Figure 3.4 shows a partial topology that is of interest for examining broker $b_2$, which resides in node $n_2$ with a redundant peer $b_2'$. Node $n_2$ is connected to two of its children through edge $e_4$ and $e_5$ and parent node $n_1$ through edge $e_2$.

### 3.3.1 Subscription Information Maintenance and Exchange

For broker $b_2$ to route messages to subscribers that are downstream of edge $e_4$ and $e_5$, $b_2$ maintains the following information:

- Set of *conjunctions* $\Lambda(e_4)$ and $\Lambda(e_5)$ for matching messages before sending on edges $e_4$ and $e_5$;
- Set of *subscriptions* $S(e_4)$ and $S(e_5)$;
- Set of *unsubscriptions* $\overline{S}(e_4)$ and $\overline{S}(e_5)$.
- Presumed knowledge of the state for edge $e_2$ maintained by a broker (e.g., $b_1$) in $b_2$'s upstream node $n_1$. This includes the presumed conjunction set, $\Lambda(e_2)(b_2)$, the presumed subscription and unsubscription sets $S(e_2)(b_2)$ and $\overline{S}(e_2)(b_2)$. Note that we have included $(b_2)$ in the suffix to distinguish the presumed state from the actual state.

In an asynchronous distributed system, there is no guarantee that the real state in $b_1$ is exactly as $b_2$ presumes. This state is computed at $b_2$ as follows:

$$S(e_2)(b_2) = S(e_4) \cup S(e_5) \tag{3.3.1}$$

$$\overline{S_2}(e)(b_2) = \overline{S}(e_4) \cup \overline{S}(e_5) \tag{3.3.2}$$

$$\Lambda(e_2)(b_2) = \Lambda(e_4) \cup \Lambda(e_5) \tag{3.3.3}$$

When subscription changes happen, the SHB generates information about the change (described in Section 3.3.2). This information propagates upstream and enters broker $b_2$ through the reverse direction of its outgoing edges $e_4$ and $e_5$. The propagation message, e.g. on edge $e_4$, includes the following items

- $\Delta S(e_4)$ represents the new subscriptions;
- $\Delta\overline{S}(e_4)$ represents the subscriptions that have been unsubscribed;
- $\Delta\Lambda(e_4)$ represents the conjunctions that should be added or removed if $\Delta S(e_4)$ and $\Delta\overline{S}(e_4)$ are to be applied. The elements of $\Delta\Lambda(e_4)$ take the form of "$+\lambda$" or "-$\lambda$";
- constraint set $C(e_4)$ of Boolean expressions represents the conditions on $S(e_4)$ and $\overline{S}(e_4)$ at $b_2$ in order for it to apply the aforementioned changes.

**Incremental Change Application Constraints** An example of the aforementioned constraint is the existence of a particular subscription $s$ in an upstream broker. That is, the upstream broker has received information about $s$ and has not learned of an unsubscription of $s$. This constraint could result from that the current broker has aggregated away some conjunctions of the new subscription due to those conjunctions being covered by subscription $s$'s conjunctions. Thus, in order for the upstream broker to accept the incremental change generated by this broker, it needs to have a subscription like $s$ that covers the conjunctions in the incremental change that are aggregated away.

We note that the constraints specify a set of sufficient conditions. The upstream broker can contain no subscription $s$ but other subscriptions that cover the aggregated-away conjunctions. The constraint is choosing in a way based on a principle of *locality*. That is, since the current broker contains a subscription $s$, it is highly likely that the upstream broker also does.

We give examples of constraints later in Section 3.4 in this Chapter and in Chapter 4, where we show examples of constraints both at the abstract level and

with specific encodings.

When $b_2$ receives the propagation message, it evaluates $C(e_4)$. If $C(e_4)$ is not satisfied, it ignores the message. [3] If $C(e_4)$ is satisfied, $b_2$ applies the $\Delta$'s to $S(e_4)$, $\overline{S}(e_4)$ and $\Lambda(e_4)$. Applying $\Delta S(e_4)$ and $\Delta\overline{S}(e_4)$ is to take the set union with $S(e_4)$ and $\overline{S}(e_4)$ respectively. Applying $\Delta\Lambda(e_4)$ means adding a conjunction "+$\lambda$" or removing a conjunction "-$\lambda$" if it is in $\Lambda(e_4)$.

After applying the $\Delta$'s, broker $b_2$ computes subscription changes for brokers in its upstream node $n_1$. The items include $\Delta S(e_2)$, $\Delta\overline{S}(e_2)$, $\Delta\Lambda(e_2)$ and $C(e_2)$ and are computed from the changes to the presumed state $S(e_2)(b_2)$,$\overline{S}(e_2)(b_2)$ and $\Delta\Lambda(e_2)(b_2)$. If $\Delta\Lambda(e_2) = \emptyset$, broker $b_2$ has the option not to propagate the subscription further. Otherwise, $b_2$ sends the subscription message to $b_1$ through the reverse direction of its incoming edge $e_2$. The subscription message may also be routed to other brokers (if any) in $n_1$ following any path provided by the underlying routing topology. For instance, if $b_1$ has a redundant peer connected, it may receive the subscription message from $b_1$. However, our algorithm does not rely on any synchrony between the brokers in a node.

### 3.3.2 Computing Subscription Change for an Upstream Broker

This section describes the computation of incremental subscription changes at broker $b_2$ for brokers in its upstream node $n_1$.

[3] As we focus on the safety aspect of the algorithm in this chapter, we do not discuss liveness issue here. In a working system such as the one shown in Chapter 4, a liveness mechanism will ensure that every broker eventually receives up-to-date subscription information, regardless of incremental change messages ignored due to unsatisfied constraints.

**Computing $\Delta S(e_2)$ and $\Delta\overline{S}(e_2)$** Since each subscription and its corresponding unsubscription enter the spanning tree at only one point, $S(e_4) \cap S(e_5) = \emptyset$ and $\overline{S}(e_4) \cap \overline{S}(e_5) = \emptyset$. Using Formula 3.3.1 and 3.3.2, we have $\Delta S(e_2) = \Delta S(e_4)$ and $\Delta\overline{S}(e_2) = \Delta\overline{S}(e_4)$.

**Computing $\Delta\Lambda(e_2)$ and $C(e_2)$** Broker $b_2$ computes incremental change $\Delta\Lambda(e_2)$ from $\Delta\Lambda(e_4)$ and $b_2$'s current presumed state of an upstream broker - $\Lambda(e_2)(b_2)$. Depending on the aggregation scheme used, this may generate one or more constraints. Constraint $C(e_2)$ includes these new constraints and the constraints in $C(e_4)$. We do not restrict the generic algorithm to any specific aggregation scheme. Instead, we only require that the aggregation scheme computes constraint set $C(e_2)$ and $\Delta\Lambda(e_2)$ such that if a broker initially satisfies the $S$-$\Lambda$ invariant and applies $\Delta\Lambda(e_2)$ only when the constraints are satisfied, the broker maintains the $S$-$\Lambda$ invariant.

An aggregation scheme can be one that does not aggregate the conjunctions. The constraint set is always $\emptyset$. In such a system, the subscription changes received at SHBs are flooded to all brokers without change and brokers maintain all subscription conjunctions. We describe a more sophisticated aggregation scheme in the following section.

## 3.4 Subscription Propagation with Aggregation

The aggregation of $\Lambda(e_4)$, $\Lambda(e_5)$ and $\Delta\Lambda(e_4)$ at broker $b_2$ produces the conjunction changes $\Delta\Lambda(e_2)$ and constraints $C(e_2)$ that are part of the incremental subscription change the broker sends upstream. In this section, we describe an abstract scheme

for aggregating these conjunctions based on their covering relationships. Our purpose is to provide a foundation for analyzing the family of covering-based aggregation schemes.

### 3.4.1 Aggregating Conjunctions

The fact that covering conjunctions matches all messages of their covered conjunctions allows a broker to withhold from propagating changes of conjunctions that are covered by one or more conjunctions that the broker has already propagated and still guarantees correct content filtering. That is, $\Delta\Lambda(e)$ (for any edge $e$) needs to include only conjunctions that are not covered. For the same reason, a broker only needs to maintain the non-covered conjunctions in its conjunction set $\Lambda(e)$.

As previously described, broker $b_2$ applies $\Delta\Lambda(e_4)$ to obtain a new state of $\Lambda(e_2)(b_2)$ and $\Delta\Lambda(e_2)$ is computed as the conjunction change in $\Lambda(e_2)(b_2)$. We organize the conjunctions in $\Lambda(e_2)(b_2)$ into a DAG for easy representation of the covering relationships. The DAG nodes are conjunctions with edges drawn from a covering conjunction (a parent) to each of its covered conjunction (a child). The roots of the DAG are the set of conjunctions that are not covered and hence should be propagated. We examine the changes in the DAG resulted from applying $\Delta\Lambda(e_4)$. Incremental change $\Delta\Lambda(e_2)$ and $C(e_2)$ thus represent the change to the root set and the conditions under which it happened. The initial value of $\Delta\Lambda(e_2)$ is an empty set $\emptyset$, whereas the initial value of $C(e_2)$ is set to $C(e_4)$ because the computation is based on $b_2$ satisfying constraints $C(e_4)$.

We first introduce a notation of $sub(\lambda)$ as the set of subscriptions in $S(e)-\overline{S}(e)$

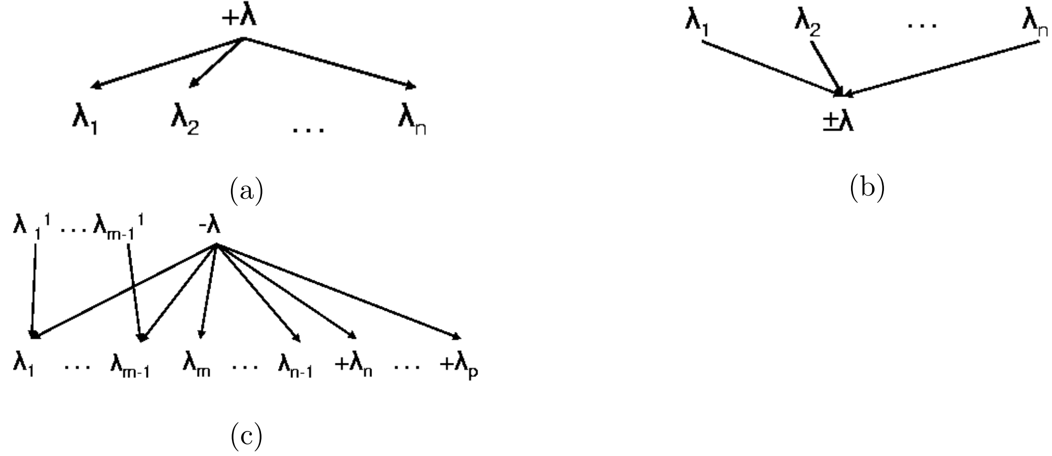



Figure 3.5: Applying Incremental Changes $\Delta\Lambda_m$ in $b_2$

over all edges at a broker such that $\lambda \in conj(s)$ for each $s$ in this set.

**Adding a Root Conjunction** Figure 3.5(a) illustrates the case of adding a conjunction $\lambda$ that will be a root in the new conjunction DAG of $\Lambda(e_2)(b_2)$.

The new root conjunction $\lambda$ may cover zero or more conjunctions. As a newly created root in the DAG of $\Lambda(e_2)(b_2)$, "$+\lambda$" should be propagated and thus belongs to $\Delta\Lambda(e_2)$. Since $\lambda$ is propagated, no new constraint is created.

**Adding or Removing a Non-Root Conjunction** Figure 3.5(b) describes the cases when a non-root conjunction *lambda* is added or removed while it is covered by existing conjunctions $\lambda_1, \ldots, \lambda_n$ that will continue to exist in the new DAG of $\Lambda(e_2)(b_2)$. Since $\lambda$ is not a root conjunction, it does not need to be propagated and thus this results in no change to $\Delta\Lambda(e_2)$.

On the other hand, the removal of $\lambda$ from the incremental change is based on the existence of at least one of its covering conjunctions. Since the constraints represent a requirement on an upstream broker's matching sets of subscriptions

and unsubscription, this is to require the existence of at least one subscription in $\bigcup_{i=1}^{n} sub(\lambda_i)$. As a result, the following constraint is added to $C(e_2)$

$$\bigvee_{s \in \bigcup_{i=1}^{n} sub(\lambda_i)} s \in S(e_2) - \overline{S}(e_2) \qquad \text{(I)}$$

**Removing a Root Conjunction** Figure 3.5(c) describes the case when an existing root conjunction $\lambda$ is removed and $\lambda$ is not covered by any new conjunction that is added to the DAG. Conjunction $\lambda_1, \lambda_2, \ldots, \lambda_{n-1}$ represent the set of conjunctions that are directly covered by $\lambda$. Conjunction $\lambda_1, \lambda_2, \ldots, \lambda_{m-1}$ ($m \leq n$) represents those conjunctions in $\lambda_1, \ldots, \lambda_{n-1}$ that are also covered by other existing conjunctions. For example, $\lambda_1$ is covered by $\lambda_1^1$. For simplicity, we only show one covering conjunction for each of $\lambda_1, \ldots, \lambda_{m-1}$. Conjunction $\lambda_n, \ldots, \lambda_p$ represent newly added conjunctions that are covered by $\lambda$.

As a removed root, $\lambda$ does not belong to the new DAG of $\Lambda(e_2)(b_2)$ and "$-\lambda$" should be added to $\Delta\Lambda(e_2)$. Because $\lambda_m, \ldots, \lambda_{n-1}$ become new root conjunctions, we add "$+\lambda_m, \ldots, \lambda_{n-1}$" to $\Delta\Lambda(e_2)$. Note that newly added root conjunction $\lambda_n, \ldots, \lambda_p$ will be handled as described in "Adding a Root Conjunction". To summarize, $\Delta\Lambda(e_2)$ is updated as follows:

$$\Delta\Lambda(e_2) \leftarrow \Delta\Lambda(e_2) \cup \{-\lambda\} \cup \bigcup_{i=m}^{n-1} \{+\lambda_i\}$$

Due to an upstream broker $b_1$ not maintaining covered conjunctions, all conjunctions of subscriptions in $b_1$ that were covered by $\lambda$ should be restored if they are not covered by other conjunctions in $\Lambda(e_2)$. Adding "$+\lambda_{m_{n-1}}$" to $\Delta\Lambda(e_2)$ partly satisfies this requirement. However, if there is a subscription $s$ in $b_1$ that is unknown to $b_2$ such that one or more conjunctions of $s$ are covered by $\lambda$, then the upstream broker may violate the $S$-$\Lambda$ invariant. To guarantee that such a case does not happen, we make sure there is no subscription in $b_1$ that is unknown to

$b_2$ through the following constraint

$$S(e_2) - \overline{S}(e_2) \subseteq S(e_2)(b_2) - \overline{S}(e_2)(b_2) \quad \text{(II)}$$

The withholding of $\lambda_1, \ldots, \lambda_{m-1}$ in Figure 3.5(c) is based on the existence of at least one of their covering conjunctions (represented by variable $\lambda_t$ in the following formula) in broker $b_1$. This is represented by the following constraint for *each* $\lambda_i$ where $i = 1, \ldots, m-1$:

$$\bigvee_{s \in \bigcup_{(\lambda_t \succeq_c \lambda_i)} sub(\lambda_t)} s \in S(e_2) - \overline{S}(e_2) \quad \text{(III)}$$

This type ( III) of constraints are the same as type I. They are only caused by different situations.

As we describe in Section 3.4.2, constraints of type I II and III are sufficient to guarantee the $S$-$\Lambda$ invariant. For efficiency of content-based routing, it is essential to guarantee that if a broker $b_1$ passes the sufficiency test against a message $m$ for an outgoing edge $e_2$, $b_1$ will filter $m$ if $m$ does not match any subscription in $S(e_2) - \overline{S}(e_2)$. That is, subscription propagation needs to prevent from adding conjunctions that do not belong to any subscription in $S(e_2) - \overline{S}(e_2)$. This is represented by the following constraints for each $\lambda_i$ in $\lambda_m, \ldots, \lambda_{n-1}$ that is added to $\Delta\Lambda(e_2)$:

$$\bigvee_{s \in sub(\lambda_i)} s \in S(e_2) - \overline{S}(e_2) \quad \text{(IV)}$$

### 3.4.2 Operation and Constraint Properties

If a publish/subscribe system only uses the aforementioned operations and constraints, every broker will maintain the $S$-$\Lambda$ invariant. The following theorem is a formal statement of this property.

**Theorem 3.2.** *For any broker $b$ and an outgoing edge $e$ in a redundant spanning*

*tree, if the $S(e)$, $\overline{S}(e)$ and $\Lambda(e)$ sets of $b$ initially satisfy the $S$-$\Lambda$ invariant and are only modified by the operations and constraint type I, II and III described in Section 3.3.2, the broker maintains $S$-$\Lambda$ invariant.*

The use of type IV constraints guarantees that the broker does not maintain extra conjunctions other than those needed for performing content matching for the subscriptions it maintains. The following theorem is a formal statement of this property.

**Theorem 3.3.** *We consider broker $b$ and its outgoing edge $e$ in a network using operation and all types of constraints described described above. For every conjunction $\lambda \in \Lambda(e)$, there exists a subscription $s$ such that $\lambda \in conj(s)$ and $s \in S(e) - \overline{S}(e)$.*

*Proof.* We prove by induction on the number of incremental updates broker $b$ processes.

1. **Base case. Broker $b$ has processed $0$ incremental updates.**

   Initially, $S(e) = \overline{S}(e) = \Lambda(e) = \emptyset$, so this satisfies the theorem.

2. **Induction assumption**

   Suppose the theorem holds for broker $b$ before processing an incremental update ($\Delta S(e)$, $\Delta\overline{S}(e)$, $\Delta\Lambda(e)$ and $C$) from its downstream.

3. **Induction rule**

   We prove for every conjunction $\lambda \in \Lambda(e) + \Delta\Lambda(e)$, there exists a subscription $s$, such that $\lambda \in conj(s)$ and $s \in S(e) \cup \Delta S(e) - \overline{S}(e) - \Delta\overline{S}(e)$.

   There are two types of events to consider:

(a) "$+\lambda$"$\in \Delta\Lambda(e)$ - because this may result in (new) conjunctions that do not have a subscription in $S(e) - \overline{S}(e)$ sets. This can occur in two cases:

   i. $\lambda$ belongs to a new subscription $s$. In this case, $s \in \Delta S(e)$ and $s \notin \Delta\overline{S}(e)$.

      Because $\Delta S(e)$ stays the same throughout the propagation process 3.3.2, it is exactly the same as the value that is initially computed at the SHB. Because the members in $S(e)$ and $\overline{S}(e)$ are originated at SHBs and each subscription only connects from one SHB, $\Delta S(e) \cap \overline{S}(e) = \emptyset$. We thus have $s \notin \overline{S}(e)$. Therefore

      $s \in S(e) \cup \Delta S(e) - \overline{S}(e) - \Delta\overline{S}(e)$

      The theorem holds.

   ii. $\lambda$ is generated in the case of "Removing a Root Conjunction" described in Page 55. From constraint IV, we have $\exists s \in sub(\lambda)$ such that $s \in S(e) - \overline{S}(e)$. Because there is a "$+\lambda$" but not "$-\lambda$" in $\Delta\Lambda(e)$, $s \notin \Delta\overline{S}(e)$. Thus

      $s \in S(e) \cup \Delta S(e) - \overline{S}(e) - \Delta\overline{S}(e)$

      The theorem holds.

(b) $s \in \Delta\overline{S}(e)$ - because this may result in $b$ not removing conjunctions whose subscription are removed. Initially, every $s \in \Delta\overline{S}(e)$ will have a "$-\lambda$" in $\Delta\Lambda$. Two cases could happen to this "$-\lambda$" when a downstream broker, e.g., $b_1$ computes the $\Delta\Lambda$ for subscription aggregation. We examine the activities happening in $b_1$ and prove that if "$-\lambda$" is removed by $b_1$ from the $\Delta\Lambda(e)$ it sends upstream, an upstream broker will not have $\lambda$ in its $\Lambda(e)$ DAG either.

i. "$-\lambda$" stays in $\Delta\Lambda(e)$. This can only happen in the case of "Remove a Root Conjunction". The theorem holds in this case.

ii. "$-\lambda$" is removed. This can only happen in the case of "Removing a non-Root Conjunctions" described in Section 3.4.1. In this case, a type I constraint is generated.

Thus there exists $s'$ and $\lambda' \in conj(s')$ such that $s' \in S(e) - \overline{S}(e)$ and $\lambda' \succeq_c \lambda$. Because $s'.\lambda'$ exists in downstream broker $b_1$'s DAG, it is not being removed. Hence $s' \notin \Delta\overline{S}$. Thus we have

$s' \in S(e) \cup \Delta S(e) - \overline{S}(e) - \Delta\overline{S}$.

From theorem 3.2, either $s'.\lambda' \in \Lambda(e) + \Delta\Lambda(e)$, or $s'.\lambda'$ is covered by a conjunction in it. In both cases, because $s'.\lambda' \succeq_c \lambda$, $\lambda$ will be removed by the minimization, and thus the theorem holds.

□

## 3.5 Subscription Propagation with Filter/Conjunction Merge

As previously mentioned, subscription propagation is an optimization that may provide better system performance. It is a technique that trade off computation overhead of performing content matching versus communication overhead that is incurred in transmitting extra messages in the system that may be discarded later.

Opyrchal et al. ([51]) observed that under some range of conditions, a simple flooding algorithm is viable; and under conditions of high selectivity and high regionalism of subscriptions, content-based routing performs significantly better;

however, the specific algorithm to use depends upon the economics of deployment.

Mühl et al. ([49]) argued that as the number of subscriptions increases, the overhead on performing content matching is likely to increase while the selectivity of subscriptions decreases. As a result, it is viable to use fewer but simpler subscription filters even though this may result in a few extra messages being transmitted in the system.

The flooding algorithm and the previously described subscription propagation/content-based routing algorithm are the two extremes of the spectrum where the maximum simplicity of routing and efficiency of communication are achieved, respectively. Mühl et al. ([49]) have also argued the usefulness of what they call *imperfect filter merging*. That is, in some cases, it is desirable to achieve a middle ground where the simplicity of content matching and efficiency of communications are traded-off. This usually refers to a technique known as *subscription merging* by which a broker or brokers in a content-based publish/subscribe system substitute and propagate a small number of *coarser* subscription filters instead of a large number of *finer* subscription filters that are largely overlapping.

We think that *subscription merging* can be a very useful mechanism in terms of improving system performance under certain circumstances. Hence, it is necessary to explore how such a mechanism can be accommodated in our framework.

It is obvious that whether and what *coarse* filters to use for substitution should be an adaptive decision based on the distribution of the *original* subscription filters and the distribution of the message patterns in terms of matching those subscriptions. With this decision, a mechanism is required to enable this merging of *finer* subscriptions. We discuss in this section such an enabling mechanism that

supports reliable delivery in our redundant routing networks. The whole system that is capable of performing adaptive subscription merging is beyond the scope of this thesis and may be explored in the future.

### 3.5.1 Merging Brokers

In a system that is capable of applying subscription merging, *merging subscriptions* are generated by the system to replace the original subscriptions from the downstream of a broker.

Our system can be extended to support subscription merging by using a special *merging broker* for hosting these *merging subscriptions* at each broker that performs merging. Specifically, for each broker $b$ in a non-leaf and non-root node $n$ in the redundant routing tree, we create a special broker $b^m$ and its residing leaf node as a child of $n$. This is possible because all brokers in our system are *virtual* and a physical broker can implement one or more *virtual brokers*. Thus, we can create the merging broker $b^m$ by letting the physical broker of $b$ to implement an additional *virtual broker*.

We further make sure that one can distinguish a *merging subscription* from other subscriptions from downstream. This can be done through encoding the identity of the tree node $n$ in the subscription's identification.

The broker network shown in Figure 3.1 is translated into a network in Figure 3.6 with merging brokers ${b_2}^m$ in node ${n_2}^m$ and ${b_2}'^m$ in node ${n_2}'^m$.

### 3.5.2 Algorithm Sketch

A publish/subscribe system with subscription merging usually operates in two ways: merge a number of subscriptions by supplying a few covering (or merging)

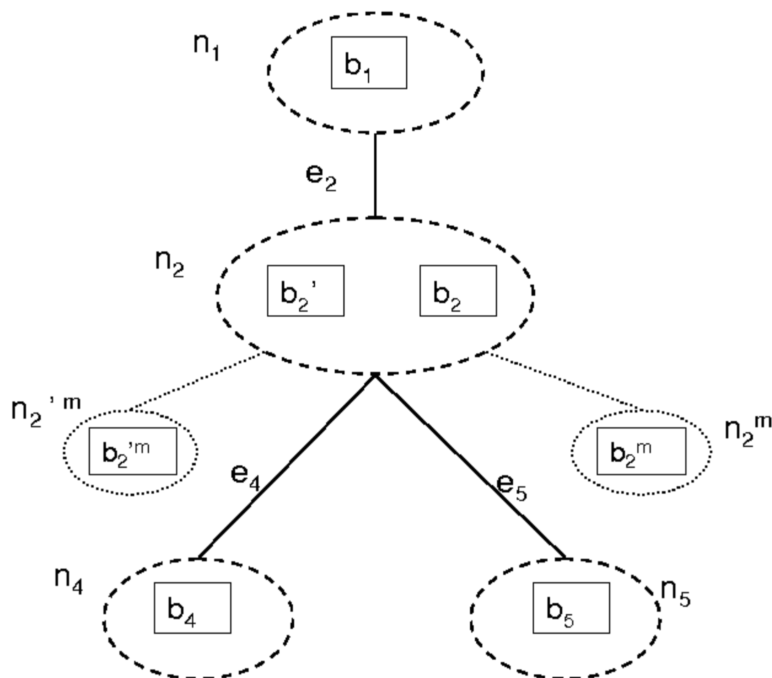



Figure 3.6: Redundant Routing Network Supporting Subscription Merge

subscriptions and replace the merging subscriptions with the original subscriptions in the reverse direction.

Subscription merging can be performed at any broker other than the PHB as the PHB has no further upstream broker to propagate subscriptions. The decision on whether and what merging subscriptions to use can be made locally at each broker through an algorithm. A such typical algorithm should take into consideration the histories of the communication overhead of wasted messages and the computation overhead of matching the large number of original subscriptions. The algorithm then should make decisions based on its prediction of these tradeoff overheads. This aspect is a future extension of our work and is outside the scope of this thesis.

The mechanism for enacting this changes works natually by adapting the generic algorithm previously described in this chapter with the subscriptions from the merging broker. Thus, the system propagates merging or original subscription filters with the constraint conditions. Upon restart from a merging broker crash, a special unsubscription of all the merging subscriptions will be issued and the broker can then begin using new merging subscriptions.

## 3.6 Summary

In this chapter, we have discussed a general algorithmic model for subscription propagation, content-based routing and reliable delivery in a redundant broker network. We have also shown how a mechanism that allows adaptive subscription merging can be accommodated in this framework through the use of special *virtual merging brokers*.

# Chapter 4

# Subscription Propagation Using Virtual Time Vectors

We have discussed a generic subscription propagation and *sufficiency-directed* content-based routing algorithm in Chapter 3. The generic algorithm uses abstract concepts of subscription sets and leaves each concrete protocol with an option to pick its own representation of these sets.

In this chapter, we present a working subscription propagation protocol that utilizes virtual time vectors as an instrument for representing subscription sets. We show how this protocol can be interpreted as an encoding of the generic algorithm and hence its correctness can be derived from the correctness of the generic algorithm. We then discuss the liveness and failure handling features of the protocol.

## 4.1 Protocol Description

The centerpiece of this protocol is the use of subscription virtual start and end times as an encoding method of the generic algorithm.

### 4.1.1 Virtual Times

Perhaps one of the most important concepts in distributed computing is *virtual time* ([34]) or *logical clocks* ([40]). These concepts reflect people's intuition that some numerical instrument may be a useful abstraction of the state of a distributed system and can reflect the dependency/ordering of events.

We use virtual times as an abstraction of the state of subscriptions at a SHB. That is, instead of saying *currently subscription $s_2$ and $s_3$ are connected at the SHB, and subscription $s_1$ has just disconnected*, a more concise representation is *the SHB is at virtual time 4* (due to the arrival of subscription $s_1$, $s_2$, $s_3$ and the subsequent departure of $s_1$).

In this protocol, a SHB/leaf broker maintains a *virtual time clock*. The leaf broker ensures the monotonicity of its virtual clock, despite crash recovery. For the discussions in this chapter except Section 4.3, we assume the virtual clock is an integer counter. The virtual clock starts at one when the SHB starts.

The events that advance the virtual clock are client subscriptions and unsubscriptions at the SHB. Whenever a SHB's aggregated subscriptions change and the SHB decides to propagate the change to its upstream brokers, the SHB advances its virtual time clock by 1.

A client subscription/unsubscription may or may not change the *aggregated subscriptions* of its SHB, depending on whether existing/remaining subscriptions

*cover* the client's subscription or not. As described in Chapter 3, the aggregated subscriptions are the subscriptions a broker presents to its upstream brokers in order to receive messages matching its own subscriptions.

The SHB assigns *virtual start times* to its hosted subscriptions, a.k.a. subscriptions connected at this SHB. All new subscriptions are assigned a virtual start time that is at most the value of the virtual clock's current time. The SHB also assigns *virtual end times* to the subscriptions that are departing using the current clock value. This virtual end time, even though does not need to be maintained in our system, defines the life span of a subscription together with its *virtual start time*.

Next, we discuss how virtual start/end times are assigned. As described in Section 3.2.1, we assume that a client subscription is in the form of a set of Boolean conjunctions. The assignment of virtual times to a client subscription is calculated based on the covering relationship of the existing subscription conjunctions and the conjunctions of the client's subscription.

### 4.1.2 The Conjunction DAG

At a SHB, the assignment of virtual start times to new subscriptions and the computation of aggregated subscriptions to its upstream brokers are based on the covering relationship of subscription conjunctions. We use a directed acyclic graph (DAG) to facilitate the representation and discovery of the covering relationship. The DAG is constructed by modelling conjunctions as nodes and drawing a directed edge from a covering conjunction to a covered conjunction. As the covering relationship is transitive, we omit the transitive edges. Initially, broker conjunc-

tion DAGs are empty until client subscriptions are connected.

Several examples of conjunction DAGs are shown in Figure 4.2 in Page 70. Each conjunction is represented as a rectangle with a virtual start time, and an oval representing the list (called *subscriber list*) of subscribers or downstream routing tree nodes whose subscriptions contain the conjunction.

Given a conjunction $c$, operations such as searching for the covering conjunctions of $c$, inserting $c$ and removing $c$ are simplified and expedited by the DAG. The broker can conduct a breadth-first search for conjunctions that *immediately* covers $c$ or is *immediately* covered by $c$. By "a conjunction $c'$ *immediately covers* $c$", we mean that there are no other conjunctions in the DAG that are covered by $c'$ and themselves cover $c$. Inserting conjunction $c$ into the DAG involves putting $c$ between its immediate covering and covered conjunctions and removing the old edge between any pair of its immediate covering and covered conjunctions. Vice versa, removing a conjunction $c$ involves removing $c$ and adding an edge between any pair of its immediate covering and covered conjunctions. The DAG, as a result, maintains the following properties

- *Completeness*: to the extent of the capability of the covering test algorithm used, every covering relationship is represented in that there is a path between every pair of conjunctions that have a covering relationship detected by the covering test algorithm;
- *Minimality*: there is no transitive edge;
- *Monotonicity*: every node has a higher virtual start time than any of its descedents. We defer the discussion of this property to next section.

### 4.1.3 Handling New Subscriptions and Assigning Virtual Start Times

To calculate the virtual start time of a new subscription, we first calculate a virtual start time for each of its conjunctions. We ignore the trivial case that there is an existing conjunction in the DAG that is the same as the new conjunction.

For each conjunction $c$ of the new subscription, the broker inserts $c$ into the DAG and set $c$'s virtual start time to the minimum of the virtual start times of its immediate covering conjunctions. If no such covering conjunction exists, $c$ becomes a new root node, and its virtual start time is set to the current time of the broker's virtual clock. As the broker's virtual clock only monotonically increasing, this operation ensures the monotonicity of virtual start times from a node to any of its descedent nodes.

The above process is repeated for every conjunction of the new subscription. Their virtual start times are used to calculate the virtual start time of the new subscription. As we know, to ensure reliable delivery, message delivery for a new subscription can not start until matching messages for all its conjunctions start to arrive. Hence, we set the virtual start time of the new subscription to the maximum of the virtual start times of its conjunctions'.

We use examples below to further illustrate this process.

Figure 4.1 shows a broker network with three levels of nodes: top level root node $N_{31}$ where the publisher hosting broker $PB_1$ resides; middle level node $N_{21}$ and $N_{22}$ where the intermediate routing nodes $B_1$, $B_2$ and $B_3$, $B_4$ reside respectively; and leaf level nodes $N_{11}$, $N_{12}$ and $N_{13}$ where the subscriber hosting broker $SB_1$, $SB_2$ and $SB_3$ reside respectively. A publisher $p$ connects at $PB_1$ and subscribers $s_1$, $s_2$, $s_3$, $s_4$ and $s_5$ connect at $SB_1$, $SB_2$ and $SB_3$, respectively. We now

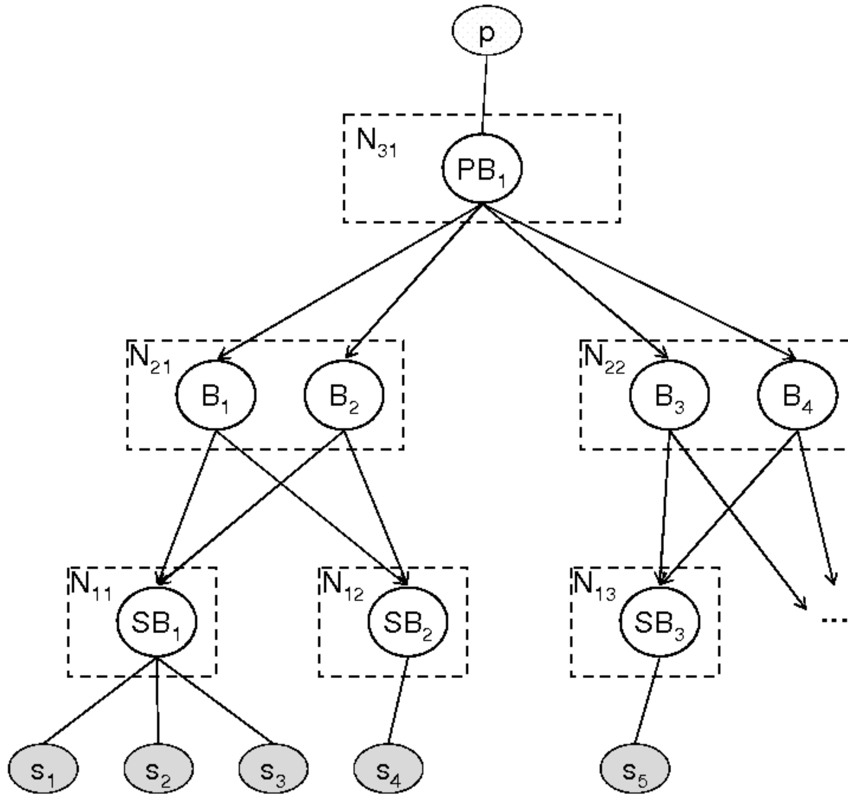


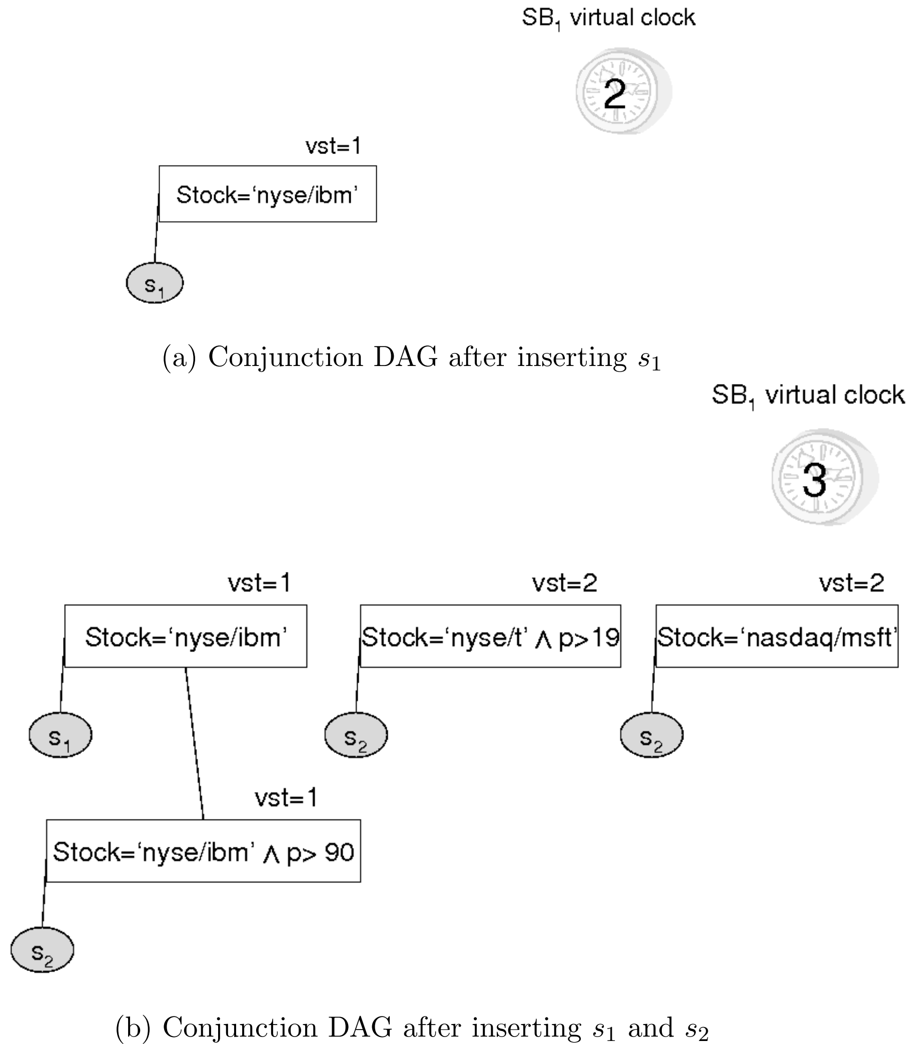

Figure 4.1: A Redundant Broker Network

examine the changes at subscriber hosting broker $SB_1$, $SB_2$ and $SB_3$ when they process client subscriptions.

**Example 4.1.** *Initially, there is no subscription at any of the brokers in the system. Hence, the brokers have an empty conjunction DAG and a virtual clock time of* 1*. When subscriber* $s_1$ *at* $SB_1$ *submits the following subscription:*

$$stock = \text{‘}nyse/ibm'$$

*the only conjunction of this subscription is inserted into the conjunction DAG. If broker* $SB_1$ *decides to propagate all pending subscription changes since the last time it has done so, it increments its virtual clock time by* 1 *and this conjunction*




(a) Conjunction DAG after inserting $s_1$

(b) Conjunction DAG after inserting $s_1$ and $s_2$

Figure 4.2: State at Broker $SB_1$ After Inserting Subscriptions $s_1$ and $s_2$

*becomes the only conjunction with virtual start time of* 1*. The conjunction DAG and virtual clock time are shown in Figure 4.2(a).* □

In the above example and the following ones, we use a notation to represent topic hierarchies. Topic hierarchy is a concept supported in many publish/subscribe systems. In '*nyse/ibm*', '/' is the hierarchy separator that represents the sub-topic '*ibm*' under '*nyse*'. As we will see from later examples,

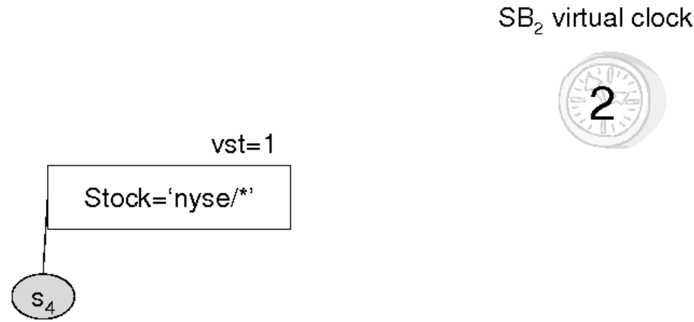



Figure 4.3: State at Broker $SB_2$ after Inserting Subscription $s_4$

wildcards such as '$*'$ are used to represent any topic under a hierarchy.

Example 4.2 is a similar example showing the change at broker $SB_2$ when subscriber $s_4$ submits the first subscription at $SB_2$.

**Example 4.2.** *Initially, broker $SB_2$ has an empty conjunction DAG and subscriber $s_4$ submits the following subscription:*

$$stock = `nyse/*'$$

*When $SB_2$ propagates this latest change, its virtual clock advances to* 2*. Figure 4.3 shows the state of $SB_2$ after processing and propagating subscriber $s_4$'s subscription.* □

Now both broker $SB_1$ and $SB_2$ have non-empty conjunction DAG and have propagated subscriptions to upstreams at this point. We delay the discussion of the propagation process until Section 4.1.5 and continue to examine broker $SB_3$ and $SB_1$ when more subscriptions are submitted.

**Example 4.3.** *When subscriber $s_5$ at broker $SB_3$ submits it subscription*

$$stock = `nasdaq/goog'$$

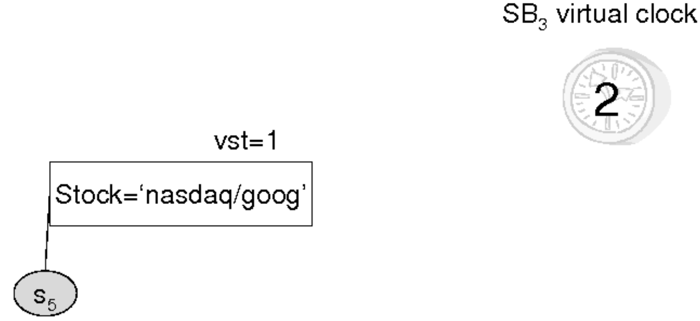



Figure 4.4: State at Broker $SB_3$ after Inserting Subscription $s_5$

*broker $SB_3$'s adds the conjunction to its empty DAG and advances its virtual clock to 2. The state of $SB_3$ is shown in Figure 4.4.*

**Example 4.4.** *Suppose subscriber $s_2$ at broker $SB_1$ submits the following subscription where attribute $p$ represents the price of the stock:*

$$stock = `nyse/ibm' \wedge p > 90 \vee stock = `nyse/t' \wedge p > 19 \vee stock = `nasdaq : msft'$$

*This subscription contains three conjunctions: the first one is covered by an existing conjunction in $SB_1$'s DAG and hence assigned the same virtual start time of 1, and the other two are new conjunctions and hence assigned a virtual start time of 2, which is the broker's current clock value. As a result, there are changes to the aggregated subscriptions at broker $SB_1$ and needs to be propagated. When broker $SB_1$ propagates this change, the virtual clock is advanced to 3. The state of $SB_1$ after propagating this change is shown in Figure 4.2(b).* □

We note that the nodes in the conjunction DAG are assigned non-increasing virtual start time as one travels from a root node to the covered nodes. Thus computing virtual start time of a new conjunction only takes into consideration

the immediate covering nodes. As a reminder, transitive arcs are omitted from the conjunction DAG.

### 4.1.4 Handling Unsubscription

When a subscriber unsubscribes a subscription, the broker first locates in the DAG the conjunctions of the subscription. For every conjunction of the subscription, the subscriber is removed from the subscriber list of the conjunction node. If a conjunction node does not have any subscriber in its list, the conjunction node may be removed from the DAG.

The removed subscription is assigned a virtual end time using the broker's current virtual clock time. This virtual end time represents the end of life span of the subscription, and from this point, message routing and delivery in the system do not concern this subscription.

At the end of the client subscribing/unsubscribing process, if any roots are added or removed, the SHB's aggregated subscriptions change and the SHB needs to propagate the change to its upstream brokers. When this happens, the SHB advances its virtual clock by one. In addition, an incremental update is generated and propagated to upstream brokers in the redundant routing tree.

We note that although we have discussed the subscribing and unsubscribing process as if there is only one request at a time, the algorithm actually processes requests in batches. The size of the batch is a tuning parameter and may be administratively configured. The broker only advances its virtual clock at the end of the batch process if the set of root nodes is changed.

We also note that once a subscription is removed, there is no need to store

information for it. The virtual end time of the subscription is not really assigned and is never maintained in the system. It is a conceptual value we use to describe the mapping of the protocol from the generic algorithm.

### 4.1.5 Computing an Incremental Subscription Change Message

As previously described, if a client subscription/unsubscription results in changes in the set of root nodes of a conjunction DAG, the leaf broker/SHB computes and propagates incremental changes about the aggregated subscriptions.

An incremental change contains

- the name of the originating leaf broker, e.g. $SB_1$;
- the virtual time $v_1$ of $SB_1$ when the change occurred;
- a list of additive/subtractive conjunctions $\{+c_1, +c_2, \ldots, -c_i, -c_{i+1}, \ldots\}$;
- the subscribing tree node; and
- a constraint vector on receiving broker's $V^{\mathrm{b}}$ virtual time vector.

where $c_1$ and $c_2$ are the newly-added root DAG nodes, $c_i$ and $c_{i+1}$ are the old roots that were just removed from the DAG.

The constraint vector is initially set to $(SB_1, v_1 - 1)$ at the SHB. That is, in order to apply this incremental update to change its state of subscriptions, the receiving broker should have received information of all subscriptions connected at leaf broker $SB_1$ with virtual start time $< v_1$. As we will see later in Section 4.1.6, this constraint requires the receiving broker to have a virtual time vector of $v_1 - 1$ for this SHB.

Our examples in Section 4.1.3 produce the following incremental subscription changes:

1. $SB_1$ generates: "$SB_1$, 1, $\{+(stock = \text{'}nyse/ibm')\}$, $N_{11}$, $cons = (SB_1 : 0)$"

2. $SB_2$ generates: "$SB_2$, 1, $\{+(stock = \text{'}nyse/*')\}$, $N_{12}$, $cons = (SB_2 : 0)$"

3. $SB_3$ generates: "$SB_3$, 1, $\{+(stock = \text{'}nasdaq/goog')\}$, $N_{13}$, $cons = (SB_3 : 0)$"

4. $SB_1$ generates: "$SB_1$, 2, $\{+(stock = \text{'}nyse/ibm' \wedge p > 90), +(stock = \text{'}nyse/t' \wedge p > 19), +(stock = \text{'}nasdaq/msft')\}$, $N_{11}$, $cons = (SB_1 : 1)$"

The leaf broker sends the incremental change to a broker in its parent node. For example, SHB $SB_1$ and $SB_2$ may send its incremental changes to broker $B_1$ in $N_{21}$ in Figure 4.1, and SHB $SB_3$ may send its incremental change to broker $B_3$ in node $N_{22}$. As brokers in the same tree node are usually fully connected, $B_1$ and $B_3$ forward the incremental change to other brokers ($B_2$ and $B_4$) in the same node ($N_{21}$ and $N_{22}$).

### 4.1.6 Propagating Subscription Changes at An Intermediate Routing Broker

In the previous sections, we have discussed how SHBs handle client subscriptions, assign them virtual start times, generate and propagate the incremental subscription change messages. We describe here the processing of the incremental subscription changes at the intermediate routing brokers.

A broker at an intermediate node maintains a state that consists of a conjunction DAG and a virtual time vector. The virtual time vector contains one element

for each SHB that is in the downstream of the broker's residing node. We denote this virtual time vector as $V^{\mathrm{b}}$. Each element in the $V^{\mathrm{b}}$ vector has an initial value of 0.

Upon receiving an incremental change, a broker checks whether its $V^{\mathrm{b}}$ vector *satisfies* the constraint specified in the incremental change. That is, for each SHB appearing in the constraint, the broker checks the corresponding element in its $V^{\mathrm{b}}$ vector. If those elements have the same values as specified in the constraint, the broker *satisfies* the constraint and can apply the incremental change.

If the constraint is not satisfied, the broker cannot apply the incremental change. Furthermore, if some elements of the broker's $V^{\mathrm{b}}$ vector are smaller than those of the constraint vector's, the broker detects its subscription information is lagging behind, and initiates liveness mechanism to get up-to-date. We defer further discussion of the issues concerning an unsatisfied constraint until Section 4.3.

To apply the incremental subscription change, the broker first applies the additive and subtractive conjunctions carried in the incremental change. For an additive conjunction, if a node with the same conjunction already exists in the DAG, the broker adds the tree node specified in the incremental change to the subscriber list of the existing node. If such a node does not exist, the broker inserts a new node for the additive conjunction. The tree node becomes the only subscribers in the subscriber list of the new conjuntion node.

Vice versa, the subtractive conjunctions are applied by removing the tree node from the subscribers lists of the existing conjunction nodes in the DAG. If the subscriber lists of a conjunction node becomes empty, the conjunction node is

removed from the DAG.

A non-leaf node broker's conjunction DAG (e.g., Figure 4.5(a) and (b)) is similar to that of a leaf node broker's, except that we do not record virtual start time vectors for conjunctions as the intermediate broker does not need to assign virtual start times to subscriptions.

After applying all the additive/subtractive conjunctions, the broker updates its $V^{\mathrm{b}}$ vector by setting the element of the SHB to the value specified in the incremental change. The broker has thus completed applying an incremental subscription change message and can compute a new incremental change for sending to its upstream brokers.

As we have discussed, there could be more than one peer broker in the same upstream routing tree node and we pick **any** broker among these peers to send an incremental subscription change. This broker then forwards the incremental subscription message to its peers, utilizing possibly higher quality intra-node broker-broker physical links. We further delegate to this original receiving broker the task for computing and sending a new incremental subscription change to further upstream nodes.

The chosen broker thus computes all five elements of an incremental subscription change in the following way as we describe:

- the name of the originating leaf broker/SHB. This is the same as in the incremental change this broker received.
- the virtual time at the SHB. This is the same as in the incremental change this broker received.
- the additive/subtractive conjunctions in the new incremental change are

computed as a result of the root node changes in the DAG.

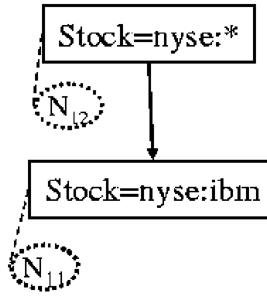
- the subscribing tree node. This is set to the tree node where the current broker resides.
- the new constraint vector. If the incremental change is a pure additive change and no aggregation happened in the current broker, the constraint vector of the new change is unchanged. Otherwise, the constraint vector is set to the old value of $V^{\mathrm{b}}$ vector of the broker prior to its change.

The original receiving broker of the incremental change then forwards the new incremental change to a broker in its parent tree node.

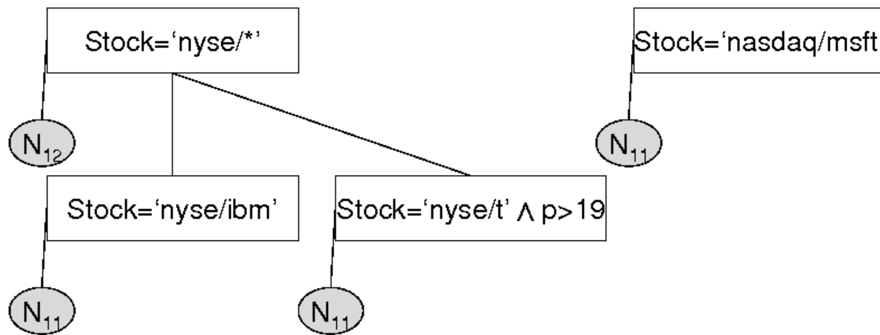
We further illustrate the above process through several examples. For ease of description, we first repeat the incremental subscription change messages generated by the SHBs, which is first described in Section 4.1.5 in Page 75.

1. $SB_1$ generated: "$SB_1$, 1, $\{+(stock = \text{`}nyse/ibm')\}$, $N_{11}$, $cons = (SB_1 : 0)$"
2. $SB_2$ generated: "$SB_2$, 1, $\{+(stock = \text{`}nyse/*')\}$, $N_{12}$, $cons = (SB_2 : 0)$"
3. $SB_3$ generated: "$SB_3$, 1, $\{+(stock = \text{`}nasdaq/goog')\}$, $N_{13}$, $cons = (SB_3 : 0)$"
4. $SB_1$ generated: "$SB_1$, 2, $\{+(stock = \text{`}nyse/ibm' \wedge p > 90), +(stock = \text{`}nyse/t' \wedge p > 19), +(stock = \text{`}nasdaq/msft')\}$, $N_{11}$, $cons = (SB_1 : 1)$"

**Example 4.5.** *In the broker network shown in Figure 4.1, broker $B_1$ has initial $V^{\mathrm{b}}$ vector of all 0's. Hence, when it receives the first incremental changes from $SB_1$ and $SB_2$, the contraints are satisfied and $B_1$ applies the incremental changes.*




(a) State of Broker $B_1$ after Receiving one Incremental Update from each of $SB_1$ and $SB_2$.




(b) State of Broker $B_1$ after Receiving an Additional Incremental Update from $SB_1$.

Figure 4.5: Broker $B_1$ Subscription DAGs

*Its conjunction DAG contains conjunction "stock = 'nyse/ibm'" from $SB_1$ (a.k.a. $N_{11}$ since $SB_1$ is the only residing broker) and conjunction "stock = 'nyse/*'" from $SB_2$ (a.k.a. $N_{12}$). The $V^{\mathrm{b}}$ vector of $B_1$ becomes $(SB_1 : 1, SB_2 : 1)$ after applying the incremental changes.*

*The state of $B_1$ at this stage is shown in Figure 4.5(a).*

*Broker $B_1$ also computes two new incremental subscription changes as a result of the above two messages. These messages are:*

$V_b$ Vector: ($SB_3$:1)

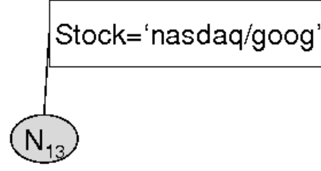



Figure 4.6: State of Broker $B_3$ after Receiving Subscription Incremental Change from $SHB_3$

$SB_1$, 1, $\{+(stock = \text{`}nyse/ibm')\}$, $N_{21}$, $cons = (SB_1 : 0)$

*and*

$SB_2$, 1, $\{+(stock = \text{`}nyse/*')\}$, $N_{21}$, $cons = (SB_2 : 0)$.

□

The following example shows the state at broker $B_3$ after receiving the first subscription change message from $SHB_3$.

**Example 4.6.** *Figure 4.6 in Page 80 shows the state of broker $B_3$ after receiving the first subscription change message from $SHB_3$:*

$$SB_3,\ 1,\ \{+(stock = \text{`}nasdaq/goog')\},\ N_{13},\ cons = (SB_3 : 0)$$

*As a result, broker $B_3$ generates and sends to broker $PB_1$ an incremental subscription change:*

$SB_3$, 1, $\{+(stock = \text{`}nasdaq/goog')\}$, $N_{22}$,$cons = (SB_3 : 1)$.

□

**Example 4.7.** *In the previous Example 4.4, broker $SB_1$ sends the second incremental change when it receives subscription $s_2$. This incremental change contains*

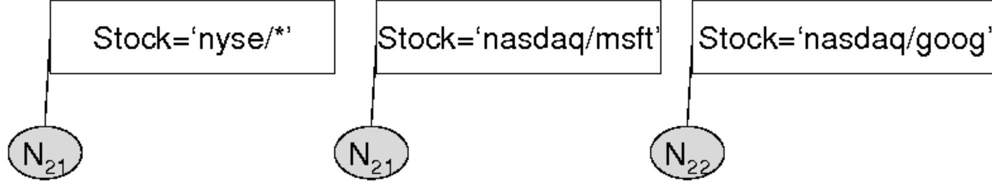



Figure 4.7: State of Broker $PB_1$ after Receiving 3 Changes from $N_{21}$ and 1 Change from $N_{22}$

*two additive conjunctions* $stock = \text{`}nyse/t' \wedge p > 19$ *and* $stock = \text{`}nasdaq/msft'$. *The constraint vector is* $(SB_1 : 1)$.

*When broker* $B_1$ *receives this incremental change, its* $V^{\mathrm{b}}$ *vector satisfies the constraint vector and hence can apply the change by adding the two conjunctions into its DAG. In addition, broker* $B_1$ *sets its* $V^{\mathrm{b}}$ *vector to* $(SB_1 : 2, SB_2 : 1)$. *The state of* $B_1$ *is shown in Figure 4.5(b).*

*Broker* $B_1$ *also computes a new incremental change for sending to its upstream node* $N_{31}$. *As conjunction* $stock = \text{`}nyse/t' \wedge p > 19$ *is covered by an existing conjunction* $stock = \text{`}nyse/*'$, *it is aggregated away. Hence the new incremental change is:*

$SB_1$, 2, $\{+(stock = \text{`}nasdaq/msft')\}$, $N_{21}$, $cons = (SB_1 : 1, SB_2 : 1)$.

□

Similar processing happens at PHB $PB_1$ in the root node of the routing tree. After receiving the first two incremental changes from broker $B_1$ in node $N_{21}$ (Example 4.5) and one incremental change from broker $B_3$ in node $N_{22}$ (Example 4.7), broker $PB_1$'s $V^{\mathrm{b}}$ vector changes to $(SB_1 : 1, SB_2 : 1, SB_3 : 1)$.

After $PB_1$ receives the third incremental change from broker $B_1$ in node $N_{21}$ (Example 4.6), broker $PB_1$ has a $V^{\mathrm{b}}$ vector value of $(SB_1 : 2, SB_2 : 1, SB_3 : 1)$ and a conjunction DAG as shown in Figure 4.7.

### 4.1.7 Data Message Routing

Prior to this section, we have discussed how subscriptions are propagated from SHBs to intermediate brokers to the PHB in the root node. We have illustrated how the brokers maintain their state of $V^{\mathrm{b}}$ vectors and conjunction DAGs according to the subscription changes. In this section, we discuss how this information is used to route data messages.

Data messages are published through the root broker (e.g., $PB_1$) of the routing tree. Before sending a newly published message, $PB_1$ assigns to it a $V^{\mathrm{m}}$ vector. A PHB keeps a persistent record of the highest $V^{\mathrm{m}}$ vector it has ever assigned and ensures that new messages are assigned monotonically non-decreasing $V^{\mathrm{m}}$'s. This highest $V^{\mathrm{m}}$ vector value is restored every time the PHB recovers from a crash failure and is used for assigning to data messages. The PHB's $V^{\mathrm{b}}$ vector can lag behind its $V^{\mathrm{m}}$ vector because unlike the $V^{\mathrm{m}}$ vector, $V^{\mathrm{b}}$ vector (as long as the state conjunction DAG it represents) is not stored persistently. When the PHB recovers after a crash, $V^{\mathrm{b}}$ vector is initialized to all zeros. The recovered broker executes a protocol to retrieve the latest subscription information and updates its $V^{\mathrm{b}}$ vector. Details of this issue is discussed later in Section 4.3.

The advancement of the PHB's $V^{\mathrm{b}}$ vector may change its $V^{\mathrm{m}}$ vector. The $V^{\mathrm{m}}$ vector starts to advance when one or more elements of the PHB's $V^{\mathrm{b}}$ vector are greater than the corresponding elements of the $V^{\mathrm{m}}$ vector. In summary, a PHB's

$V^{\mathrm{m}}$ vector only changes monotonically while a PHB's $V^{\mathrm{b}}$ vector may retrogress and lag behind of its $V^{\mathrm{m}}$ vector.

Other than assigning the $V^{\mathrm{m}}$ vector to messages, the PHB performs routing in the same way as an intermediate routing broker. The following discussion on routing applies to both the PHB and the intermediate routing brokers.

For an incoming message, a broker performs matching to decide to which downstream routing tree nodes it should send the message. There have been a lot of previous research in matching algorithms, including the one studied in the Gryphon system. We do not research content matching algorithms in this thesis.

Furthermore, the broker compares its $V^{\mathrm{b}}$ vector with the $V^{\mathrm{m}}$ vector of the message for each downstream child node. It does so by *slicing* both vectors with only the elements for the SHB's in the subtree rooted at the child node. By *slicing*, we refer to the operation that extracts only the elements in a virtual time vector that are relevant for a set of SHB's.

For the slice of each downstream child node, if the $V^{\mathrm{b}}$ vector is element-wise no less than the $V^{\mathrm{m}}$, the broker sends the message according to the result of the matching algorithm. That is, the broker send a message to the child node if and only if the matching results show a match for the downstream network rooted at the child node. Otherwise, if one or more element in the $V^{\mathrm{b}}$ vector slice is smaller than that of the $V^{\mathrm{m}}$ vector, the broker conservatively sends the message to the child node, regardless of the matching result. This slicing and test of $V^{\mathrm{b}}$ and $V^{\mathrm{m}}$ for each child node is the *sufficiency test* we have discussed in Chapter 3.

When a broker sends a message to its child node, the broker can pick **any** broker in the child node, without worrying about the subscription state of that

broker. This property of our protocol allows the routing layer to make decisions based on system conditions such network utilization and availability, other than the state of subscription information at a downstream broker. It is a unique feature provided by our protocol/algorithm.

As we can see, it is possible for the broker to send down messages that do not match any subscription. This only happens at non-leaf brokers. In the leaf broker, as it always has the latest subscription information, its $V^{\mathrm{b}}$ vector (containing only one element for itself) always satisfies the sufficiency test and only the matching messages will be delivered to subscribers.

**Example 4.8.** *Consider broker $PB_1$ in the network in Figure 4.1, Page 69. $PB_1$'s $V^{\mathrm{b}}$ vector is $(SB_1 : 1, SB_2 : 1, SB_3 : 1)$ before it receives the incremental update triggered by subscription $s_2$ at $SB_1$. Suppose the current highest $V^{\mathrm{m}}$ vector has the same value and is assigned to data message $m_1(nyse/ibm, 95)$ and $m_2(nyse/t, 20)$.*

*$PB_1$ routes these two messages to each downstream node, $N_{21}$ and $N_{22}$. When routing to $N_{21}$, $PB_1$ chooses broker $B_1$ to send message $m_1$ and broker $B_2$ for message $m_2$. Broker $B_1$ has a $V^{\mathrm{b}}$ vector of $(SB_1 : 2, SB_2 : 1)$ and broker $B_2$ has a $V^{\mathrm{b}}$ vector of $(SB_1 : 1, SB_2 : 1)$ due to the delay of the second incremental update originated at $SB_1$ and forwarded by $B_1$. Both broker $B_1$ and $B_2$ will route their data messages to $SB_1$ as the sufficiency tests are satisfied for $SB_1$ and there is a match from node $N_{11}$ at both brokers.*

□

**Example 4.9.** *We consider a case in Example 4.7 where broker $B_1$ forwards the same incremental change to its peer broker $B_2$ and sends $PB_1$ a newly computed incremental change. Broker $B_2$ does not receive the incremental change forwarded*

*by $B_1$ or the message carrying the change is delayed. $PB_1$ receives and processes the new incremental change computed and sent by broker $B_1$. Broker $PB_1$ advances its $V^{\mathrm{b}}$ vector to $(SB_1 : 2, SB_2 : 1, SB_3 : 1)$ while broker $B_2$'s $V^{\mathrm{b}}$ vector stays at $(SB_1 : 1, SB_2 : 1)$.*

*$PB_1$ assigns $V^{\mathrm{m}}$ vector $(SB_1 : 2, SB_2 : 1, SB_3 : 1)$ to $m_3(nyse/ibm, 98)$ and $m_4(nyse/t, 22)$ and sends $m_3$ to $B_1$ and $m_4$ to $B_2$ when it routes messages to node $N_{21}$. At node $N_{21}$, both broker $B_1$ and $B_2$ send their messages to $SB_1$ because $B_1$ finds a matching subscription from $SB_1/N_{11}$ and broker $B_2$, even though without a match for $N_{11}$, detects its sufficiency test fails.*

□

In Chapter 4.2, we will prove the correctness of this protocol by mapping it to the generic algorithm. We first give here the intuition of the correctness.

The sufficiency test is satisfied when a broker's $V^{\mathrm{b}}$ vector is equal to or greater than $V^{\mathrm{m}}$ with regard to the relevant leaf brokers. When it is greater, the broker can have *wider* (matching more messages) conjunctions as new subscriptions may have happened. It can also have *narrower* conjunctions as unsubscriptions may have happened. When the conjunctions are wider, the broker obviously passes all messages matching the subscriptions required by the $V^{\mathrm{m}}$ vectors plus more that match the new subscriptions. In the narrower case, a broker drops messages matching only unsubscribed subscriptions at a leaf broker.

### 4.1.8 Detecting the Subscription Delivery Starting Point

For a new subscription $s$, its SHB must decide a safe point from which the system can deliver a gapless, in-order stream of published messages. As described in

Section 3.2.4, this is to look for a point (message) in a message stream and for every message after this point in the stream, their $S_m$ sets contain the new subscription.

In this protocol, this is accomplished by comparing a message's $V^{\mathrm{m}}$ vector element for this SHB with the virtual start time of the subscription $s$. As soon as the SHB sees a message whose $V^{\mathrm{m}}$ element for the SHB is greater than or equal to the $s$'s virtual start time, the SHB starts to deliver matching messages for $s$.

**Example 4.10.** *Broker $SB_1$ receives $m_1$. Even if $m_1$ matches subscription $s_2$, this is not a delivery starting point for $s_2$ as $m_1$'s $V^{\mathrm{m}}$ vector element for $SB_1$ equals to* 1 *and is less than $s_2$'s virtual start time which is* 2*. This is correct because there is no guarantee a later message (e.g.,* $(nyse/t, 20)$*) will not be filtered out if it had been routed through broker $B_2$. Broker $SB_1$, instead, starts delivery for $s_2$ from message $m_3$.* □

## 4.2 Mapping the Generic Algorithm

We have described a virtual time vector based protocol in the previous section. This protocol is a concretization of our generic algorithm (described in Chapter 3) by providing both an encoding of the subscription sets ($S$ and $S_m$) and constraints, and an implementation of the sufficiency test. We describe in this section this encoding and implementation, and how they are mapped into the concepts in the generic algorithm.

### 4.2.1 Encoding the Subscription Sets $S$ and $S_m$

As described in the previous section, every new subscription is assigned a virtual start time by its SHB. We denote this virtual start time as $v^{\mathrm{s}}$. Conceptually,

a departing subscription is assigned a virtual end time using the SHB's current clock when unsubscription is requested. We denote this virtual end time as $v^{\mathrm{e}}$. For subscriptions that are not yet unsubscribed, their $v^{\mathrm{e}}$'s are undefined.

Every routing broker $b$, including the PHB, maintains a virtual time vector $V^{\mathrm{b}}$, with one element for each SHB in its downstream. The default element value is 0. This virtual time vector is updated when $b$ processes the subscription information.

We now examine the encoding of the subscription set $S$ maintained by a broker $b$.

At any given point of time in the system, consider broker $b$ with virtual time vector $V_b = [(shb_1, v_1), \cdots, (shb_z, v_z)]$, where the SHBs in this vector are in the downstream of $b$. We denote subscriptions entered/unsubscribed at $shb_j$ up to this time as $shb_j.\mathbb{S}$ and $shb_j.\overline{\mathbb{S}}$.

For each outgoing edge $e_i$ of broker $b$, $S(e_i)$ is defined as

$S(e_i) = \{s \mid (s \in shb_j.\mathbb{S}) \wedge (s.v^{\mathrm{s}} \leq v_j)\}$;

and $\overline{S}(e_i)$ is defined as

$\overline{S}(e_i) = \{s \mid (s \in shb_j.\overline{\mathbb{S}}) \wedge (s.v^{\mathrm{e}} \leq v_j)\}$.

where $shb_j$ ($j \in [1..z]$) is a SHB in the downstream of an outgoing edge $e_i$.

Thus, broker $b$'s $V^{\mathrm{b}}$ vector is a concise representation of $S(e_i) - \overline{S}(e_i)$. It denotes that $b$'s conjunction DAG contains the same or covering conjunctions for every *active/unsubscribed* subscription $s$ at $shb_j$ with $s.v^{\mathrm{s}} \leq v_j$ and $s.v^{\mathrm{e}} > v_j$. This is due to that the incremental update originated from $shb_j$ at virtual time $v_j$ always carries a covering conjunction for a conjunction $c$ of the new subscription that has entered at time $v_j$ or the constraint ensures that broker $b$ contains a covering subscription for $c$.

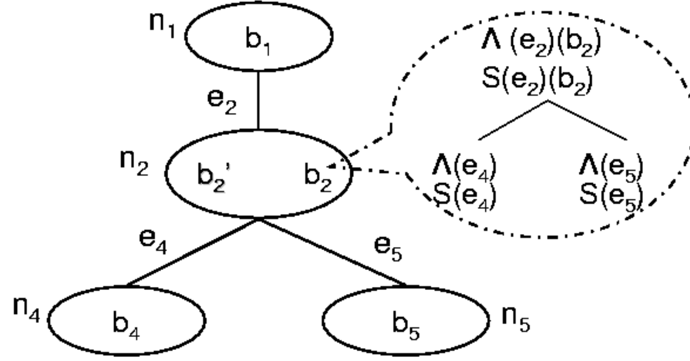



Figure 4.8: Binary Redundant Spanning Tree

The encoding of $S_m$ set of a data message is through its $V^{\mathrm{m}}$ vector. Assume a data message $m$ carries a virtual time vector $m.vv = [(shb_1, v'_1), \cdots, (shb_z, v'_z)]$. The $S(m)$ set of data message $m$ is defined as

$S(m) = \{s \mid \ (s \in shb_j.\mathcal{S}) \wedge (s.v^{\mathrm{s}} \leq v'_j)\}$.

that is, the subscriptions entered at $shb_j$ with virtual start time no later than $v'_j$.

We examine how this algorithm embodies the constraints and sufficiency test.

### 4.2.2 Encoding the Constraints

In this virtual time vector based algorithm, constraints are embodied by requiring that two brokers on each end of an edge $e$ have the same $V^{\mathrm{b}}$ vector values for SHBs that are in the downstream of $e$.

We use broker $b_1$ and $b_2$ in Figure 4.8 to describe this constraint encoding. Figure 4.8 is the same as the previous Figure 3.4 shown in Chapter 3. In the topology shown, broker $b_1$ and $b_2$ are the brokers each in a parent-child node $n_1$ and $n_2$. An edge $e_2$ connects these two nodes.

We declare that this $V^{\mathrm{b}}$ vector value equivalency implies the satisfaction of all constraints computed at broker $b_2$ when $b_2$ applies the incremental change it receives for computing a new incremental change sent to $b_1$.

First, from the definition of the subscription sets, $b_2$ and $b_1$ having the same vector values for SHBs in the downstream of $e_2$ means that

$$S(e_2) = S(e_2)(b_2) = S(e_4) \cup S(e_5) \tag{4.2.1}$$

$$\overline{S}(e_2) = \overline{S}(e_2)(b_2) = \overline{S}(e_4) \cup \overline{S}(e_5) \tag{4.2.2}$$

We have $S(e_2) - \overline{S}(e_2) = S(e_2)(b_2) - \overline{S}(e_2)(b_2)$, constraint II is satisfied.

Type I, III and IV require in broker $B_1$ the existence of some subscriptions that also exist in broker $B_2$. They are always satisfiable if $B_1$ and $B_2$ have the same set of subscriptions.

### 4.2.3 Implementing the Sufficiency Test

The virtual time vector based algorithm embodies the sufficiency test by requiring the brokers to have $V^{\mathrm{b}}$ vector elements that are greater than or equal to the corresponding elements in the message's $V^{\mathrm{m}}$ vector for SHBs in the downstream of $e$.

Consider broker $b_2$ in Figure 4.8 and a data message $m$ received through its incoming edge $e_2$. We prove below that $b_2$ is sufficient for $m$ on an outgoing edge (e.g., $e_4$) if its vector elements are sufficiently large for the relevant SHBs. From the definition of *sufficient* (Definition 3.6), we need to show that $S(m, e_4) - \overline{S}(e_4) \subseteq S(e_4) - \overline{S}(e_4)$. This is satisfied if for every $s \in S(m, e_4) - \overline{S}(e_4)$, $s \in S(e_4) - \overline{S}(e_4)$. We prove this property below.

For every $s \in S(m, e_4) - \overline{S}(e_4)$, $s \in S(m, e_4)$ and $s \notin \overline{S}(e_4)$. From the aforementioned mapping of $S(m, e_4)$, we have $s.v^{\mathrm{s}} \leq m.vv(s.shb)$, where $s.shb$ denotes the SHB at which $s$ subscribes. Because the algorithm requires sufficiency test of $b_2.vv(s.shb) \geq m.vv(s.shb)$, we have $s.v^{\mathrm{s}} \leq b_2.vv(s.shb)$. Hence, from the mapping of $S(e_4)$, we have $s \in S(e_4)$. Because $s \notin \overline{S}(e_4)$, $s \in S(e_4) - \overline{S}(e_4)$. The sufficiency test is satisfied.

## 4.3 Liveness and Failure Handling

We have discussed in this chapter a subscription propagation protocol based on virtual start times and virtual time vectors. We discussed how the protocol operates in the failure-free scenarios. In this section, we discuss how our system detects and handles failures.

The failure-resistant feature of our work comes from two capabilities:

1. That our system is capable of continuing data message routing even in the presence of failures and resulting asynchronous progress in subscription state among peer brokers;

2. That our system is capable of recovering from failures and operating in a more efficient mode.

The first is a direct result of using *sufficiency-directed content-based routing.* Under this scheme, even if the subscription information on a route is out-of-date, our system can still utilize the route to share the load of message traffic. This allows our system to tolerate failures such as link delay and the loss of subscription change messages. It also allows a broker that has just recovered from

a crash failure to participate in message routing without having to wait to get the most recent subscription state. Furthermore, it allows our system to accept new subscriptions and start message delivery for them even though all but one path among a set of redundant paths have failed.

On the other hand, our system functions more efficiently when the subscription information on a routing path is up-to-date or sufficient for the data messages. This avoids the need to conservatively send a message to a broker in the downstream node regardless of whether the broker has found a matching subscription from the downstream. This conservatively-always-send feature is a possible cause of extra messages that may only be discarded later at the SHBs, resulting in wasted network bandwidth.

Our system contains a mechanism for coping with failures and keeping the subscription information on routing paths up to date. As a result, we can recover from failures such as broker crashes, message losses, message arriving out-of-order and message duplication.

We discuss below this mechanism for various failure types.

### 4.3.1 Recovery from Broker Crashes

There are three types of brokers in our network: PHB's, SHB's and intermediate routing brokers. Each type of broker may crash and then recover later. We discuss in this section how these brokers recover from crash. As mentioned in Section 3.1.1 in Chapter 3, a physical broker may assume more than one role, that is, a physical broker may be a PHB, a SHB and an intermediate routing broker. In this case, it performs recovery for each of its roles.

**Persistent State on Brokers**

Brokers may contain some persistent state. There can be two types of SHBs in our system. One type has access to persistent storage and can thus host *durable subscriptions*. Another type does not have access to persistent storage and thus can only host *non-durable subscriptions*.

Up to this point, we have discussed subscriptions that will receive gapless, in order delivery of events only when they are connected to the system. A durable subscription is a type of subscription that will receive such reliable delivery of events even when they are intermittently connected to the system and even when the SHB can crash and recover.

Even though our work has been implemented with support for durable subscriptions, the details of supporting durable subscriptions [13] add little innovation to this work. We hence do not discuss durable subscriptions in full detail. We only discuss here the issues relevant to recovering durable subscriptions from a SHB crash.

For a SHB that hosts durable subscriptions, it stores in persistent storage information on durable subscriptions that have been subscribed through this SHB. Such information includes the content filters and virtual start times of the durable subscriptions as well as what messages have been acknowledged by the subscriptions. Our work only concerns about the content filters and virtual start times.

For a SHB that does not host durable subscriptions, it does not keep a persistent state.

Intermediate routing brokers do not keep any persistent state.

A PHB keeps in persistent storage the highest $V^{\mathrm{m}}$ vector it has assigned to

data messages in order to assign monotonically non-decreasing $V^{\mathrm{m}}$ vectors to data messages even across several crash failures.

#### Virtual Start Times Revisited

In Section 4.1.1, we discussed why the virtual clocks at SHBs should only increase monotonically. To simplify the discussion in the failure-free scenario, we discussed virtual clocks as integer counters. We discuss here the actual virtual clock data structure we used in our work. This virtual clock generates monotonic times even across broker crash failures.

There are many possible techniques to implement a monotonic virtual clock, such as persistently storing an upper bound of the highest clock value. This is not feasible at a SHB that does not have persistent storage and only hosts non-durable subscriptions. We adopt an approach that makes use of a monotonic system clock. The virtual time thus contains a pair of {*current life, integer count*}. The *current life* element is the more significant part of the virtual time and is taken as a snap shot of the SHB's system clock when the SHB starts or restarts after a crash. Thus, no persistent storage is needed. The integer counter takes an initial value of 1 every time the broker starts and only increments when needed. As a result, the virtual times generated are guaranteed to monotonically increase.

We further assume that system clock time never overflows. This assumption is practical as modern systems use 64-bit clocks such as the Java Platform.

#### Crash Recovery at a SHB

The recovery at a SHB includes restoring the persistently stored information about durable subscriptions. The content filters and the virtual start times of the durable

subscriptions are used to populate the initial state of the conjunction DAG. At a SHB that does not host durable subscriptions, this is not needed and the SHB starts with an empty conjunction DAG.

The SHB initializes the virtual clock by taking a snap shot of the system clock and setting the integer counter to 1. The SHB then sends the first incremental subscription change message to its upstream brokers. This message represents all the durable subscriptions. This message is still an incremental change message even though this first initial incremental change contains the whole state of the SHB. The incremental subscription change contains the additive conjunctions of the root nodes of the DAG, and a virtual time of $\{$*current life,* 1$\}$. The SHB then advances its virtual clock to $\{$*current life,* 2$\}$. The SHB is thus fully recovered from the crash.

#### Crash Recovery at an Intermediate Broker

The intermediate routing brokers maintains no persistent state. Upon recovery from a crash, an intermediate broker initializes its $V^{\mathrm{b}}$ vector to all 0's. The conjunction DAG at the intermediate broker is empty. The broker's subscription state (conjunction DAG and the $V^{\mathrm{b}}$ vector) obviously still lags behind the system and will need to catch up. We discuss this issue in the next section on liveness.

#### Crash Recovery at a PHB

Other than assigning $V^{\mathrm{m}}$ vectors to data messages, a PHB performs the same way as an intermediate routing broker. The crash recovery process in the PHB thus includes the steps performed at an intermediate routing broker. In addition, the PHB recovers from persistent storage the highest $V^{\mathrm{m}}$ vector it has assigned

to data messages. This $V^{\mathrm{m}}$ vector will not advance until one or more elements of the PHB's $V^{\mathrm{b}}$ vector exceeds the corresponding element on the $V^{\mathrm{m}}$ vector. As a result, no write to persistent storage is needed if the $V^{\mathrm{m}}$ vector does not change. The $V^{\mathrm{m}}$ vector takes the maximum of the $V^{\mathrm{m}}$ and $V^{\mathrm{b}}$ elements. As we will discuss later in this section, the liveness mechanism will eventually bring the PHB's subscription state up-to-date. As a result, the PHB's $V^{\mathrm{b}}$ vector is typically greater than its $V^{\mathrm{m}}$ vector.

### 4.3.2 Recovery from Subscription Message Losses

Subscription change messages may get lost during transmission. As a result, one or more elements of a broker's $V^{\mathrm{b}}$ vector may lag behind the SHB's virtual clock by more than 1. As discussed in the previous section, this can also happen in a PHB or an intermediate routing broker that has just recovered from crash.

In this section, we discuss how the out-of-date $V^{\mathrm{b}}$ vectors in brokers are discovered and the liveness mechanism our system uses to bring these brokers to up-to-date.

Out-of-date $V^{\mathrm{b}}$ vector (and hence the broker's subscription state) can be detected through several means:

- A SHB receives a data message with a $V^{\mathrm{m}}$ vector in which the element for this SHB is smaller than the SHB's virtual clock time $-$ 1. This indicates that the PHB of the data message has a lower $V^{\mathrm{m}}$ vector element and hence a lower $V^{\mathrm{b}}$ vector element.
- A PHB or an intermediate routing broker receives a data message with a $V^{\mathrm{m}}$ vector in which some elements have greater values than the broker's $V^{\mathrm{b}}$

vector.

- A PHB or an intermediate broker receives a subscription change with a constraint vector in which some elements have greater values than the broker's $V^{\mathrm{b}}$ vector.

Accordingly, our liveness mechanism can be driven by either SHB or a non-SHB.

**Leaf Broker/SHB Driven Liveness**

As the sources of subscription changes and virtual times, leaf brokers ensure all publisher connecting brokers receive up-to-date subscription information and assign latest $V^{\mathrm{m}}$ vectors to data messages. For subscription/unsubscription received during virtual time $v^{\mathrm{s}}$ and propagated, a leaf broker *SB* maintains an expected starting time from which data messages should have $V^{\mathrm{m}}$ vector whose element $v^{\mathrm{s}\prime}$ for *SB* is greater than $v^{\mathrm{s}}$. This expected time can be dynamically adjusted through similar techniques that estimates TCP round trip delays. Messages received after the elapsed time with $v^{\mathrm{s}\prime} < v^{\mathrm{s}}$ trigger a full subscription state update with *SB*'s latest propagated virtual clock time. Alternatively, *SB* can repeat the incremental updates sent from $v^{\mathrm{s}\prime}$ to $v^{\mathrm{s}}$. This requires a cache for the latest incremental updates at *SB*.

**Non-Leaf Broker Driven Liveness**

As described above, a PHB or an intermediate routing broker $b$ can detect it has an out-of-date $V^{\mathrm{b}}$ vector by receiving a data message with some higher $V^{\mathrm{m}}$ vector elements or a constraint vector with some higher elements.

The broker $b$ uses negative acknowledgements. It initiates a negative acknowledgment message toward the leaf broker $SB$ for whom $b$'s $V^{\mathrm{b}}$ vector is lagging behind. Such negative acknowledgment may be satisfied by $SB$ or a broker on the route from $b$ to $SB$ with the required subscription information.

### 4.3.3 Recovery from Subscription Message Arriving Out-of-Order

Because a subscription change message contains a virtual time from the originating SHB, subscription change messages arriving out-of-order can be easily detected by comparing the integer counter part of the virtual time. The *current life* element of the virtual time should be the same as the latest the broker has received except for the first subscription change message with an integer counter of 1. Thus, a broker can always accept a subscription change message with a virtual time that has a higher *current life* element than the broker's current $V^{\mathrm{b}}$ vector element. The broker can also accept a subscription change message if the message's virtual time value has the same *current life* element and the integer counter value is 1 greater than the broker's $V^{\mathrm{b}}$ vector element. In other case, the subscription change message is arriving out-of-date.

If the out-of-order message is in the future with a higher virtual time, the message may be buffered and processed later after the messages with smaller virtual times have arrived and been processed. The liveness mechanisms described above will ensure that the missing messages with smaller virtual times will eventually arrive. If the out-of-order message is out-of-date, it can simply be discarded.

### 4.3.4 Recovery from Subscription Message Duplication

Similar, duplicate subscription messages can be easily detected by examining their virtual times. They do not need to be processed and can be discarded.

## 4.4 Summary

In this section, we have discussed a working subscription propagation and content-based routing protocol based on subscription virtual times and virtual time vectors. We have explained how this protocol can be mapped to the generic algorithm. We have also discussed its fault tolerance characteristics and failure handling mechanism. We will present the implementation details and evaluation of this protocol in Chapter 6.

# Chapter 5

# Dynamic Access Control

In the previous chapters, we have discussed client subscription information as a type of control information that can affect message routing in a publish/subscribe system. By propagating and maintaining subscription information to/at routing brokers, the system can optimize message routing and thus send fewer wasted messages as compared to the flooding scheme. Similar to subscription information, subscribing client access control is another type of control information that can affect message delivery. By propagating subscriber access control information to routing brokers, further savings in communication cost may be achieved.

In this chapter, we discuss the issue of client access control in content-based publish/subscribe systems. We will define what *dynamic access control* means and its implication for reliable delivery. We first present a service model of dynamic access control that provides deterministic service guarantee of message delivery. We then present an algorithm that implements this model.

## 5.1 Deterministic Service Model

We present in this section a deterministic service model of dynamic access control. We describe the various entities involved in dynamic access control and their roles, a content-based form for specifying access control rules and the clear starting points of access control changes.

### 5.1.1 System Entities & Content-based Rules

In our service model, there are two types of entity that are involved in access control.

**Security administrator** The security administrator is the ultimate authority of access control in the system. The security administrator decides (based on external factors such as client service contracts) the access rights for client principals (defined below) and/or whether there should be any change to their existing access rights. The security administrator instructs the system of his/her decisions through an administrative interface.

In a large system, there may be multiple security administrators. As the changes made by each administrator may affect overlapping sets of clients, the system should accept the changes in a transactional and serializable manner. For the purpose of this thesis and simplicity of discussion, we consider the security administrators as an abstract single entity that initiates a single sequence of policy changes.

**Client principals** Clients in our system have associated principals which is decided/verified by the system through authentication when clients connect. A client

can connect to the system, publish messages or subscribe and receive messages. The client's capability to connect, publish and/or subscribe/receive messages is regulated by the access rights of its principal. For example, if a client is interested in receiving stock quotes, financial news and reports of IBM Corp but its principal has only access rights to stock quotes, the client will not receive any news and reports even though it requests them.

The access control rules in our system are associated with principals. Multiple publishing/subscribing clients running on behalf of the same principal can connect at different places in the system. There are two types of principals in our system, *group* and *individual*. A group principal is a collection of individuals or recursively, other group principals. Access rights granted to a group principal are automatically granted to all members of the group, and recursively to the members of a member group.

**Content-based form of Access Rights** The access rights of a principal include the right to connect, the right to publish and the right to subscribe to and receive messages. We adopt a content-based form for specifying access control rules of these three rights. An access control rule takes the following form of a tuple of three elements:

[Principal, Access type, Content filter]

A rule of such form specifies that a principal has the right to *connect* to the system, *publish* or *subscribe* to messages matching a content filter. While publish and subscribe rules can take a non-trivial filter, connect rules are specified with $true$ or $false$ to indicate the right to connect or not. For example, the rules that

allow a principal *John Doe* to connect and subscribe to stock quotes are specified as follows:

[John Doe, Connect, True]

[John Doe, Subscribe, type='quote']

The access control rules are maintained internally in *positive* forms in that all rules specify what a principal is allowed to do. Negative forms specifying what a principal is not allowed to do are provided as a convenience to security administrators and are converted internally to positive forms by taking the negation of the content filters.

A publishing/subscribing client on behalf of a principal is allowed to publish messages that match the publishing rules of its principal and is allowed to receive messages that match BOTH its subscription filters and the subscribing rules of its principal. This allows the system to provide 1) **information authenticity** by allowing only authorized sources to publish messages; 2) **information confidentiality** by only distributing messages to authorized subscribers; 3) **protection against denial-of-service (DoS) attacks** initiated by malicious subscribers who request a large number of messages that are only going to be discarded. This large number of messages can result in congestion in the network and impair the system's capability to serve other clients.

**Group Access Control** Group and individual principals share the same form of connect, publish and subscribe access rights. In addition, a new type of rule, *member list*, exists for group principals. For example, a *premium subscribers* group that includes *Jane Smith* and *James Brown* and has subscribing rights to all stock

quotes, news and reports has the following access control rules:

[Premium group, Member list, $\{$Jane Smith, James Brown$\}$]

[Premium group, Subscribe, type='quote' $\vee$ type='news' $\vee$ type='report']

All members in a group are automatically granted the access rights of the group. Thus, the access rights of an individual principal are the union of the individual's rights and the rights of all group principals it belongs to. Hence, *Jane Smith* and *James Brown* will have access to all stock quotes, news and reports in addition to other access rights they are granted.

### 5.1.2 Clear Starting Points of Access Control Changes

We present in this section our deterministic service model that provides clear starting points for access control changes. In this model, access control rules/changes are initiated by the security administrator at the administrative console and stored into a persistent storage called **ACL DB**. At any time, the security administrator may specify a number of changes pertaining to one or more principals. All these changes are considered as a batch that must be enforced atomically. After the security administrator confirms each batch of changes, the changes are propagated throughout the broker network.

A broker can host one or more message streams. The publishers can connect to any PHB in the system and are assigned to any stream by the PHB. Each stream contains in-order the messages published by one or more publishers. For each of these streams, the broker picks a starting point to enact the new access control rules. The starting point is chosen in a way such that: 1) successive batches of changes get later starting points; and 2) the starting point is late enough so that

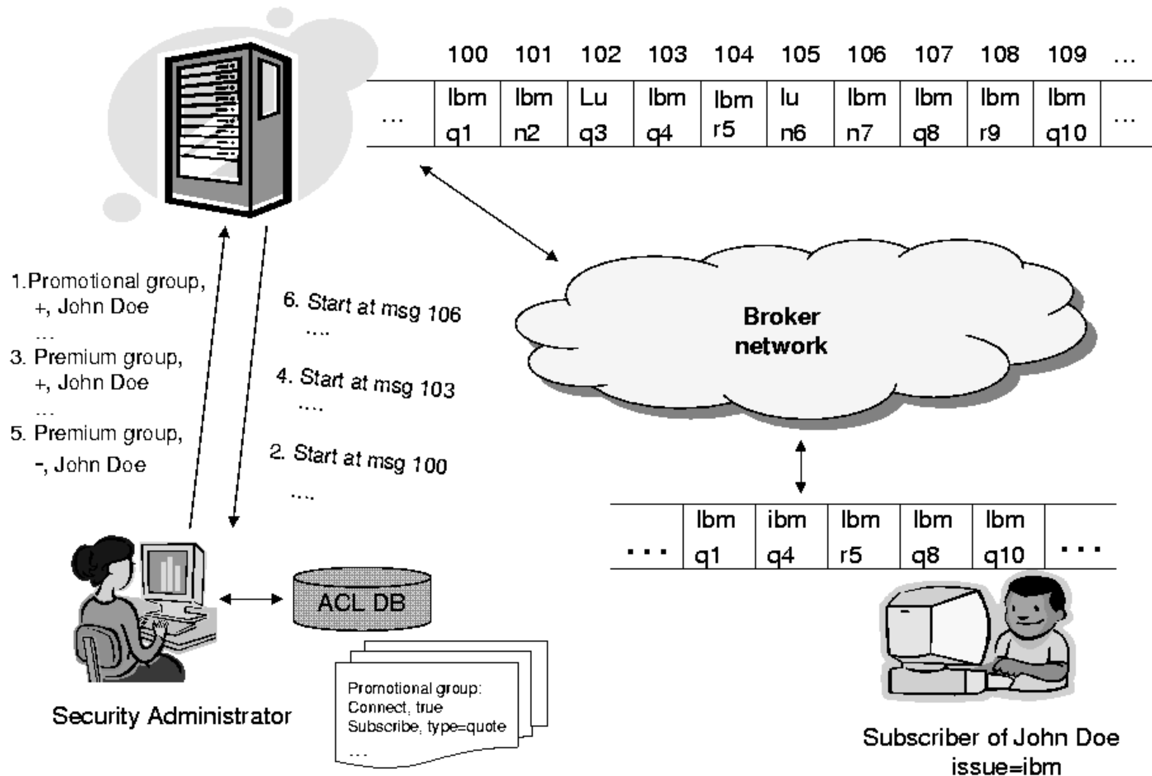



Figure 5.1: Service Model of Dynamic Access Control

no message after the starting point could have been delivered according to the old rules. This constraint can be easily achieved by designating a newly published message on the stream as the starting point. The starting point information is sent back to the security administrator for future inquiries and references. The new rules are enforced uniformly throughout the system on all messages after the starting points, no matter where the publishing/subscribing clients on behalf of the affected principal(s) connect.

We illustrate the effect of an access control policy change using an example in which a principal *John Doe*'s subscribing rights went through 3 phases of changes:

1. *John Doe* became a member of the promotional group which had subscribing access only to stock quotes; 2. *John Doe* became a premium subscriber and subsequently gained subscribing access to all three types of financial information; 3. *John Doe*'s premium subscription expired and as a result he lost subscribing rights to financial news and reports.

Shown in Figure 5.1, a subscriber on behalf of principal *John Doe* connected to the system and requested a subscription of issue='ibm'. Under the service model, every time the access rights of *John Doe* changes, the system provides a clear starting point in each message stream such that 1) a message *before* the starting point is delivered to the client if and only if the message satisfies both the subscription filter and access right filter *before* the change; and 2) a message *after* the starting point is delivered to the client if and only if it satisfies both the subscription filter and access right filter *after* the change. In the message stream shown in Figure 5.1, if the starting points chosen are message 100 for the first access change, message 103 for the second access change and message 106 for the third change, the messages delivered to the client will be 100, 103, 104, 107, 109. Notice that non-quotes are only delivered in the range $[103, 105]$. In a system that has more than one message stream, this activity happens to all streams, each with its individual start points.

### 5.1.3 Subscription Propagation, Content-based Routing and Reliable Delivery

A valid implementation of access control can be one in which the PHB and intermediate brokers forward all published messages that match client subscriptions

to SHBs, and SHBs enforce access control by delivering messages that match not only a client's subscription but also its access rights. Such a solution will be a perfectly correct implementation, but it may waste considerable bandwidth sending messages that will be later discarded.

As described in the previous chapters, subscription propagation is an optimization which may result in fewer wasted messages being sent to SHBs in exchange for requiring the PHB and intermediate brokers to acquire knowledge about subscription predicates and perform filtering. By propagating clients' access rights along with their subscriptions, further savings in communication cost may be achieved.

In Chapter 3 and 4, we described protocols for subscription propagation. These protocols preserve reliable delivery and enable free routing choices on any of the redundant paths for system availability and load sharing. Furthermore, we point out that for a subscription, its reliable delivery starting point on a published message stream can be chosen as any point in the stream provided that none of the messages after the starting point has been acknowledged so that the system may have reclaimed the persistent storage occupied by the message.

In the next section, we use the reliable delivery and subscription propagation protocols as building blocks for constructing an efficient and highly available distributed protocol that enforces the deterministic semantics of dynamic access control to publishing/subscribing clients. We adopt a domain-based trust model. All brokers within the same domains trust each other. Brokers that do not trust each other should be put into different domains and cross-domain communication is regulated by assigning access control rules according to their trust levels. For simplicity, we discuss the protocols under one trusted domain. This is of practical

use as in a lot of commercial cases, publish/subscribe systems are deployed in a managed environment under the complete control of an administrator. The work can be extended to multiple trusted domains by treating a domain as a special publishing/subsubscribing client and assigning a principal to the domain. The clients connected to the system through an un-trusted domain can only access messages that satisfy both the domain's right and their own access right.

## 5.2 Protocol Description

Our protocol provides the deterministic service guarantee of message delivery through 1) distributing access control information to brokers that host relevant principals; 2) restricting publishing activities by accepting only messages satisfying the publisher's publishing rights; 3) restricting client subscriptions using their subscribing rights; 4) propagating restricted subscriptions and hence enforcing access control in the routing brokers by performing content filtering on both the clients' subscriptions and access rights; 5) final enforcement of access control at the SHB. We describe each of these aspects.

### 5.2.1 Protocol Overview

As mentioned in Section 5.1.2, each batch of access control changes made by the security administrator is stored into a persistent storage called ACL DB and enforced atomically. The ACL DB assigns a control version to identify an atomic batch of changes and therein a new state of access control rules resulted. The control version is an integer that is incremented for each batch. The ACL DB distributes each control version of changes along with the version number.

The access control changes may be specified in the form of incremental changes to avoid sending the full state of access control rules at each version. In addition, there is no need for a PHB or SHB to receive an access control change if principals connected at it are not affected by the change. As a result, the distribution of access control changes is provided by reliable delivery and optimized by letting PHB/SHB establish a subscription to access control of each connected principal.

The edge brokers (PHB/SHB) maintain in a cache the access control rules for all principals that are currently connected. Upon the connection of a client on behalf of a new principal, the edge broker performs a request/reply protocol to retrieve an initial version of access control rules for the principal. The broker also establishes a subscription to receive future access control changes concerning the principal. The subscription is established in a way that the broker will receive every access control version change after the initial version. Our subscription propagation protocol can guarantee this property.

Upon receiving a new version of access control changes, a broker updates its cache by applying the changes to the rules of the affected principals. In addition, each broker hosting a message stream picks a position in the stream as the starting point of the new control version and notifies the security administrator of the starting point. There is one such starting point for each message stream. The starting points can be stored into the ACL DB to provide answers to future inquiries.

For every message received from a publisher, the PHB first checks whether the message matches the publishing rights of the principal and rejects any illegal publications. For every legitimate message after the starting point of a new control

version, the PHB sends the message together with the new control version. As existing messages on the stream may have already been sent with old access control versions and delivered according to the old access control rules, the starting point is chosen as the position immediately after the latest message in the stream.

At each SHB, the original client subscriptions are intersected with the appropriate subscribing rights of the principal. Instead of propagating the original subscription filters, the SHB propagates this restricted subscription filters and the version of the subscribing right applied. As a result, the intermediate broker does not need to maintain any access control rules but still participates in content filtering to avoid sending messages to where there is no legitimate subscribers. The intermediate broker does maintain a vector of access control versions aside from the restricted subscription filters, with one element for each downstream SHB. An access control version in the vector identifies the most recent version of access control rules from which the restricted subscription filters were created. This vector of access control versions are compared to the control version of a message at content filtering and routing time. If every element of the vector is no less than the control version of the message, the intermediate broker routes the message the same way as before (i.e., according to content matching and subscription sufficiency test as described in Chapter4). Otherwise, as a conservative measure, the broker sends the message toward the SHB for which the intermediate broker does not have a sufficient access control version in the vector. That is, the enforcement of access control is delayed to the downstream brokers. If a message with a higher access control version arrives at a SHB, the SHB delays the processing of the message until it receives the access control information that matches the message's.

In the rest of this section, we describe the protocol in details through an illustration. We use examples of subscribing rights as it presents the most challenges.

### 5.2.2 Distributing Access Control Information

As previously mentioned, access control information may take the form of incremental changes and as a result, should be delivered in order and without gaps.

The distribution of access control information leverages the reliable delivery service provided by the underlying publish/subscribe system. The access control messages are published into a reliable message stream. There are two access control messages for each version of change: one on the rule changes and one on the access control version change.

For example, the first access control change in Figure 5.1 grants subscribing right to messages matching type='quote' to clients of principal *John Doe*. If this change increments the latest access control version from 10 to 11, the following two messages will be published by the ACL DB:

Type: ACL Change

Principal_list: {John Doe}

Version: 11

ACL_changes: {John Doe, subscribe, +, type='quote'}

and

Type: ACL Version Update

Version: 11

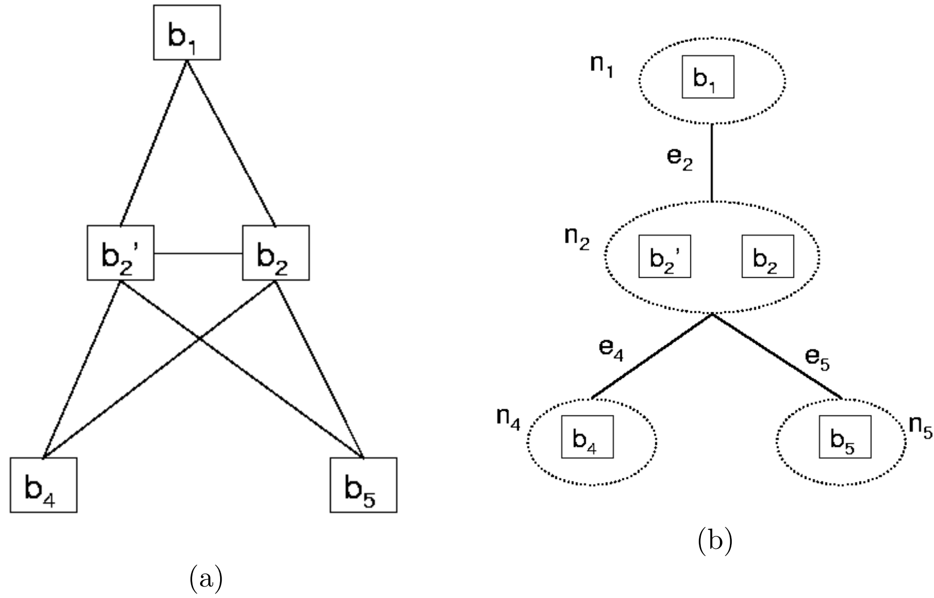



Figure 5.2: Redundant Routing Networks

To receive the updates from the ACL DB, an edge broker establishes a subscription to receive access control version update and for each principal whose clients are currently connected at the broker, a subscription to the incremental access control changes concerning the principal. For example, if a subscriber of *John Doe* connects at broker $b_5$ in Figure 5.2, the subscriptions for access control information at $b_5$ are

type='ACL Version Update'

and

type='ACL Change' $\wedge$ Principal_list contains 'John Doe'

These two subscriptions are submitted with reliable delivery requirements and once established in the system, guarantee that $b_5$ receives every version update and every access control change for each connected principal.

### 5.2.3 Handling Client Subscriptions

A precondition of any security mechanism is the authentication process. Through authentication, a client establishes an identity with the system and security mechanisms operate in terms of the identity. Authentication is an essential part of system security and can be achieve through various ways such as use of a digital certificate. The details of authentication is beyond the scope of this work. Our solution to providing the deterministic service model of dynamic access control can use any authentication mechanism. The end result of the authentication process is to associate a principal with a connecting client.

Upon receiving a client connection with principal $p_1$, the edge broker (e.g. $b_5$) searches for the access control information for $p_1$ in its local cache. If there are other connected clients of $p_1$, access control information for $p_1$ is already maintained by this broker. If no such client exists, $b_5$ executes a protocol to retrieve an initial version of access control rules for the principal. This initial state retrieval is performed through a two-step message exchange:

1. Broker $b_5$ publishes a message requesting an initial access control state of principal $p_1$. The ACL DB is a subscriber to this initial state request and as a result, receives $b_5$'s request.

2. the ACL DB replies with all the access rights of $p_1$ and all the groups $p_1$ belongs to as of the latest control version $v_1$. $b_5$ submits subscriptions for receiving the access control changes of $p_1$ and of the groups $p_1$ is a member. Broker $b_5$ designates an explicit delivery starting point to be the first message after the access control change of version $v_1$.

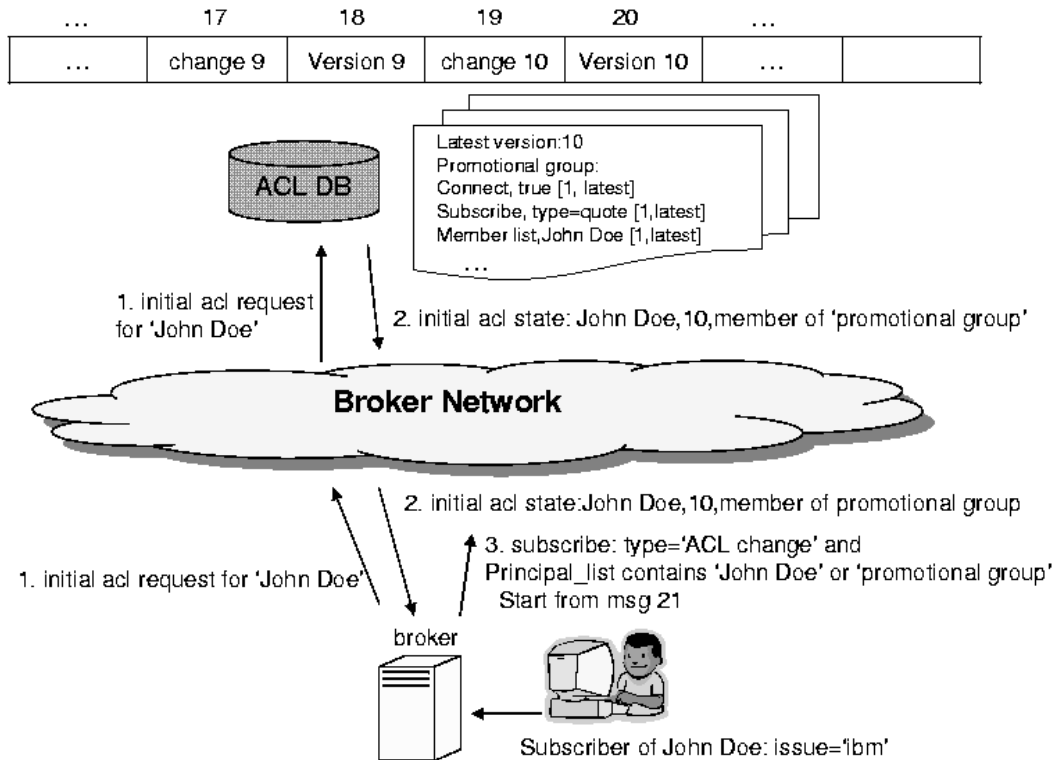



Figure 5.3: Retrieve Initial Access Control for a New Client/Principal

The designated delivery starting point of $b_5$'s subscriptions guarantees that even in the presence of communication delays, $b_5$ receives all the access control changes of $p_1$ and its belonging groups after the initial start version $v_1$. The subscription propagation and reliable delivery described in Chapter 3 guarantees the validity of the starting point.

Figure 5.3 depicts the steps of this process.

After establishing an initial version of access control for a client, the edge broker can process its subscriptions and published messages. Every published message is matched against the client's most recent publishing rights and is only accepted if the message matches the filter of the publishing right. The accepted

message is entered into the message stream and assigned an access control version number using the latest version the broker maintains.

For each subscriptions received from the client, the edge broker computes a restricted filter by intersecting the subscription filter with the filters of the principal's subscribing rights. For example, the client in Figure 5.3 has subscribing right of type='quote' at access control version 10 and submits a subscription *issue='ibm'*. The restricted subscription is type='quote' $\wedge$ issue='ibm' and is assigned control version 10. The edge broker maintains the restricted subscription in its conjunction DAG and performs subscription propagation using the restricted subscriptions.

### 5.2.4 Propagating Restricted Client Subscriptions

As the access control changes, the system may be routing messages with multiple control versions. The SHBs thus may maintain multiple restricted client subscriptions, one for each of these control versions.

The brokers (PHB/SHB) only need to maintain the latest access control rules. This is because a PHB only appends messages at the end of a message stream and thus uses the latest publishing rights and control version and at a SHB, the delivery for a new subscription only starts with messages of the latest control version. As a result, a SHB propagates a restricted subscription of the latest subscribing rights. When the SHB propagates the restricted subscriptions, the propagation message includes an access control version from which the restricted subscription is computed, as well as the subscription. As the SHB only uses the latest control version, the control version information in the subscription propagation message

is monotonically non-decreasing.

Assume the example in Figure 5.3 where the subscriber of *John Doe* connects at broker $b_5$ in the topology shown in Figure 5.2. The subscription message broker $b_5$ propagates includes control version 10 and a filter type=‘quote’ $\wedge$ issue=‘ibm’.

The intermediate brokers maintain a vector of access control versions. The vector contains one element for each SHB that is in the downstream of the intermediate broker. This vector is updated when the broker receives a subscription propagation message with access control versions that are greater than the elements in the vector. Since the access control versions of the messages are monotonically non-descreasing, this vector of access control versions at an intermediate broker is also monotonically non-descreasing. The intermediate broker applies the information in the subscription message to update the conjunction DAG and propagates the subscription changes to further upstream.

Continue with our example, broker $b_2$ and $b_2'$ may maintain a control version vector of $\{b_4 : 9, b_5 : 10\}$ when they have received from $b_5$ the restricted subscriptions of control version 5 but $b_4$ may still lag behind at control version 9.

### 5.2.5 Handling Dynamic Access Control Changes

As described in Section 5.2.2, each access contorl change is distributed through two messages: a message for the rule change and a message for updating the latest access control versions of the receiving brokers. We discuss in this section the processing at a broker upon receiving these two types of messages.

As describe earlier in this chapter, an access control rule change is only distributed to brokers at which there are clients on behalf of principals that are

affected by the change. The access control rule change carries a version number $v$. Upon receiving such a message, a SHB applies the change to obtain the access control rules of the connected principals at version $v$. If the access control change removes a connected principal from a group $g$, the SHB removes the subscription to the access control of $g$ if there is no other connected principal that is a member of $g$. If the access control change adds a connected principal to $g$, the SHB retrieves access rules of the group at version $v$ if no information is maintained for $g$. Such an inquiry of the access control rules of a principal at a certain version is sent to the ACL DB which maintains the access control rules of each version. In this case, the new subscribing filter of an affected member includes the group's subscribing filter.

For every principal that is affected by the access control change, the SHB recomputes restricted subscriptions using the access control rules at version $v$. These restricted subscriptions are assigned the latest received access control version $v$. The restricted subscriptions are then entered into the broker's conjunction DAG as a single atomic batch to ensure the atomicity of the enforcement of the new version of access control rules. Any changes to the root nodes of the conjunction DAG is propagated further upstream with the control version $v$ through the subscription propagation mechanism.

In our example, when broker $b_5$ receives access control change 11 that admits *John Doe* into the premium group, $b_5$ retrieves the premium group's access control rules at version 11 and recomputes the restricted subscriptions. As a result, subscription ((type='quote' ∨ type='news' ∨ type='report') ∧ issue='ibm') at version 11 is entered into conjunction DAG and exists along with the former subscription

of (type=‘quote’ $\wedge$ issue=‘ibm’) at version 10. This subscription is propagated further upstream and updates the data structure at broker $b_2$ and $b_2'$ with the new subscription and control version vector $\{b_4 : 9, b_5 : 11\}$.

Different from access control changes that are only distributed to SHBs with affected principals, access control version update is delivered to every SHB. A SHB with affected principals ignore this message because its control version and subscription change is already updated and propagated as previously discussed. A SHB without affected principals assumes an emtpy set of restricted subscriptions for the new version of access control and propagates a subscription change message to update the access control version vector of the upstream routing brokers'.

### 5.2.6 Routing Data Messages

As described in Chapter 3 and 4, data message routing decisions are computed based on content matching and broker subscription information. To provide the deterministic service guarantees for dynamically changing access control rules, the data messages are assigned monotonically non-decreasing access control versions at the PHB. This access control version is compared with the version numbers on a routing broker's access control version vector. If the vector elements for some SHBs are less than the message's control version, indicating the broker does not have sufficient restricted subscriptions from the access control rules required by the message's contrl version, as a conservative measure, the broker sends the message toward those SHBs regardless of the result of content matching. That is, the enforcement of access control is defered to downstream brokers. Otherwise, routing decisions are made based on the result of content matching (and broker

subscription information).

If broker $b_2$ in Figure 5.2 receives data message 100 of (lu, n6) with control version 11, $b_2$ does not need to send this message to $b_5$ as there is clearly no client at $b_5$ that has subscribed to this message AND has access to it. However, $b_2$ sends this message to broker $b_4$ regardless because its control version vector has 9 for $b_4$ which is less than 11. This feature of only using content-based routing when there is sufficient information is what enables the high availability of the system as messages can be routed through any broker regardless of its state of subscription information.

### 5.2.7 Enforcing Access Control

The enforcement of access control is distributed throughout the broker network by propagating restricted subscriptions that are intersections of the original client subscriptions and a version of access control rules. The routing brokers (other than SHBs) may enforce access control rules by filtering out messages that do not match any restricted subscriptions. The routing brokers may also defer the enforcement by sending the messages in the case it does not have a sufficient large access control version vector for the message. It is up to the SHBs to perform the ultimate enforcement of access controls.

A SHB, upon receiving a data message, compares its latest access control version with that of the message's. If the SHB access control version is smaller, the SHB must delay further distribution of the data message until an access control change with a sufficiently large version has been received. As the access control changes are distributed using reliable delivery, those messages are guaranteed to

arrive and the SHB will then be able to continue processing of the message.

The SHB first performs matching of the data message. This produces a list of restricted subscriptions that match the data message. For each restricted subscription, if its access control version is exactly the same as that of the data message's, the SHB delivers the message to the client whose original subscription produced the restricted subscription. Otherwise, the data message is not delivered.

In the example in Figure 5.1, even though broker $b_5$ (where the client of *John Doe* connects) may receive message 108 with control version 12, the message will not be delivered because the matching restricted subscription has a different control version 11.

### 5.2.8 Sunsetting of Access Control Rules/Versions

As described earlier in this chapter, for each client subscription, a SHB may maintain multiple restricted subscriptions, one for each access control version for which the system is currently routing a data message. The SHB may decide when it is safe to remove a restricted subscription based on what messages the SHB has delivered to the subscribers connected at it. This decision can be made totally locally as the SHB can check the control versions of the latest data messages on each message stream it has delivered to the subscribers. Restricted subscriptions with control versions less than the minimum of the control versions of these data messages can be discarded. The SHB modifies its conjunction DAG by removing the conjunctions of these restricted subscriptions. If the root nodes of the conjunction DAG is changed, the SHB will propagate the subscription change to its upstream brokers.

## 5.3 Summary

In this chapter, we have described the problems of enforcing dynamic access control in a content-based publish/subscribe system. We described a deterministic service model of dynamic access control with regard to reliable delivery. We then presented an algorithm that implements this service model using the existing subscription propagation and relible delivery service.

In the next chapter, we present the implementation and evaluation of this protocol along with that of the subscription propagation protocol described in Chapter 4.

# Chapter 6

# Implementation and Evaluation

In Chapters 4 and 5, we described protocols that support subscription propagation, content-based routing, reliable delivery and dynamic access control with deterministic service guarantee in a publish/subscribe system with redundant paths. Using these protocols, our system tolerates failures by allowing redundant brokers to work interexchangeably and yet does not need to reach consensus among these brokers in the system.

In this chapter, we discuss the most important implementation issues. These issues range from the kind of covering test we use to the mechanisms that optimize system performance. We then present both analytical and experimental evaluation of our subscriptioin propagation and dynamic access control protocol.

## 6.1 Implementation

We have implemented our work in the Gryphon system [10, 4]. The Gryphon system is a pure Java implementation of a content-based publish/subscribe system

with native I/O libraries for optimized I/O performance. In this high performance multi-threaded environment, we have encountered and addressed issues including performance bottlenecks and hot lock contention points. In this section, we discuss these problems and our solutions after a description of the covering test algorithm we used.

### 6.1.1 Covering Test

In our work, subscription aggregation is based on covering relationships between conjunctions. There are algorithms for implementing covering relationships such as described in [41] and [19]. It is an open issue as to how efficiently and completely covering tests can be performed.

Our work can use any algorithm that tests the covering relationship between a pair of conjunctions. For our implementation, the requirement on the covering test is *safety* rather than *completeness*. That is, if a covering test algorithm states the existence of covering relationship between a pair of conjunctions, such a relationship **must** exist. On the other hand, if a covering test algorithm does not find the existence of covering relationship between a pair of conjunctions, such a covering relationship may in fact exist. This property of our protocol offers the flexibility to choose a wide range of covering test algorithms.

The completeness of the covering test algorithm affects the number of conjunctions a broker maintains for performing content matching on data messages. It does not change the number of data messages that a broker will route because a covered conjunction, although propagated, is totally redundant. For example, if a covering test algorithm can not discover that conjunction $stock = `nyse/*'$

covers conjunction $stock = `nyse/ibm'$, both conjunctions will be propagated to and maintained by upstream routing brokers instead of just $stock = `nyse/*'$. However, this does not change the fact that a broker with these two different sets of conjunctions will route a data message of $(stock : `nyse/t', p : 19)$ but not $(stock : `nasdaq/msft', p : 25)$. As we will see in Section 6.3, the matching algorithm performs very efficiently even when the number of subscriptions is fairly large (e.g., $10^5$ subscriptions).

In our implementation, we use a covering test algorithm that is based on the *equality of conjunctions*. That is, we test the equality of the stringified representations of two conjunctions after performing *syntax normalization*. The *syntax normalization* of conjunctions refers to the analysis and transformation that disregard the syntax differences such as the difference between $stock = `nyse/ibm' \wedge p > 90$ and $p > 90 \wedge stock = `nyse/ibm'$.

Designing a sophisticated covering test algorithm is a challenge that is outside the scope of this work. However, because we only assume safety of the covering test output, our implementation provides a good test-bed for making new incremental changes to existing covering test algorithms. For example, one may extend our equality-based covering test algorithm to include regular expression matching of a string pattern that can identify the covering relationship between $stock = `nyse/*'$ and $stock = `nyse/t'$.

The research leading to more sophisticated covering test algorithm is a future direction of this work. In the performance evaluation, we used subscriptions that only covers each other when they are exactly the same. As a result, the covering test algorithm we used is as good as any sophisticated algorithm for purposes of

our experimental analysis.

### 6.1.2 Multiple Publisher Hosting Brokers

For simplicity, we have described our protocol from the standpoint of a single PHB. The routing tree has a single root and all the subscribers are connected at SHBs that are only located at the leaf nodes of the tree. The subscriptions are propagated from the leaf nodes toward one single direction – the root of the tree.

This is our conceptual model. In reality, our system allows publishers and subscribers to connect at any leaf broker. As a result, any edge (non-internal) broker can be the destination of subscription propagation (PHB) and any edge broker can be a SHB. Thus, subscriptions originated at a SHB are flooded toward each edge broker, a.k.a. a potential PHB. In addition, the aggregation is performed for every outgoing edge. For example, if a broker $b$ resides in a routing tree node $n$ that has three edges $e_1$, $e_2$ and $e_3$, subscription information entering $b$ from edge $e_1$ of $b$'s residing tree node will be aggregated with subscription information from edges $e_2$ and $e_3$ and the results of the aggregation will be sent through edges $e_3$ and $e_2$, respectively. The same is true for subscriptions from edges $e_2$ and $e_3$.

Another overhead incurred in the support of multiple PHBs is at the intermediate routing brokers. Unlike in a routing tree with a single PHB, an intermediate routing broker needs to maintain subscription information for every edge of its residing tree node instead of just a downstream defined on the tree rooted at the single PHB. The subscription state maintained at an intermediate routing broker thus includes aggregated subscriptions for every part of the routing network. In addition, the $V^{\mathrm{b}}$ vector of the intermediate routing broker contains elements for

every SHB in the network. Nevertheless, for any data message, there is a single source - its originating PHB. Thus, there is no confusion at an intermediate routing broker as to where the downstream brokers are and what are the SHBs in the downstream network. As a result, there is no confusion as to how the slicing of $V^{\mathrm{b}}$ and $V^{\mathrm{m}}$ vectors for the downstream SHBs should be done.

We envision that the aforementioned overhead typically would not be a performance problem as the subscription changes are not frequent compared with the rate at which data messages are published and matching algorithms can be very efficient even in the presence of large number of subscriptions. There are additional techniques such as *publication advertisement* that can further reduce this overhead. *Publication advertisement* is a mechanism that allows publishers to declare the kind of data message they are going to publish. Using advertisements, subscription information can be directed only to where the relevant advertisements are declared. It can be viewed as a subscription to subscription information. We do not discuss advertisement in further detail in this work.

### 6.1.3 Sufficiency Test Result Cache

The cost of performing sufficiency test and the cost of transferring a data message carrying a $V^{\mathrm{m}}$ vector is a potential overhead in our system compared to a system that does not support our propagation protocol. As previously discussed, the sufficiency test is the test that compares the broker's $V^{\mathrm{b}}$ vector with the $V^{\mathrm{m}}$ vector the data message carries. The $V^{\mathrm{b}}$ and $V^{\mathrm{m}}$ vectors have one element for each SHB in the system. The test is performed by doing an element-by-element comparison of the vectors. As a result, the number of SHBs/leaf brokers affects

system performance and scalability.

The communication cost is also affected by the number of SHBs in the system as the length of the $V^{\mathrm{m}}$ vector carried by a data message is proportional to the number of SHBs. We address these two issues using a $V^{\mathrm{m}}$ vector digest mechanism and a cache of previously computed sufficiency test results. The performance cost of this mechanism is measured both analytically and experimentally in Section 6.2.3 and Section 6.3.7.

#### $V^{\mathrm{m}}$ Vector Digest

Instead of using full $V^{\mathrm{m}}$ vectors in data messages, we use fixed-length vector digests. A vector digest is a *unique*, short representation of a $V^{\mathrm{m}}$ vector. We use a digest that is 64-bit.

When a new $V^{\mathrm{m}}$ vector is first assigned to a message at a PHB, the PHB assigns a digest to $V^{\mathrm{m}}$ by taking a snapshot of a monotonically increasing value. We use the machines's system clock value as the digest value. As the PHB assigns monotonically non-decreasing $V^{\mathrm{m}}$ vectors and digests are also monotonic, the system satisfies the following monotonicity - if $V_m < V'_m$ then $d < d'$, with regard to the same PHB. In the above formula, $d$ and $d'$ are the digests for $V^{\mathrm{m}}$ and $V'_m$ respectively.

Unlike the $V^{\mathrm{m}}$ vectors, a PHB does not store the highest digest assigned in persistent storage. As a result, the PHB may assign two different digests to the current highest $V^{\mathrm{m}}$ vector if it crashes and recovers. Thus $d' > d \Rightarrow V'_m \geq V_m$. This means that if we receive a message with a digest $d'$ that is greater than an earlier digest $d$, the $V'_m$ vector represented by $d'$ is at least as large as $V^{\mathrm{m}}$ represented by $d$. As a result, one can safely reuse the sufficiency test results

based on the values of $V^{\mathrm{m}}$ and $d$. As described later in this section, this property is used in the sufficiency test result caching mechanism.

We describe below the mechanism that uses *fixed-length* $V^{\mathrm{m}}$ digests instead of the original $V^{\mathrm{m}}$ vectors in data messages. The sufficiency test a broker performs is still based on the $V^{\mathrm{m}}$ vectors of the data messages. However, our mechanism allows the digest to convey the right $V^{\mathrm{m}}$ vector to a downstream routing broker. Hence, sufficient test can still be performed as usual.

In this $V^{\mathrm{m}}$ vector digest mechanism, a routing broker maintains information on whether any data message with a new $V^{\mathrm{m}}$ has been sent through each physical link that connects this broker to a broker in its downstream routing tree node. Such information is purged when the physical link fails and recovers. For each physical link, the first data message with the new $V^{\mathrm{m}}$ vector going down the link carries the original $V^{\mathrm{m}}$ AND its digest. This conveys to the downstream broker the mapping between a digest and its corresponding $V^{\mathrm{m}}$ vector. Later messages only need to carry the digest until the $V^{\mathrm{m}}$ vector changes or the link fails and recovers. Hence, most messages have a fixed length payload incurred by subscription propagation. This addresses the scalability of communication overhead with regard to the large number of SHBs in the system.

In the Gryphon system, broker to broker links are implemented as TCP connections and thus are FIFO (First-In-First-Out). As a result, the first data message that carries a new $V^{\mathrm{m}}$ vector and its digest is guaranteed to arrive before any data message with only the digest arrives. This property ensures that a broker can always know the $V^{\mathrm{m}}$ vector of a data message from its digest. However, as we will discuss below, the correctness of the system does not depend on this. Indeed,

our mechanism may improve performance even if the links are only approximately FIFO. That is, the links will deliver most even if not all messages in order.

When this FIFO property is violated (e.g. due to broker or link failures), some data messages with only a digest may arrive before the message carrying the mapping. In such cases, the sufficiency test can not be performed. These out-of-order data messages can be conservatively sent to all downstream brokers, just as if the sufficiency tests had failed. When the links are nearly FIFO, most messages with a digest will arrive after the message carrying both the original $V^{\mathrm{m}}$ vector and the digest, and thus saving the cost of communication while not imposing a significant impact on content-based routing.

The FIFO link property may also be violated at a SHB. Similarly, the SHB may receive a data message with only a digest before it receives the message carrying the mapping. In this case, the SHB cannot pick this message as a delivery starting point for a new subscription. Such a new subscription typically has a virtual start time that is greater than the virtual start times of all existing subscriptions whose delivery starting points are before this current data message in the message stream. If a new subscription has a virtual start time that is less than or equals to some of the existing subscriptions, the current message with only a digest can still serve as the delivery starting point. On the other hand, message delivery for existing subscriptions are never affected by a data message with only a digest due to that data messages in the same stream are assigned monotonically non-descreasing $V^{\mathrm{m}}$ vectors.

In a system that does not use FIFO links, negative acknowledgements can be used to retrieve the mapping between a digest and the $V^{\mathrm{m}}$ vector it represents.

If a broker receives a data message with only a digest whose corresponding $V^{\mathrm{m}}$ is not known, the broker can delay the processing of the data message until it retrieves the mapping of the original $V^{\mathrm{m}}$ vector and the digest through negative acknowledgements. This eliminates some of the conservative flooding, but at a cost of incurring processing delays. As this is not the case with the Gryphon system, this mechanism is not needed in our work.

#### Sufficiency Test Cache

As mentioned earlier in this section, another cost incurred by our protocol is the computation overhead of the sufficiency test. To deal with this issue, we use a cache of previously computed sufficiency test results.

In the vast majority of application scenarios, subscription change rate is much slower than the data message rate in the system. Our capability to batch subscription changes further exaggerates this trend. The result of this factor is the same $V^{\mathrm{m}}$ vector or digest being assigned to a large number of data messages.

This assignment of same $V^{\mathrm{m}}$ vector or digest to large numbers of data messages enables us to do further optimization. The broker in our system can reuse the sufficiency test result computed for a data message for all later data messages with the same $V^{\mathrm{m}}$ or digest. As a result, this sufficiency test needs only be performed on the first data message carrying a $V^{\mathrm{m}}$ vector until either the message's $V^{\mathrm{m}}$ vector changes or the broker's $V^{\mathrm{b}}$ vector changes.

In our system, each broker maintains a cache of the sufficiency test results. This cache is indexed by the digest and the assigning PHB. Table 6.1 shows the structure and content of a sufficiency test cache at a routing broker.

As we can always be conservative and flood a message until it reaches a SHB,

Table 6.1: Sufficiency Test Cache

| PHB | Digest | $V^{\mathrm{m}}$ vector | Sufficient |
|---|---|---|---|
| $PB_1$ | 74848251409924096 | $\{SB_1$:(1142093150284,110),...$\}$ | false |
| $PB_2$ | 74848250754236416 | $\{SB_1$:(1142093150284,109),...$\}$ | true |
| ... | ... | ... | ... |

the sufficiency test cache only needs to maintain one entry for each PHB. That is, the highest digest it has seen from the PHB. This is because usually the highest digest is the current one in use. In addition, since the digest is monotonic with the original $V^{\mathrm{m}}$ vector, any message originated from the same broker with a digest no greater than the cache entry can reuse the cache result. Otherwise, the broker conservatively computes $false$ as the sufficiency test result for the message. The broker updates its cache entry for a PHB when it receives a message with a higher $V^{\mathrm{m}}$ vector and digest from the PHB. When the broker advances its $V^{\mathrm{b}}$ vector due to receiving new subscription changes, the cache entry for PHBs with a $false$ sufficiency results is invalidated and recomputed lazily when a data message from the PHB arrives.

### 6.1.4 Improving System Concurrency Level

In order to achieve high throughput, the Gryphon system uses a pool of threads to process incoming messages and quickly switch them through to the next hop. This parallelism greatly improves system throughput. In addition, processing delay is also reduced because a message can be processed whenever there is a thread available without having to wait for the completion of processing of all messages

arriving in front of it. Like in many systems, this level of concurrency is subject to synchronization for accessing shared variabls. Achieving highly concurrent access to shared variables is thus critical for the system performance.

In our protocol, a PHB needs to assign $V^{\mathrm{m}}$ vectors to data messages. The value of current $V^{\mathrm{m}}$ vector in use is stored in a shared variable. Multiple threads need to access this shared variable when they are processing incoming messages from publishers. As we have discussed in Section 4.3.1, the PHB's $V_m$ vector takes the maximum of the $V^{\mathrm{m}}$ and its $V^{\mathrm{b}}$ vector elements. As a result, when the PHB receives subscription propagation messages that advance its $V^{\mathrm{b}}$ vector, this shared variable may need to be updated. Hence, this shared variable is a potential hotspot for lock contention.

We analyzed the access pattern of the shared variable, and identified that most accesses are read-only with occasional writes due to the low frequency of subscription changes compared with the data message rate in the system. Hence, we developed a re-entrant multi-reader-single-writer lock that increases the level of concurrency of read operations. We have experienced a performance improvement through the use of this mechanism.

## 6.2 Analytical Model and Results of System Scalability

In this section, we analyze the asymptotic cost of the sufficiency test computation and data message communication due to the use of virtual time vectors $V^{\mathrm{m}}$ and $V^{\mathrm{b}}$. We perform this analysis because this computation and communication overhead is the single most significant overhead introduced by our protocol. In Section 6.3, we will provide experimental results on the overhead of the sufficiency test as well.

### 6.2.1 Analytical Model

As described in our protocol, the data message routing is computed by performing matching on the message content and performing sufficiency test on the message's $V^{\mathrm{m}}$ vector and the broker's $V^{\mathrm{b}}$ vector. The data message may carry a $V^{\mathrm{m}}$ vector digest $d$ as well as the original $V^{\mathrm{m}}$ vector if the message is the first message assigned the $V^{\mathrm{m}}$ vector going down a FIFO link. As the first data messages carrying new $V^{\mathrm{m}}$ vectors arriving at a broker contain an $V^{\mathrm{m}}$ vector element for each downstream SHB, the communication overhead on these first messages are affected by the number of SHBs in the system. Later messages only carrying digests are not affected.

As described in Section 6.1.3, we have implemented a caching mechanism that re-uses the previously computed sufficiency test results. For any given PHB and a $V^{\mathrm{m}}$ vector, the cached result only needs to be computed the first time a data message carrying a new $V^{\mathrm{m}}$ vector arrives or the broker's $V^{\mathrm{b}}$ vector changes. The $V^{\mathrm{m}}$ and $V^{\mathrm{b}}$ vectors have sizes that are proportional to the number of SHBs in the system. As a result, the computation overhead of the sufficiency test on the *first* data message with a $V^{\mathrm{m}}$ vector is affected by the number of SHBs. We analyze the scalability of our protocol with regard to the number of SHBs in the system under our mechanism using a sufficiency test result cache.

We describe in the rest of this section a number of varibles that capture the system's characteristic that are relevant to the scenario described above. These variables affect both communication overhead and sufficiency test computation overhead. We delay the variables that affect only communication overhead or sufficiency test computation overhead to their own sections, respectively.

For simplicity, we assume a ratio $r$ between data message publishing rate and subscription change rate (PHB virtual time vector change rate) in the system. We only examine the case where $r$ is a large value because:

- The system has the capability to batch subscription changes and hence can control the subscription change rate to be a small value;
- Efficiency is an issue under heavy-load situation. In light load situation, the system has extra resource to spare. Hence, the data message publishing rate is a big value in our model.

We assume a selectivity ratio $p$ where $(0 < p \leq 1)$ for each level of tree nodes. That is, at any tree node, a message passes the content filter from any of its child node with uniform probability $p$. We further assume that each tree node has $w$ child nodes and each non-leaf non-root node in the routing tree of the broker network has two redundant virtual brokers.

We define a node's level as the number of tree edges from the PHB to the node. Thus, the PHB resides in a level 0 node.

We further define variable $SHB$ as the number of SHBs in the whole broker network. For convenience, we re-use the same notation of $SHB$ in our work.

### 6.2.2 Communication Overhead Scalability

We describe in this section the asymptotic overhead of transmitting data messages in the system. As described in the previous section, this overhead includes the overhead of transmitting the first data message carrying a new $V^{\mathrm{m}}$ vector and later data messages carrying only a *fixed-length* digest. The overall communication overhead is the weighted average of these two overheads.

For a broker at a tree node at level $l$ ($l > 0$), it receives one data message with a new $V^{\mathrm{m}}$ filter every $p^l r/2$ messages. The divisor of 2 is due to the existence of 2 peer brokers in the same node, as a result, they share the data mesage traffic. This one message with the $V^{\mathrm{m}}$ vector requires an extra communication cost due to the $V^{\mathrm{m}}$ vector. We denote this cost as $c_1 \times SHB/(w^l)$, where $c_1$ is a constant of transmitting one element of the $V^{\mathrm{m}}$ vector, and $SHB/(w^l)$ is the number of SHBs in the downstream of this broker in the system. The rest $p^l r/2 - 1$ messages requires a constant overhead of $c_2$ on communication. Note that as we assume $r$ is a big value, $p^l r/2 - 1 > 0$. Hence the average communication overhead is:

$$(c_1 \times SHB/w^l + c_2(p^l r/2 - 1))/(p^l r/2) \tag{6.2.1}$$

that is

$$c_2 - 2c_2/p^l r + SHB \times 2c_1/((wp)^l r) \tag{6.2.2}$$

The upper bound of the cost is

$$c_2 + SHB \times 2c_1/((wp)^l r) \tag{6.2.3}$$

When $r$ is a big value and the selectivity ratio $p \geq 1/w$, the dominant factor is the constant $c_2$. This result means that the communication overhead grows VERY slowly with the number of SHBs in the system. In practice, this is typically negligible.

### 6.2.3 Sufficiency Test Computation Scalability

Similarly, we can compute the cost of computing the sufficiency test. As each subscription change may trigger a new $V^{\mathrm{m}}$ vector assignment and a change on the

broker's $V^{\mathrm{b}}$ vector, the cache entry may be updated at most twice due to each of these factors. This cache entry update requires a re-computation of the sufficiency test and the cost is proportional to the number of SHBs in the downstream. Assume the cost of such a sufficiency test computation is $c_3 \times SHB/w^l$, where $c_3$ is a constant, and the cost of each cache entry look up is a constant $c_4$, the average cost is

$$(2c_3 \times SHB/w^l + c_4(p^l r/2 - 1))/(p^l r/2) \tag{6.2.4}$$

where the first item is the cost of the two sufficiency test computation and cache update and the second item is the cost of constant cache lookup for the rest of $p^l r/2 - 1$ messages that the broker routes after the cache entry update.

An upper bound of the above equation is

$$c_5 + SHB \times (4c_3)/((wp)^l r) \tag{6.2.5}$$

when $r$ is a big value and $p \geq 1/w$, the dominant factor is the constant $c_5$. This result means that the overhead of computing sufficiency test grows VERY slowly with the number of SHBs in the system. In practice, this is typically negligible.

### 6.2.4 Summary

We have shown in this section the asympototic results of the data message communication and sufficiency test computation scalability with regard to the number of SHBs. As we can see, the effect of the number of SHBs on the computation and communication overhead is offset by a very small factor. In practice, we believe this effect will be dominated by the constant factor imposed by the large number of messages that carry a $V^{\mathrm{m}}$ digest.

We show in the next section the experimental results of our work.

## 6.3 Experimental Results

The testbed for our experiments is a set of IBM RS6000 F80 machines with six 500MHz processors and 3G RAM. Each machine has dual network interfaces and is connected through a 100Mbps Ethernet network and a gigabit switch to other machines.

We focus primarily on metrics that are impacted by the specifics of our solution. These include the processing overhead incurred by our protocol as described above and the high availability feature of our protocol even in the presence of all but one broker crash in a redundant routing tree node.

### 6.3.1 The Flooding Scheme

In a system without subscription propagation, the routing brokers flood the messages to a broker in every downstream routing tree node. Filtering is only performed at the SHBs.

The flooding scheme, although enabless simple content-based routing in the middle of the network, wastes considerable network bandwidth when the subscriptions are regional and selective.

In our experiments, we have used the flooding scheme as a baseline to show the efficiency of subscription propagation in various cases. Some of these cases require regional selective delivery of messages. Some do not require selective delivery and thus are adverse scenarios to the subscription propagation scheme. We present the latter as a worst case scenario to show that subscription propagation can be done efficiently without much added overhead.

To understand the trade-offs between flooding and content-based routing with

subscription propagation, we present below a micro benchmark that studies the overhead of content-based matching.

### 6.3.2 Micro Benchmark: Content Matching Overhead

Our purpose in studying the overhead of content-based matching is to provide some quantitative guidance to help estimate how much cost message filtering will save and/or incur. This estimation, however, should be based on the usage pattern of each individual scenario. As subscription propagation provides an opportunity to trade off computation overhead on content-based matching with communication overhead saved by filtering messages, we present some results from the literature on the communication overhead in wide-area networks.

#### Network Latency

Subscription propagation is an optimization that trades computation cost of performing content matching with communication cost saved from filtering messages. The communication cost typically includes a latency of ten to several hundred milliseconds. For example, US cross country links typically have a minimum round trip time of 70 milliseconds [28]. About 40% of this overhead is due to network router processing [29]. That is, the routers have added 28 milliseconds processing delay.

The above measurement was reported annually by the International Committee for Future Accelerators (ICFA), Standing Committee on Inter-Regional Connectivity (SCIC). The measurement is performed using a tool that uses ICMP Ping packets. These Ping packets incur very light overhead. A message broker typically use TCP or UDP packets. The processing delay at each broker and net-

work routers will be even higher due to the added overhead of TCP protocol at brokers and IP routing at network routers.

**Benchmark Results**

We use the content matching engine in the Gryphon system to study the overhead of content-based routing. The Gryphon matching engine identifies a special attribute that is most frequently used in subscription filters and messages. This is a valid assumption that has been tested in many usage scenarios. The Gryphon matching engine caches the matching results on the special attribute. We study the matching overhead in both cases where subscription filters only contain this special attribute and where subscription filters contain other attributes as well. In both studies, the match engine was pre-loaded with $10^5$ random equality subscriptions on the special attribute. The data messages carry a random value selected from the range of $10^4$ values for the special attribute.

Figure 6.1 shows the overhead of performing content matching on the special attribute the system optimizes. In each test, the messaging engine performs 1 to $10^8$ matching operations. The reported results show the average costs of a single matching operation in each of these tests.

As shown in the figure, the overhead drops sharply when the number of matching operations increases. That is, each of the first several matching operations are likely to incur high overhead. As the number of matching operations the system performs exceeds a threshold at about $10^6$, the overhead of each matching operation becomes a constant of about 35 microseconds. This is due to that the Gryphon matching engine caches matching results on each distinct value of the special attribute. As the system performs content matching, the cache is pop-

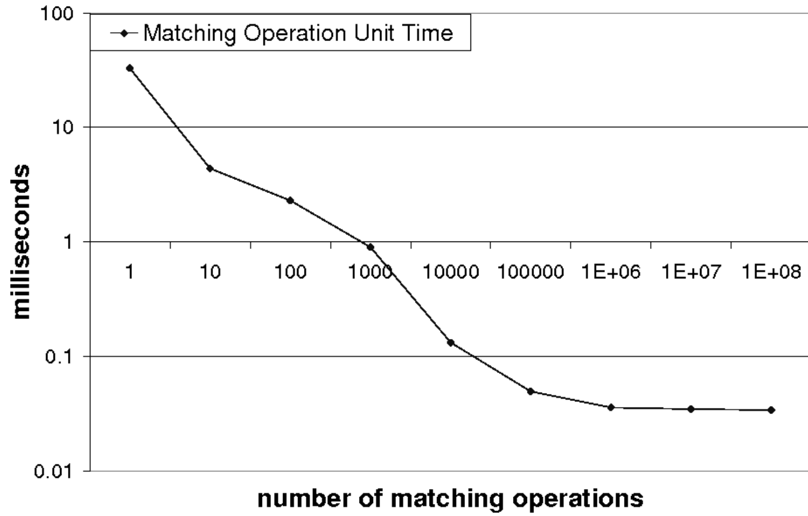



Figure 6.1: Matching Overhead on Subscriptions on Cached Attributes

ulated with each value of the special attribute. When the number of matching operations exceeds a threshold, the cache is almost fully populated with previous results and matching becomes an efficient cache retrieval operation that is of constant cost.

Figure 6.2 shows the overhead of performing content matching on other attributes which the system does not treat specially. As shown in the system, the overhead is around 50 microseconds, which is still inexpensive compared to the per-hop network latencies.

### 6.3.3 System Load Comparison in Selective Subscription Tests

This test compares the system overhead of using subscription propagation with that of using flooding when client subscriptions are selective. The workload is motivated by sensor networks where there are many publishers collecting and publishing various kinds of data and relatively few subscribers that selectively

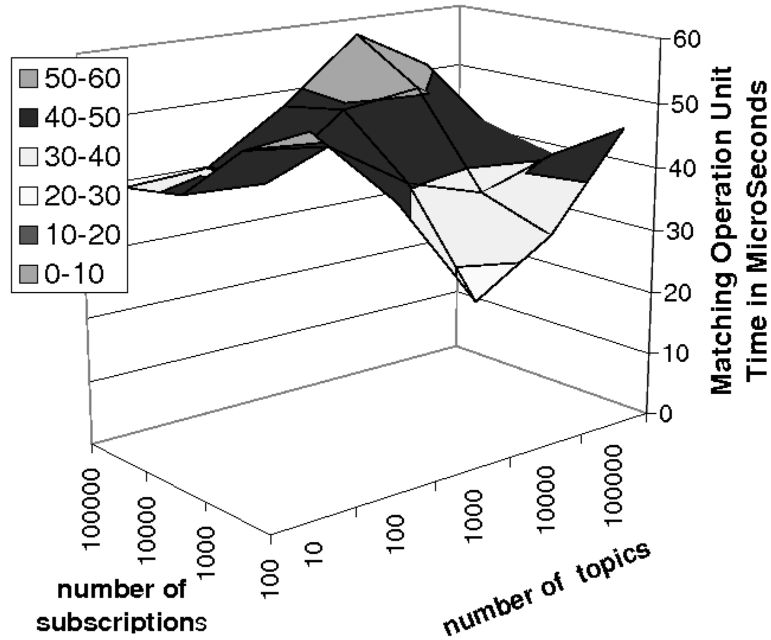



Figure 6.2: Matching Overhead on Subscriptions on Non-Cached Attributes

subscribe to data of interest.

The setup of this test is shown in Figure 6.3. In the network, there are 4 *physical publisher hosting brokers* $pb_1$, $pb_2$, $pb_3$, and $pb_4$ and 1 *physical subscriber hosting broker sb*. Each of these brokers implements a virtual broker in its own tree node. In addition, these brokers also reside in an intermediate tree node $N_5$. Four redundant routing trees can be defined by taking each of $pb_1$, $pb_2$, $pb_3$, and $pb_4$ as root node.

This network models a high-fan-in-low-fan-out network that is typical in some application scenarios like the sensor networks. In this kind of applications, a large amount of data is generated by large numbers of data sources. However, the subscribers are only interested in some of the data. In addition, there may not be many subscribers.

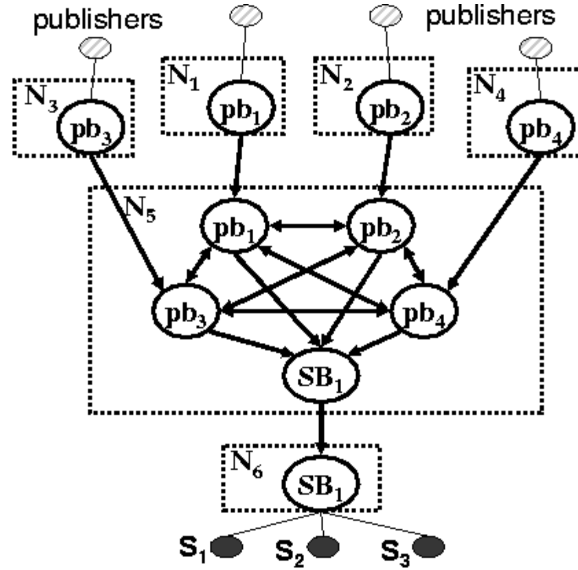



Figure 6.3: Network Topology for System Load (CPU) Comparisons

We fix the number of subscriptions at 2000 and vary the number of publishers from 2000 to $1.2 \times 10^4$. The message rate per publisher is fixed at 2 messages/second. Thus, the total incoming message rate ranges from 4000 to $2.4 \times 10^4$ messages/second. Each subscription is distinct and selects exactly the messages published by 1 publisher. Hence, the total receiving message rate of subscribers is 4000 messages/second throughout the test. The publishers are evenly distributed among $pb_1$, $pb_2$, $pb_3$, and $pb_4$.

Figure 6.4 and Figure 6.5 show the CPU utilization at the PHBs $pb_1$, $pb_2$, $pb_3$, and $pb_4$ and the SHB $sb$ under both the flooding scheme and the subscription propagation scheme. The figures are plotted using mathematical means of more than 400 data samples collected over a long running period. The error bars show the standard deviation of the collected data. We examine Figure 6.4 and Figure 6.5 in detail below.

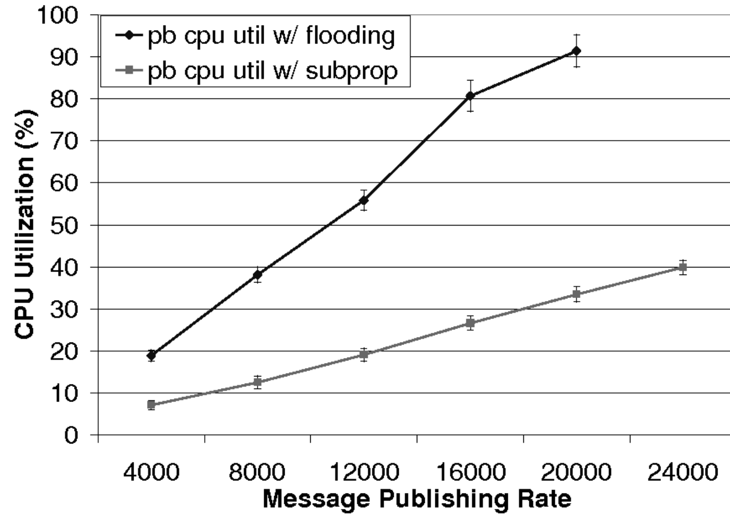



Figure 6.4: PHB System Load (CPU) Comparison

In the case of 2000 publishers and 4000 messages/second incoming message rate, all published messages are subscribed by the 2000 subscribers. This case is NOT favorable to subscription propagation. However, we do not observe a large increase of CPU utilization at *sb* in the subscription propagation scheme. Indeed, the experiment shows that the subscription propagation scheme performed slightly better. This is within the error tolerances.

When the number of publishers (and hence incoming message rate) increases, the CPU utilization at the SHB *sb* stays constant in the subscription propagation scheme because the numbers of messages subscribed to and hence need to be routed are constant. However, the CPU utilization at *sb* in the flooding scheme increases linearly even though the number of *useful* messages does not change.

As shown in Figure 6.4, in both schemes, CPU utilization at $pb_1$-$pb_4$ increases linearly with the number of publishers. However, the flooding scheme shows a much steeper slope because each of the PHBs $pb_1$, $pb_2$, $pb_3$, and $pb_4$ not only ac-

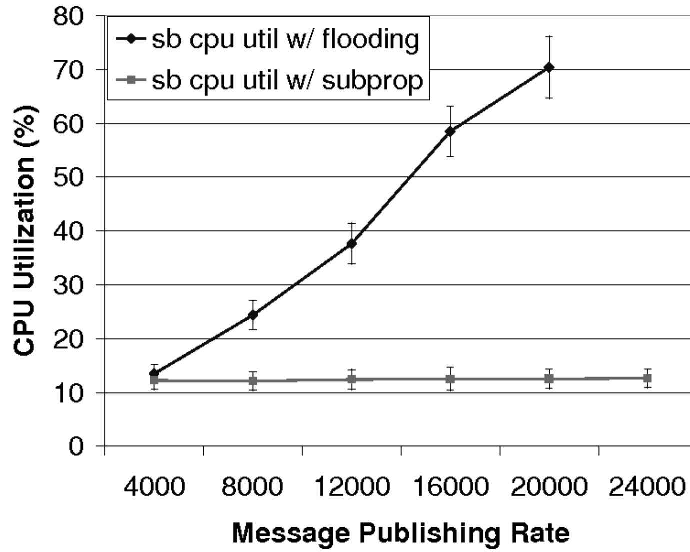



Figure 6.5: SHB System Load (CPU) Comparison

cepts messages from publishers but also sends these messages to other PHBs and receives all messages published through other PHBs. In contrast, in the subscription propagation scheme, each PHB only needs to route part of the messages to the SHB *sb*. Although the brokers need to perform content matching in the subscription propagation scheme, as our micro benchmark in Section 6.3.2 shows, we expect this overhead to be quite small compared with the communication cost. As a result, in the flooding scheme, CPU utilization at PHB $pb_1$-$pb_4$ reaches $> 90\%$ with only $10^3$ publishers compared to 31% with $1.2 \times 10^4$ publishers with subscription propagation. we were unable to scale to $1.2 \times 10^4$ publishers in this configuration with the flooding scheme.

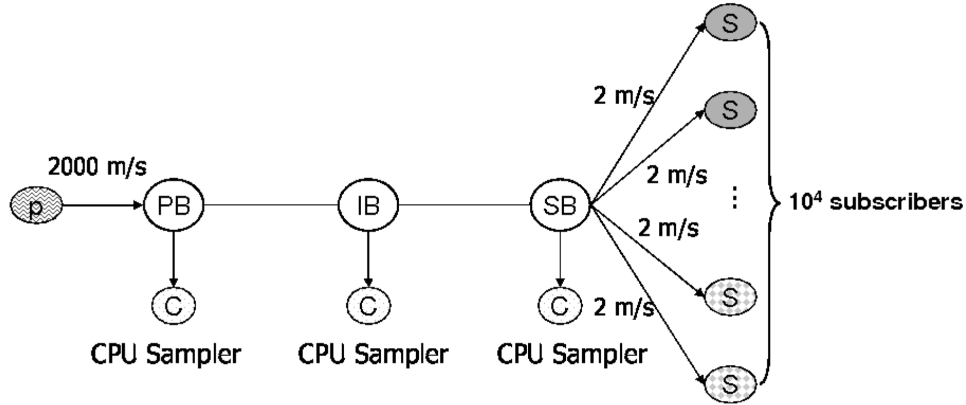



Figure 6.6: Topology Network for CPU Measurements of Access Control

### 6.3.4 CPU Overhead with Access Control Enforcement

This test compares the system performance of enforcing access control for subscribers when their subscribing rights do not change. We use a setup in which a publisher hosting broker *pb* is connected to an intermediate broker *ib*, which in turn connects to a subscriber hosting broker *sb*. The steady access control policy in this test permits the subscribers to receive all messages they subscribe to. Therefore, the broker network routes and delivers the same amount of messages as in the system when there is no support for dynamic access control. We examine the CPU overhead at each broker and compare it with our baseline results, i.e., results obtained in the Gryphon system with no dynamic access control support. This comparison represents the overhead incurred at brokers playing different roles in the protocol - access control version setter (*pb*), in-network access control enforcer(*pb*&*ib*) and end-point ultimate access control enforcer (*sb*).

Messages are injected into the system through *pb* at a rate of 2000 messages per second. We evaluate both the cases when the Gryphon special reliable delivery

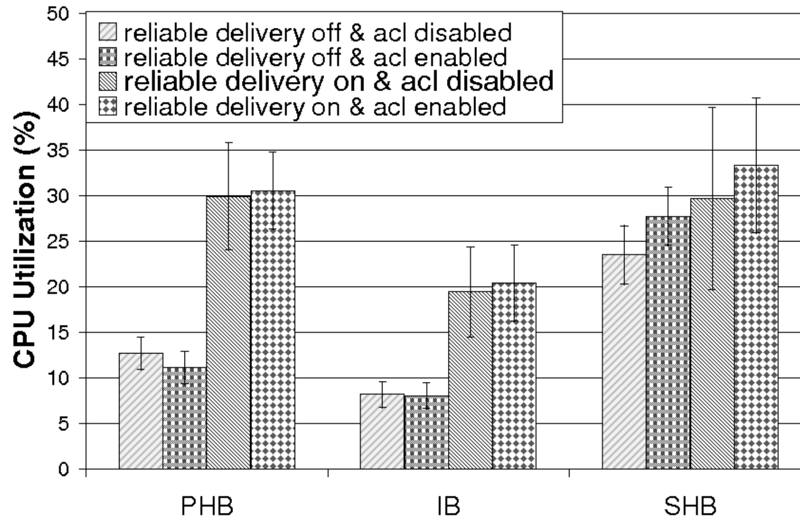



Figure 6.7: CPU Utilization at Brokers

protocol is turned on or bypassed. To eliminate the impact of different file systems used for PHB persistent message streams, we perform Gryphon message logging but do not sync to the disk. The output message rate at the SHB is 20000 messages per second to 10000 subscribers. Figure 6.7 shows the CPU utilization at each of the brokers. This test shows that the CPU overhead increase at *pb* and *ib* are very small as the overhead is mostly for assigning and/or comparing control versions. The CPU increase at *sb* is higher due to the final access control enforcement that is performed against every matching subscriber for every message.

### 6.3.5 Latency Measurements

#### Local and Remote Delivery Start Latency

In the flooding scheme, since all messages are flooded to the SHBs and only filtered at the SHBs, there was no delay of starting message delivery to a new subscrip-

tion other than the time lapse between two consecutively published messages that match the subscription. If two consecutive messages matching the new subscription are published with an interval of $t$, the delay of starting message delivery to this subscription is on average $t/2$.

In our system, if the new subscription is covered by an existing subscription, the delay of starting message delivery to the new subscription is the same as in the flooding scheme. Since the new subscription never goes out of the SHB, we call the delivery start latency of this kind of new subscriptions *local delivery start lantency*.

On the other hand, if the new subscription is not covered by any existing subscription, the new subscription will be assigned a virtual start time whose value has never been propagated out of the SHB. The SHB must advance its virtual clock and propagate the subscription along with its virtual start time. The PHB will assign data messages $V^{\mathrm{m}}$ vectors whose element for the SHB is greater than the virtual start time of the new subscription. The SHB can only start delivery for the new subscription when it sees the first such data messages. Hence, the delivery start latency for this kind of new subscriptions requires at least a round trip to and from the PHB. As a result, we call this *remote delivery start latency*.

**Measurement**

We examine three types of latency metrics: local delivery start latency, remote delivery start latency and message delivery latency. Specifically, we examine their trends with regard to the number of hops between their SHBs and a PHB.

We measure the message delivery latency as the time taken for the system to

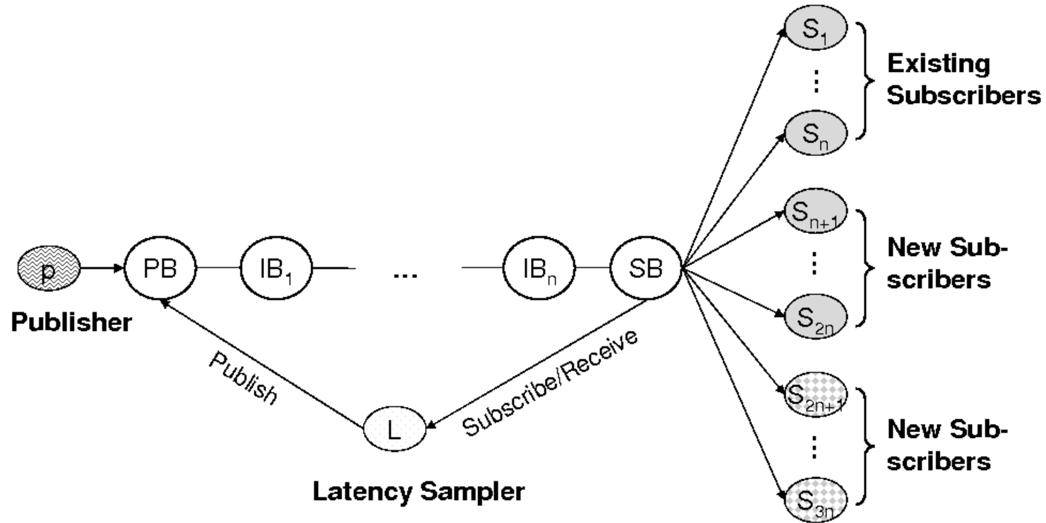



Figure 6.8: A Linear Topology Network for Latency Measurements

deliver a message from a publisher to an existing subscription. The message delivery start latencies are measured as the time elapsed from when a new subscription is submitted to the time when the first message is delivered to the subscription.

In order to examine the trends of the latency metrics with regard to the number of hops between their SHBs and a PHB, we use a linear topology network as shown in Figure 6.8. This network consists of a PHB *pb* and a SHB *sb* in its own routing tree node. Intermediate routing broker $IB_1$ to $IB_n$ connect *pb* and *sb* through 1 to $n$ hops. Each of these intermediate routing brokers is also in its own routing tree node.

The message delivery latency is measured by a *latency sampler* that publishes messages through *pb* and subscribes to its own messages through *sb*. The latency sampler records in each message the time at which the message is published and checks the difference at the time when it receives the message.

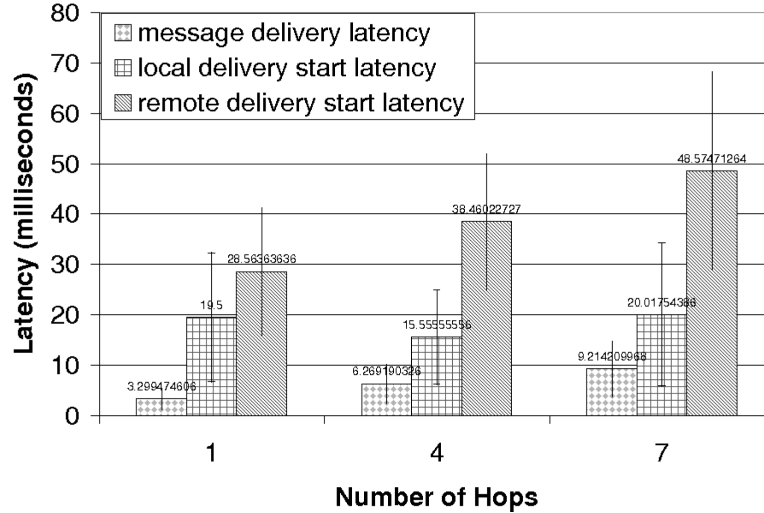



Figure 6.9: Delivery Start Latencies and Message Delivery Latency

The delivery start latency for a new subscription is measured at the subscriber by taking the difference between the time of subscription and the time of the first message delivery. As shown in Figure 6.8, there are three groups of subscribers: a group of existing subscribers $s_1$ to $s_n$, a group of new subscribers $s_{n+1}$ to $s_{2n}$ that subscribe to the same set of messages as the existing subscribers group, and a group of new subscribers $s_{2n+1}$ to $s_{3n}$ that subscribe to totally distinct set of messages from the other two groups. As a result, the first group of *new* subscribers measure the *local delivery start latency* and the second group of *new* subscribers measure the *remote delivery start latency.*

Remote start delivery latency is the sum of the following items:

1. time taken to send subscription to the SHB *sb*;
2. processing time at *sb*;
3. processing time at the intermediate routing brokers and the PHB *pb*;

4. network delays(bi-directional) at each hop;
5. expected interval till next message published that matches the subscription. If messages that match the subscription are published at a steady rate every $t$ milliseconds (ms), this time is $t/2$ on average;
6. time taken to send the message from $sb$ to the subscriber.

We expect the most significant latency impact to come from item 2 and 3. The impact of item 1 is reduced by deploying the network on a high-speed LAN with low latencies. This also reduces the impact of items 4 and 6. To reduce the impact of item 5, we use a high message publishing rate at which each subscription would receive messages at the rate of 200 messages/second. The delay caused by item 5 is thus $1000 \times (1/200)/2 = 2.5$ milliseconds. The total incoming message rate is 3200 messages/second.

Similarly, local delivery start latency is the sum of the following items:

1. time taken to send subscription to the SHB $sb$;
2. processing time at $sb$;
3. expected interval till next message published that matches the subscription.
4. time taken to send the message from $sb$ to the subscriber.

**Results**

We show our measurement results in Figure 6.9.

Figure 6.9 shows the various latencies with regard to the number of hops in the network. The latencies shown are taken as the mean of approximately $600 - 1000$

data samples. The error bars show the standard deviation of the data samples. Since the tests are long running, several Java garbage collections have happened and resulted in a few samples that are about 40 to a couple of hundred times higher than the median. We took out these outliers in our calculation of the mean and standard deviation. The percentage of outliers are in the range of $1-2$ percent.

Our test results (Figure 6.9) show that message delivery latency increases linearly from 3.29 to 6.27 to 9.21 ms with the number of hops. Local delivery start latency for covered subscriptions - stays roughly constant at around $19ms$. The local delivery start latency also represents the subscription propagation overhead in the flooding scheme. Remote delivery start latency increases linearly from $28.56ms$ to $38.46ms$ to $48.57ms$. The differences of the remote and local delivery start latency shows the network latency and intermediate routing broker and the PHB processing overhead. This is the overhead incurred by the subscription propagation protocol. This overhead increases from $9.56ms$ to $23.46ms$ to $29.57ms$ for 1, 4 and 7 hops. This shows the linearly scalability of remote delivery start latencies with the number of hops from the SHB to the PHB. The overhead of subscription processing delay per network hop is also small, at about 3.3 millisecond per increased hop.

### 6.3.6 Latency Measurement with Dynamic Access Control

We examine three latency metrics: 1) The latency of message delivery during steady state when there are no access control changes. 2) The latency of starting delivering messages to a newly connected subscriber when there are already

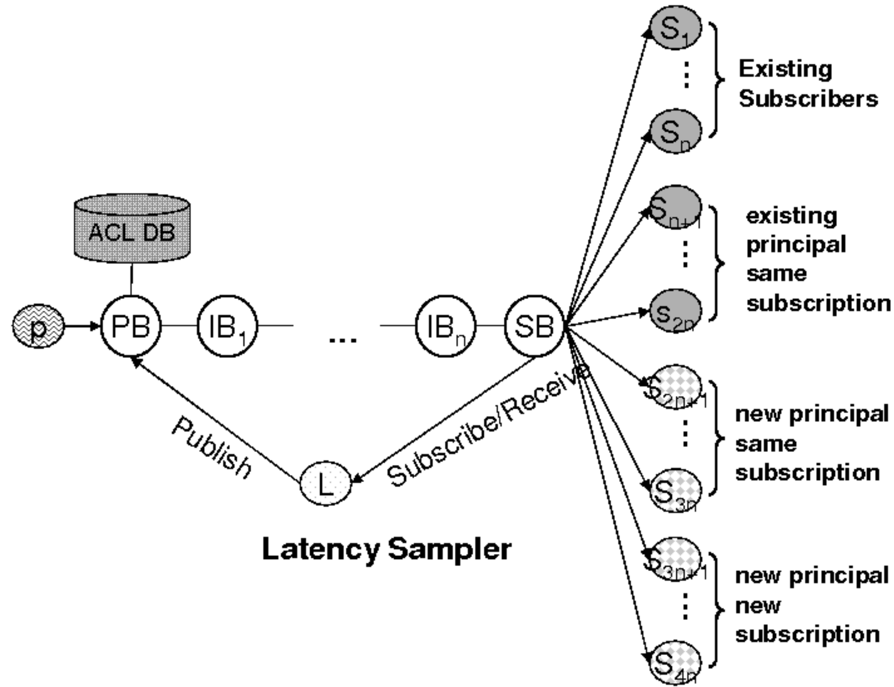



Figure 6.10: Topology Network for Latency Metrics with Dynamic Access Control

connected subscribers at the SHB on behalf of the same principal $p_1$ and the new subscriber uses the *same subscription* as an existing subscriber. We call this the *local* delivery start latency. 3) The latency of starting delivering messages to a newly connected subscriber on behalf of a new principal $p_2$, however the new subscriber uses the same subscription as some of the existing subscribers at its SHB. As a result, the SHB does not have access control rules for the new subscriber cached and must retrieve initial access control rules. However, the SHB can process the new subscription *locally* without having to propagate the subscription outside to other brokers. We refer to this metric as the *new principal local* delivery start latency. 4) The latency of starting delivering messages for a newly connected subscriber on behalf of a new principal $p_3$ AND the new subscriber uses

a new subscription that is not used by any other subscribers at the SHB. We call this the *new principal remote* delivery start latency as the SHB needs to propagate the subscription remotely to the PHB in addition to retrieving the initial access control rules.

We use a linear topology consisting of a PHB *pb* and a SHB *sb* that are connected through one or more hops of intermediate brokers $ib_1, \ldots, ib_n$. Unlike in the other experiments, the brokers are connected through a gigabit switch, the same network that the subscribers and publishers are connected to the brokers. This is due to a network reconfiguration and the unavailability of a reasonable network performance of the 100Mbps network. However, the scalability aspect is not affected.

In this experiment, we colocate the ACL DB with *pb*. Thus, in order to receive initial access control rules for a new principal, *sb* has to communicate a round trip to where *pb* resides. However, this communication is through cheaper best-effort message delivery with timeout and retries.

We measure message delivery latency using a latency sampler that publishes messages through *pb* and subscribes to its own messages at *sb*. Delivery start latencies are measured as the time between a subscriber submits its subscription and when it receives the first message.

To measure delivery start latencies, we set up a first group of subscribers that stay constantly connected to *sb*. This group connect to the system using principal $p_1$ and subscribe to all published messages. We use a second group of subscribers with the same principal $p_1$ to measure the *local* delivery start latency. This local delivery start latency includes the time taken to send the subscription to *sb*, the

time taken to retrieve the subscribing rights for $p_1$ in local cache of $sb$, the time taken to compute the restricted subscription, and the delivery latency of the first message for the subscription. As the first group of subscribers subscribed to all published message, messages matching the second group of subscriber are already in transmission even when the subscriptions are being processed. To measure the remote delivery start latency, we use a single subscriber which connects to the system using a different principal $p_2$. The remote delivery start latency thus includes the time taken to establish a subscription to the access control information of $p_2$ and the time taken to retrieve an initial state of access control rules for $p_2$ in addition to the local delivery start latency.

We inject messages into the system at a rate so that each subscriber should receive 200 messages per second. Thus the measurement of local delivery start latency is swayed by an average of $(1/200)/2 = 2.5$ milliseconds because a matching message may be already in transit.

Figure 6.11 shows the latency results with their standard deviations shown in error bars when there are 1, 4 and 7 hops from $pb$ to $sb$. In this test, the message delivery latency and the remote delivery start latency for new principals increase linearly as the hop count increases. The local delivery start latency does not increase with the hop counts. The new principal local delivery start latency does not increase until 7 hops, because the initial acl retrieval is done at subscriber connection time and thus runs in parallel with the subscriber submitting its subscription. In the case of 1 and 4 hops, the initial acl retrieval is able to complete before the subscriber submits its subscription.

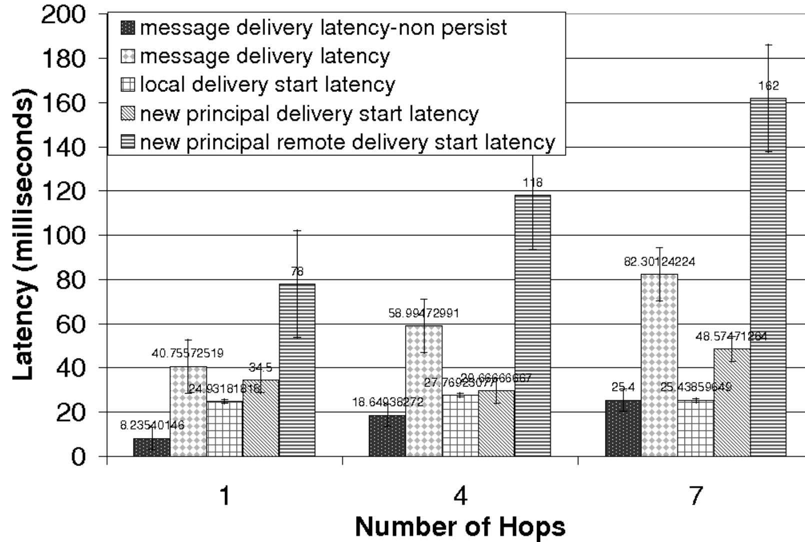



Figure 6.11: Latency Metrics with Dynamic Access Control

### 6.3.7 Scalability Measurements

In Section 6.1.3, we have discussed the scalability issues of the system with regard to the number of SHBs. We presented an optimization using a sufficient test result cache. We have also provided an asympototic analysis of the scalability in Section 6.2.3. In this section, we examine experimentally the system scalability with and without this cache mechanism. We further compare them with the experimental results obtained in the flooding scheme by using test setups that are *un-favorable* to subscription propagation.

#### Measurement

Figure 6.12 shows the network topology of our scalability experiments. The network contains a PHB $pb$, an intermediate routing broker $ib$ and a number of SHBs $sb_1$, $sb_2$, $\ldots$, $sb_n$. PHB $pb$ connects to intermediate routing broker $ib$ which in

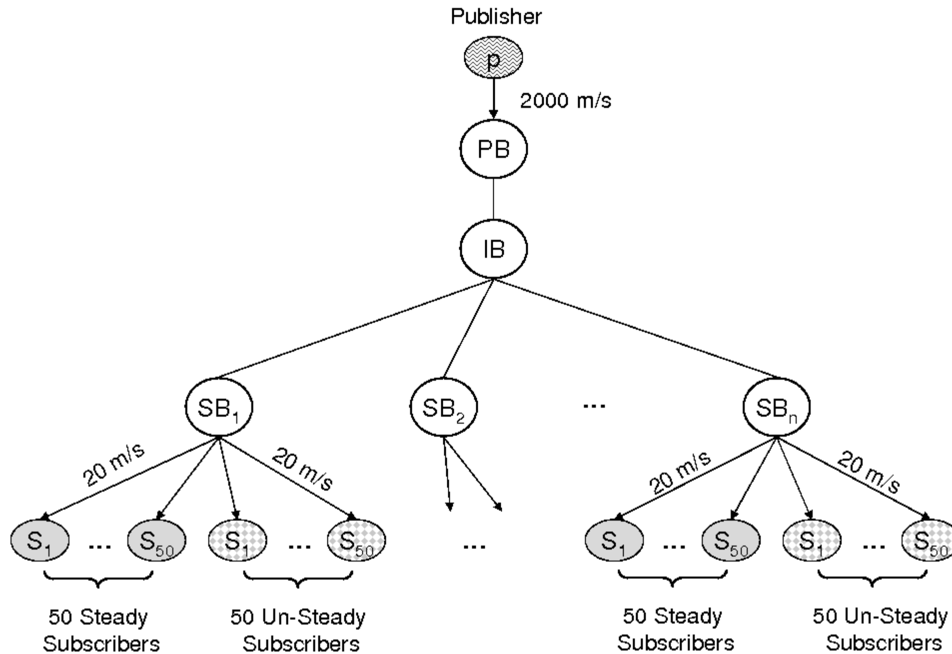



Figure 6.12: A Fan-out Topology Network for Scalability Measurements

turn connects to each of the SHBs. Each of these brokers is in a separate routing tree node. We vary the number $n$ of SHBs.

Messages are published through *pb* at a fixed rate of 2000 messages/second throughout the test. At each SHB, there are two groups of 50 subscribers. Each group receives messages at the rate of 1000 messages/second. One group is *steady* in that they stay constantly connected. The second group is *unsteady* in that they periodically un-subscribe and then re-subscribe. Since the subscribers present distinct subscriptions, each un-subscribe and re-subscribe action will cause the SHB's virtual clock to advance by 2. We setup the un-subscribe/re-subscribe action to occur at 2 seconds interval on average. Thus, each SHB's virtual time clock advances by 1 every second. In situations where un-subscribe/re-subscribe

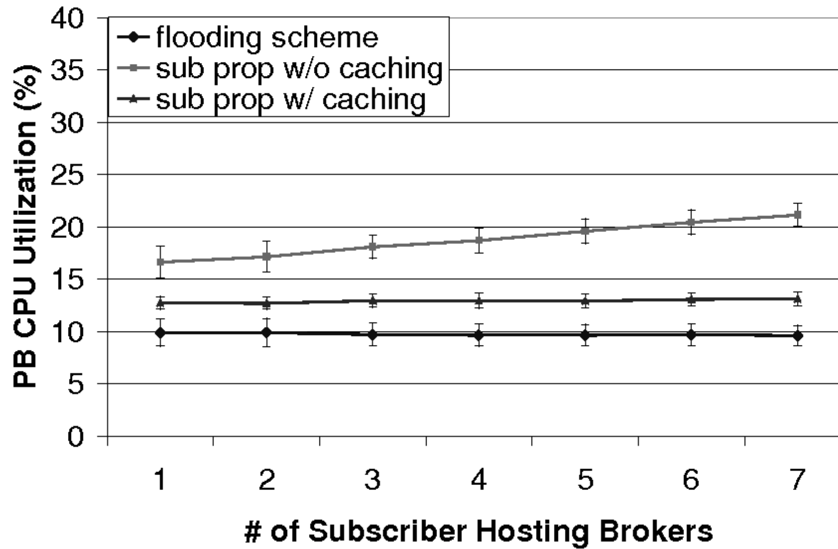



Figure 6.13: Scalability: PHB CPU Utilization Comparison

actions occur more frequently, they could be batched to reduce the rate at which the SHBs virtual clocks advance.

**Results**

Figure 6.13, 6.14 and 6.15 show the average CPU utilization at *pb*, *ib* and $sb_1$, ..., $sb_n$, each in the three schemes of flooding, subscription propagation without sufficiency test caching and subscription propagation with sufficiency test caching. The error bars show the standard deviation of the data samples.

In both the flooding scheme and the subscription propagation scheme *with* sufficiency result caching (Figure 6.13 and 6.15), CPU utilization at *pb* and $sb_1$, ..., $sb_n$ stays constant with $n$ changing from 1 to 7. The CPU differences at *ib* (Figure 6.14) and *sb* (Figure 6.15) between the two schemes are also very small. At broker *pb*, the CPU utilization in the subscription propagation scheme *with*

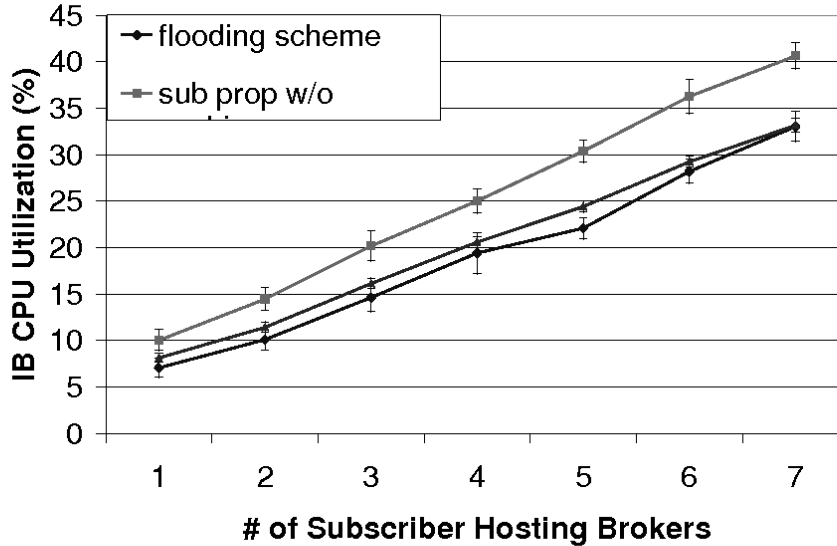



Figure 6.14: Scalability: Intermediate Broker CPU Utilization Comparison

caching is slightly higher (Figure 6.13). This is due to the sophisticated message encoding used in Gryphon. As *pb* assigns $V^{\mathrm{m}}$ digest to a message, the message has to be re-encoded and this is not needed with flooding in Gryphon. Such difference can be eliminated by encoding optimizations.

The brokers' CPU utilization in the subscription propagation scheme *without* caching show an observable linear increase with the number of SHBs at all PHB (Figure 6.13), intermediate routing broker (Figure 6.14) and SHBs (Figure 6.15). These overheads are successfully eliminated by the caching mechanism. This confirms to our analytical result presented in Section 6.2.3.

### 6.3.8 Failure Test

A property of our subscription propagation protocol is the lightweight failover characteristics of our approach. Even in the absence of a majority of brokers in a

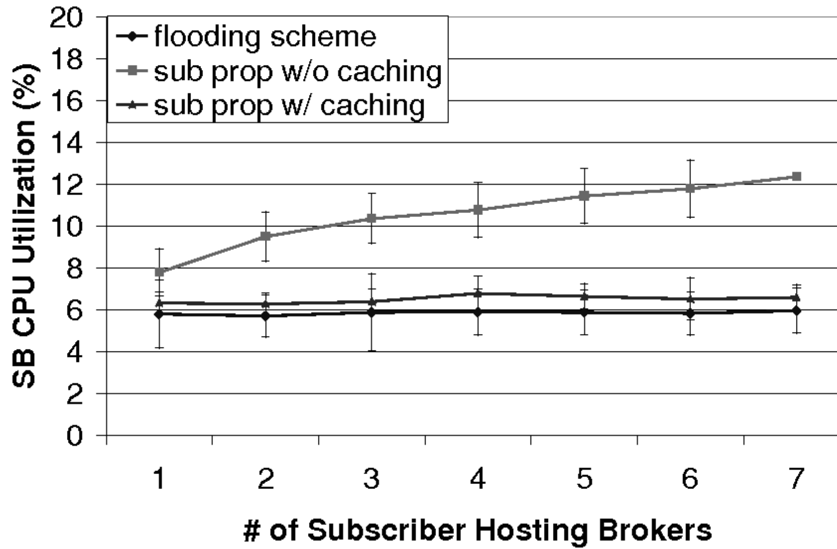



Figure 6.15: Scalability: SHB CPU Utilization Comparison

tree node, our system is able to accept new subscriptions and deliver messages for them. When a path fails, the system switches to the remaining available paths and provides continuing service.

Figure 6.16 shows the network topology of our test setup. The network is a redundant routing tree of 4 nodes and 6 brokers: broker $PB$ for publishers and $SB_{1,2}$ for subscribers, each in its own separate node and three intermediate routing broker $IB_1$, $IB_2$ and $IB_3$ that reside in the same tree node $N_{IB}$. We do not show the tree node of a broker if that broker is the only broker in the node. Broker $PB$ is connected to each of the intermediate routing broker $IB_1$, $IB_2$, $IB_3$ in node $N_{IB}$, which further connects to broker $SB_1$ and $SB_2$.

In this test, traffic from $PB$ to $SB_{1,2}$ is shared among intermediate broker $IB_1$, $IB_2$, $IB_3$. Messages are published through $PB$ at a rate of 2000 messages per second. Initially, there are 2 groups of clients connected to broker $SB_1$ and 1

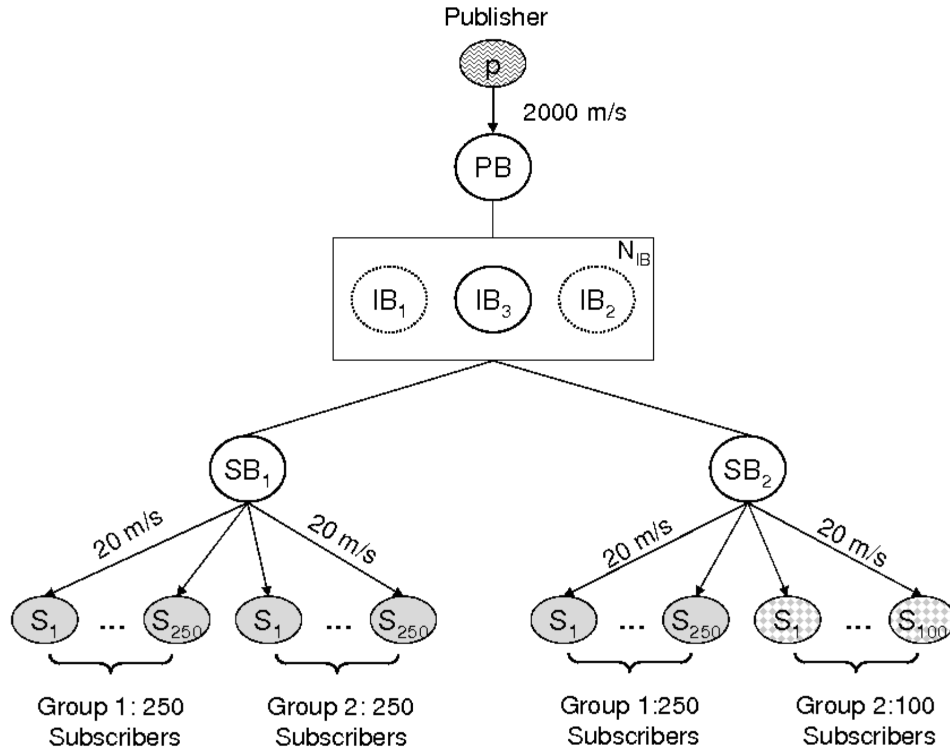



Figure 6.16: Topology Network for Fault Tolerance Test

group to broker $SB_2$, each group with 250 subscribers each getting 20 messages per second. Thus, the aggregated message rate per group is 5000. These 3 groups of 250 subscribers subscribe to half of the data messages that are published.

A second group of 100 clients connect to broker $SB_2$ at a later time, each getting 20 messages per second. These 100 new clients subscribe to the second half of all the data messages that are published. The aggregated message rate for this group is 2000 messages per second. Figure 6.17(a) shows the message rates for one of the first 3 groups and the fourth group. Figure 6.17(b) shows the CPU utilization at $IB_1$, $IB_2$, $IB_3$.

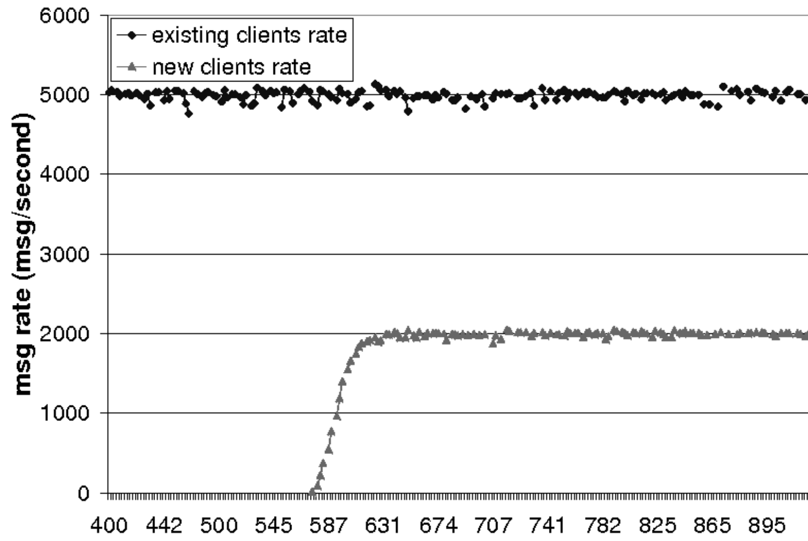
At time 400, only the first half of published messages are subscribed to, the

messages are routed through $IB_1$ and $IB_2$. Broker $IB_3$ is not used because of the simple hashing scheme used for load balancing. At time 475, $IB_1$ crashes, the system fails over to $IB_3$, and CPU utilization at $IB_3$ increases to 4% to the same as $IB_2$. About 30 seconds later, at time 505, $IB_2$ crashes, and all messages on the first 50 topics are routed through $IB_3$. CPU utilization at $IB_3$ doubles to 8%. During these routing changes the client message rate is not affected. At time 565, a new group of 100 subscribers starts to connect. These new subscribers subscribe to the second half of published data messages. Even though only $IB_3$ is available, our approach is able to make progress and starts to deliver messages for the new clients. When $IB_1$ and $IB_2$ recover about 130 and 160 seconds later at time 691 and 731, traffic is once again shared among the available paths. During this process, service to clients is not affected as their message rate stays constant, as shown in Figure 6.17(b).

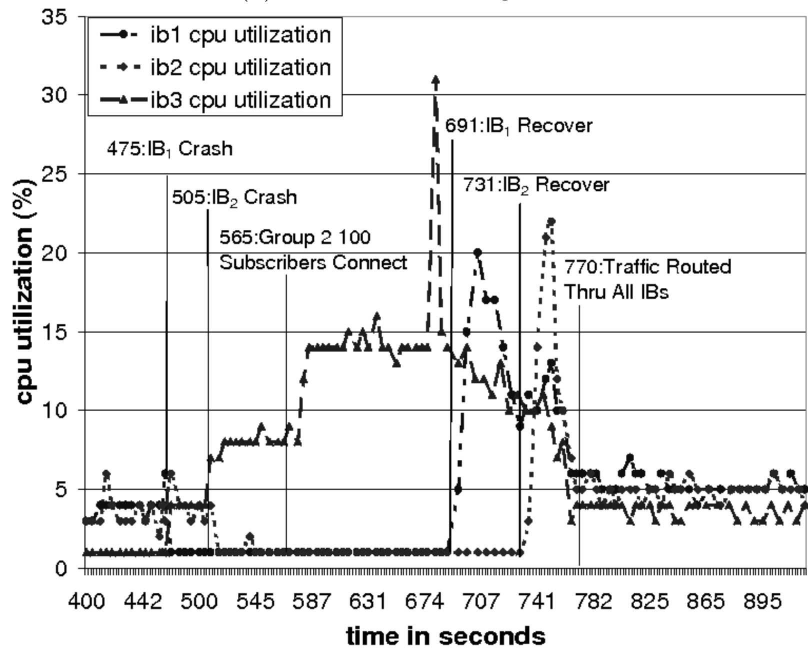
## 6.4 Summary

In this chapter, we discussed several implementation issues that range from how to build a practical system with multiple PHBs to how to improve system performance through identifying the hot contention point and by mechanisms like sufficiency test result caching and multi-reader-single-writer locks for the hot contention point.

We have also presented both analytical and experimental evaluation of our system. Our results demonstrate that the system is efficient, scalable and provides light-weight failover.




(a) subscriber message rate




(b) CPU utilization at $ib_1, ib_2, ib_3$

Figure 6.17: Client Message Rate and Intermediate Broker CPU Utilization with Crash Failure

# Chapter 7

# Conclusions and Future Work

In this dissertation, we have studied the problem of subscription propagation, content-based routing and reliable delivery in content-based publish/subscribe systems deployed over networks with redundant paths. In particular, we presented a general model of subscription propagation, content-based routing and reliable delivery. Based on this model, we designed a generic algorithm that is capable of performing subscription propagation in a redundant broker network without requiring agreement between the redundant peer brokers. Coupled with a *sufficiency-directed* content-based routing algorithm, this protocol can support reliable delivery even in the presence of redundant paths, communication asynchrony, link failures and broker crashes.

We have implemented a subscription propagation protocol using a special encoding of virtual start times of subscriptions and virtual time vectors. We further extended our work to support the dissemination of access control information to content-based publish/subscribe systems. We developed a content-based access control and deterministic service model of dynamic access control changes. We

evaluated the performance of the protocol both analytically and empirically.

Our results demonstrated the viability of using control information to improve messaging service performance through content-based routing. We demonstrated that this can be achieved in systems that require reliable delivery and high availability through redundant routing brokers and paths. We have shown that control information propagation and content-based routing can be implemented efficiently through the use of distributed protocols that are asynchronous and light weight. There is no need to achieve concensus among the redundant brokers/paths. Coupled with proper system implementation and optimization techniques, a content-based publish/subscribe system can achieve efficiency through the use of control information to suit the needs of applications that require selective, regional, reliable and highly available distribution of messages.

This work can be further extended in the following ways:

- Research and design algorithms for conducting covering test that are sound, efficient and more complete in that if there is a covering relationship, it can usually finds it, i.e., not too conservative.

- Research and design algorithm and mechanism that adaptively conducting different degrees of subscription merging in stable system operations, ranging from the finest degree of matching exactly what is subscribed to the coarsest of matching all messages. The adaptive decision should be based on the application scenarios, subscription patterns, message patterns and their temporal distributions. The algorithm should make a sound decision in trading off matching time versus communication cost. The mechanism should be efficient in carrying out the decisions, reporting of system conditions such as

traffic load.

In addition, event processing systems have evolved from traditional systems that focus on efficient event filtering and routing to systems that aim at providing rich support in event patern detection and business decision making. This new type of systems are typically built on top of the event delivery service of an existing messaging system. We anticipate that future directions of research in these systems will be on the timeliness of service on the low layer of message delivery to the richness as well as timeliness of support in the upper layer. The solutions will innovatively combine technologies from various fields including distributed systems, database systems, data mining and artificial intelligence.

# Bibliography

[1] *Corba Fundamentals and Programming.* John Wiley & Sons Inc (Computers), 1996.

[2] M. Abadi and J. Feigenbaum. Secure circuit evaluation. *Journal of Cryptology*, 2(1), 1990.

[3] M. Abadi, J. Feigenbaum, and J Kilian. On hiding information from an oracle. *Journal of Computer & System Sciences*, 39(1), 1989.

[4] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Principles of Distributed Computing, 1999*, 1999.

[5] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. *SIGOPS Operating Systtem Review*, 35(5), 2001.

[6] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Survey*, 36(4), 2004.

[7] Jean Bacon, David Eyers, Ken Moody, and Lauri Pesonen. Secur-

ing publish/subscribe for multi-domain systems. In *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference*, 2005.

[8] Jean Bacon, Ken Moody, and Walt Yao. A model of Oasis role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4), 2002.

[9] Sebastien Baehni, Patrick Th. Eugster, and Rachid Guerraoui. Data-aware multicast. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, 2004.

[10] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999*, 1999.

[11] Andras Belokosztolszki, David M. Eyers, Peter Pietzuch, Jean Bacon, and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In *Proceedings of the International Workshop on Distributed Event-Based Systems*, 2003.

[12] Sumeer Bhola, Robert Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, 2002.

[13] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably supporting durable subscriptions in a publish/subscribe system. In *Proceedings of the*

*International Conference on Dependable Systems and Networks (DSN'2003)*, 2003.

[14] Jean-Camille Birget, Xukai Zou, Guevara Noubir, and Byrav Ramamurthy. Hierarchy-based access control in distributed environments. In *Proceedings of the IEEE International Conference on Communication*, 2001.

[15] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), 1993.

[16] Kenneth Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, 1987.

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

[18] R. Chand and P.A. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'03), Cambridge, MA*, April 2003.

[19] Raphael Chand and Pascal Felber.

[20] David Cheriton and Willy Zwaenepoel. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems (TOCS)*, 3(2), 1985.

[21] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong.

Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies*, 2001.

[22] Jon Crowcroft and Ian Pratt. Peer to peer: peering into the future. *Advanced lectures on networking*, 2497, 2002.

[23] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9), September 2001.

[24] A.K. Datta, M. Gradinariu, M. Raynal, and G. Simon. Anonymous publish/subscribe in p2p networks. In *Proceedings of International Parallel and Distibuted Processing Symposium*, 2003.

[25] Danny Dolev and Dalia Malki. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), 1996.

[26] Miscrosoft Erik Christensen, IBM Research Francisco Curbera, Microsoft Greg Meredith, and IBM Research Sanjiva Weerawarana. Web services description language (wsdl) 1.1.

[27] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, 1992.

[28] International Committee for Future Accelerators (ICFA). Standing Committee on Inter-Regional Connectivity (SCIC). Icfa scic network monitoring report, 2006.

[29] International Committee for Future Accelerators (ICFA). Standing Commit-

tee on Inter-Regional Connectivity (SCIC). Icfa scic network monitoring report, 2003.

[30] Object Management Group. Corbaservices: Common object services specification. 1998.

[31] RTI Inc. Ndds: The real-time publish/subscribe middleware.

[32] SONIC Software Inc. http://www.sonicsoftware.com/index.ssp.

[33] TIBCO Software Inc. http://www.tibco.com.

[34] David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3), 1985.

[35] Paul Judge and Mostafa Ammar. Security issues and solutions in multicast content distribution: A survey. *IEEE Network Magazine*, Januaray/February 2003.

[36] Marc Kaplan. Methods for efficient multicast message distribution in a highly scalable and available network messaging service. us patent pending:yor8-1999-0920, 1999.

[37] Balachander Krishnamurthy and David Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10), 1995.

[38] Peter S. Kruus. A survey of multicast security issues and architectures. In *Proceedings of the 21st National Information Systems Security Conference*, 1998.

[39] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGOPS Operating System Review*, 34(5), 2000.

[40] Leslie Lamport. Time, clock, and the ordering of events in a distributed system. *Communications of the ACM*, 21, 1978.

[41] Guoli Li, Shuang Hou, and Hans-Arno Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS*, 2005.

[42] Jie Lu and Jamie Callan. Content-based retrieval in hybrid peer-to-peer networks. In *Proceedings of the twelfth international conference on Information and knowledge management*, 2003.

[43] Microsoft Luis Felipe Cabrera, Microsoft Craig Critchley, Microsoft Gopal Kakivaya, Microsoft Brad Lovering, BEA Systems Matt Mihic, BEA Systems David Orchard, TIBCO Software Shivajee Samdarshi, Microsoft Jeffrey Schlimmer (Editor), Microsoft John Shewchuk, and Microsoft David Wortendyke. Web services eventing.

[44] Silvano Maffeis. Adding group communication and fault-tolerance to corba. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, 1995.

[45] Patrick Mazza. Powering up the smart grid. In *http://www.climatesolutions.org/pubs/pdfs/PoweringtheSmartGrid.pdf*. Climate Solutions, 2005.

[46] Zoltan Miklos. Towards an access control mechanism for wide-area publish/subscribe systems. In *Proceedings of International Workshop on Distributed Event-Based Systems*, 2002.

[47] Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), 1996.

[48] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, September 2002.

[49] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Filter similarities in content-based publish/subscribe systems. In *Proceedings of International Conference on Architecture of Computing Systems (ARCS'02)*, 2002.

[50] Vinod Muthusamy and Hans-Arno Jacobsen. Small-scale peer-to-peer publish/subscribe. In *P2P Workshop MobiQuitous 2004*, 2004.

[51] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Rob Strom, and Daniel Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In *Proceedings of Middleware*, Springer LNCS 1795, 2000.

[52] Lukasz Opyrchal and Atul Prakash. Secure distribution of events in content-based publish subscribe systems. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[53] Peter R. Pietzuch. *Hermes: A Scalable Event-based Middleware*. PhD thesis, University of Cambridge, 2004.

[54] Peter R. Pietzuch and Jean Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[55] Federal Information Processing Standards Publication. Data encryption standard, 1977.

[56] Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Computing Surveys*, 35(3), 2003.

[57] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Computer Communication Review*, 31(4), 2001.

[58] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, 2001.

[59] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2), 2003.

[60] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, 2001.

[61] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design

of a large-scale event notification infrastructure. In *Proceedings of 3rd International Workshop on Networked Group Communication (NGC 2001), UCL, London, UK*, November 2001.

[62] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (Middleware'01)*, 2001.

[63] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proceedings of AUUG2K, Canberra, Australia*, April 2000.

[64] David Spence and Tim Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[65] Mudhakar Srivatsa and Ling Liu. Securing publish-subscribe overlay services with eventguard. In *Proceedings of the 12th ACM Conference on Computer and Communication Security*, 2005.

[66] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1), 2003.

[67] Yan Sun and K. J. Ray Liu. Scalable hierarchical access control in secure group communications. In *Proceedings of the 23rd Conference of the IEEE Communications Society*, 2004.

[68] David Tam, Reza Azimi, and Hans-Arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Proceedings of 1st International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, 2003.

[69] Akamai Technologies, Computer Associates International, Fujitsu Laboratories of Europe, Globus, Hewlett-Packard, IBM, SAP AG, Sonic Software, and TIBCO Software. Publish-subscribe notification for web services (verison 1.0).

[70] Akamai Technologies, Computer Associates International, Fujitsu Laboratories of Europe, Globus, Hewlett-Packard, IBM, SAP AG, Sonic Software, and TIBCO Software. Ws-base notification.

[71] Akamai Technologies, Computer Associates International, Fujitsu Laboratories of Europe, Globus, Hewlett-Packard, IBM, SAP AG, Sonic Software, and TIBCO Software. Ws-brokered notification.

[72] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, 2003.

[73] Christos Tryfonopoulos, Stratos Idreos, and Manolis Koubarakis. Publish/subscribe functionality in ir environments using structured overlay networks. In *Proceedings of 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2005.

[74] Robbert van Renesse, Kenneth Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Communications of the ACM*, 39(4), April 1996.

[75] Antonino Virgillito. *Publish/Subscribe Communication Systems: from Models to Applications.* PhD thesis, Universita degli Studi di Roma "La Sapienza", November 2003.

[76] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, 2002.

[77] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen Wang. Subscription partitioning and routing in content-based publish/subscribe systems. In *Proceedings of 16th International Symposium on Distributed Computing(DISC'02)*, 2002.

[78] Yan Yan, Yi Huang, Geoffrey Fox, Shrideep Pallickara, Marlon E. Pierce, Ali Kaplan, and Ahmet E. Topcu. Implementing a prototype of the security framework for distributed brokering systems. In *Proceedings of the International Conference on Security and Management*, 2003.

[79] Eiko Yoneki and Jean Bacon. An adaptive approach to content-based subscription in mobile ad hoc networks. In *Proceedings of 2nd IEEE Annual Conference on Pervasive Computing and Communications, Workshop on Mobile Peer-to-Peer Computing*, 2004.

[80] Eiko Yoneki and Jean Bacon. Content-based routing with on-demand multicast. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, 2004.

[81] B. Zhao, L. Huang, A. Joseph, and J. Kubiatowicz. Exploiting routing redundancy using a wide-area overlay. Technical Report UCB/CSD-02-1215, University of California, Berkeley, 2002.