The Design of A Fast and Flexible Internet Subscription System Using Content Graphs

by

Joanna L. Kulik

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

March 2004 [June 2004]

Author .....
Department of Electrical Engineering and Computer Science
March 23, 2004

Certified by...
David Clark
Senior Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Design of A Fast and Flexible Internet Subscription System Using Content Graphs

by

Joanna L. Kulik

Submitted to the Department of Electrical Engineering and Computer Science
on March 23, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation describes the design and evaluation of the Fast, Flexible Forwarding system (F3), a distributed system for disseminating information to networked subscribers. It examines existing subscription approaches, proposes F3 as an alternative to these approaches, and presents results from comparisons of F3 and other subscription approaches.

Existing subscription approaches examined in the dissertation fall into three categories: unicast, single-identifier multicast, and content-based multicast systems. Careful examination of these approaches suggests that none is able to support complex subscription requests from large numbers of subscribers at high data rates.

F3, the systems proposed as an alternative, shares many features with other multicast systems. Like many multicast systems, for example, F3 uses an overlay network of routers to distribute messages to subscribers. F3 differs from other systems, however, in its use of preprocessors to analyze messages before routing begins. Preprocessors carry out analyses of the relationships between subscription topics, and store the results in special content graph data-structures. Preprocessors share the results of their analyses by distributing content graphs to routers in the F3 network. Using content graphs, F3 routers can determine the relationships between subscriptions and notifications more efficiently than in previous approaches.

Four studies compared performance of F3 and competing subscription systems. In the four studies, subscription systems handled such tasks as disseminating baseball scores, distributing traffic alerts, and disseminating generic subscriptions formatted as attribute-value pairs. The four studies examined system performance in both simulated network environments and on a working router. Performance characteristics examined in the studies included size of forwarding tables and processing speeds at routers.

Results from these experiments showed that F3 does not overproduce messages, as do unicast systems. F3 also outperformed single-identifier multicast systems in such areas as message production, table size, and subscription overhead. The most

significant finding of the studies, however, was that F3 processing speed surpassed the speed of a state-of-the-art content-based system by orders of magnitude in scenarios with large numbers of subscribers. Overall, these results suggest that F3 is a promising development in the area of Internet subscription systems.

Thesis Supervisor: David Clark
Title: Senior Research Scientist

You're the top! You're the Colosseum,
You're the top! You're the Louvre Museum,
You're a melody from a symphony by Strauss,
You're a Bendel bonnet, a Shakespeare sonnet,
You're Mickey Mouse.

From "You're the Top"
Music and lyrics by Cole Porter

Acknowledgments

Counting both my undergraduate and graduate days, I have spent more than a third of my life at MIT. After all these years, I am happy to say, ILTFP, or I Love This Fascinating Place. It's where I learned that no challenge is ever too difficult to face, no hypothesis too crazy to entertain, and no solution too perfect to question. MIT helped me turn a childhood hobby into an adult career, and it transformed me from a budding Dramashop performer into a committed computer scientist. First of all, therefore, I have to thank MIT.

What makes MIT so special to me is not the familiar landmarks – the nerd crossing on Kresge Oval, the Smoot marks on Harvard Bridge, or the Infinite Corridor - but rather the wonderful people that I've met here. Foremost among them is my thesis advisor and hero, Dave Clark. Most people know Dave as a brilliant Internet pioneer, systems architect, writer, speaker, and all-around stand-up guy. I am lucky enough to be in the band of people who also know Dave as a truly generous advisor. Dave encouraged me from the start to find a research problem that I was passionate about, and he went on to challenge me to find my own solution to it. During my dissertation struggles, he provided me with anecdotes, opinions, insights, criticism, and encouragement whenever needed. Because of Dave, I had a graduate school experience of uncommon creative and intellectual freedom, and I am grateful for it.

I am also thankful for the guidance and support of my dissertation readers, Barbara Liskov and Hari Balakrishnan. In my sophomore year at MIT, I learned from Barbara that engineering was an art as well as a science. This insight first drew me into research in computer science, and it has influenced my work ever since. More recently, Barbara gave a careful reading to my thesis proposal and my dissertation, and her thoughtful comments helped me to make important improvements to both. Five years ago, Hari Balakrishnan provided me with my first opportunities to publish

a research paper and to assist in the development of his graduate course. Later, in a brief chat, Hari set me on the road toward finding a dissertation topic in the field of event notification. Another chat with Hari led me to design the key experiments for my dissertation. I am deeply indebted to Hari not only for these contributions but also for all the other insights and opportunities that he has provided for me throughout my doctoral career.

I also extend heartfelt thanks to John Wroclawski and Karen Sollins. John and Karen critiqued various versions of practice talks until I finally felt able to present my ideas with some confidence and poise. I enjoyed John's wry humor at meetings of our research group as much as I appreciated Karen's down-to-earth intuitions. John and Karen have led everyone in our research group to higher standards of research.

I will never forget my MIT friends and the conversations that I had with them. My thanks go out to Xiaowei Yang, Peyman Faratin, Becky Shepardson, George Lee, Arthur Berger, Alex Snoeren, Chuck Blake, Hariharan Rahul, and Elliot Schwartz. In our conversations together, we raised sunken submarines, founded billion-dollar startups, and pondered the low pixel resolution of impressionist art. Most important, we refined each other's ideas continually. Our conversations shaped my research as much as coursework did.

Special thanks to Steve Bauer, our local connoisseur of all things Oreo. Steve became my officemate four years ago, and little did he know then how much the job would entail. He has served as an armchair psychologist, research advisor, lonely hearts consultant, entrepreneurial ally, and true-blue friend. It was Steve who first suggested that I think about privacy in subscription systems, and this suggestion led me to develop the fundamental approach presented in this dissertation.

I have also depended on several friends outside of school to brighten the way through these last years. These include Nina Yuan, Cindy Adams, Chia Lin Chu, Ana Licuanan, Julie Wallace, and Lee Kilpatrick. Most of all, I have Mike Halle thank for his insightful vision, unflagging friendship, and steadfast support. Ever the holographer, Mike ensured that I didn't miss out on all the marvelous colors and dimensions of life while completing this work.

Finally, I am indebted to my favorite researchers, my parents, Drs. James and Chen-Lin Kulik, for their contributions to this work. Ever since I can remember, my father's writing has served as a model for my own. I can only hope to communicate ideas half as elegantly and simply as he can. My mother's determination and independence have also served as a model. Not only did my mother buck the expectations

for her gender and get her own PhD, she left her country, learned a new language, and assimilated a new culture to do it. My parents have been collaborators for more than three decades in research, in building a home, and in raising a child, and to me, their daughter, their complete partnership seems a rare and wonderful thing. To my parents, truly you are the top!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet is, among other things, the greatest information warehouse ever built. But information is not meant just for storage on warehouse shelves. To be useful, information must get into people's hands and heads, and it must often get there very quickly. Investors need to know immediately when stock market prices begin to tumble. Researchers need to know right away when relevant research findings are released. Everyone needs to know when threatening weather is on the way. Information on such topics makes its way to the Internet very quickly, but the information gets into the hands of users much more slowly. The Internet does not send out alerts, and users have to find new Internet information by themselves.

The Internet provides some tools to help users find what they need. Users can employ search engines like Google to dig deep into the Internet warehouse. They can bookmark relevant Web pages to make future lookups easier. They can use canned queries to automate their queries even more, and they can schedule these queries to run periodically. But use of such tools do not alert users to the arrival of new information on the Internet. To catch new information as it is created, user-initiated queries would have to run almost continuously, and such queries would be highly inefficient. Continuous queries by millions of Internet users would clog the network with redundant queries and responses.

Researchers have for nearly a decade been looking for a better way to change the Internet from an information warehouse to an information clearinghouse. The mechanisms that they are developing can be grouped together under the heading Internet subscription systems. Put simply, a subscription system is a communication mechanism for delivering information from information providers to subscribers over the network. Subscribers first sign up to receive information on topics that are important to them. When an information provider submits new information to the subscription

system, the information is automatically distributed to the appropriate subscribers. Internet subscription systems are thus designed to give subscribers exactly the information they need as quickly as easily possible.

Developers began using subscription systems to handle communications in networked applications almost two decades ago. Early developers approached subscription systems from a variety of vantage points, and they gave their subscription devices a variety of names. In 1998 a workshop held at the University of California at Irvine, the Workshop on Internet-Scale Event-Notification (or WISEN), gave some focus to these disparate efforts. The workshop was one of the first occasions when researchers from around the world convened to discuss their shared interests in subscription systems, and it gave the field its first widely used name, wide-area event notification. More important, the workshop provided taxonomies of relevant systems, presented a bibliography of references, and focused attention on the significant problem of extending event notification to global scale systems. The workshop thus became a milestone in the development of subscription systems.

The next phase in the development of Internet subscription systems began when commercial firms jumped onto the event notification bandwagon and began developing what they called publish subscribe systems. Microsoft began work on its Scribe and Herald systems [49, 9] ; IBM developed Gryphon [14]; and AT&T developed READY [24]. Publish-subscribe systems were specifically designed to publish HTML and XML text documents, but the high-profile work of major technology companies brought a new identity to the area. The terms publish-subscribe (or pub-sub, for short) all but replaced the term event notification as the name for the area.

Today, many leading technology firms have Internet subscription systems under development. Commercial systems already in use include Talarian Products' Smart Sockets and TIBCO's Rendezvous [59]. Academic centers working on the development of Internet subscription systems include the University of Colorado's Software Engineering Research Laboratory [13] and the Distributed Systems group at the University of Cambridge Computer Laboratory [44]. Academic interest in these systems is not restricted to a single field. One is just as likely to find articles on subscription systems in the proceedings of SIGSOFT as one is to find them in PODC, Middleware, SOSP, or SIGCOMM.

In addition, several organizations are now using Internet subscription systems to send information to subscribers. For example, the Major League Baseball website sends breaking baseball news [35]; Google and the New York Times provide news updates by email [58, 22]; businesses use TIBCO [59] servers to subscribe to inventory

updates; and brokerage firms such as ScottTrade provide streaming, real-time stock market quotes [50]. These applications are probably just a hint of things to come. Users should soon be able to pull up news pages that contain the latest news on topics of their own choice, for example. Like current television news channels, the pages can include streaming video announcement on the stock-market, sports, weather, traffic, and text news. Unlike television news, the content of these pages can be completely customized. The weather reports will be for a specific user's geographic area; the traffic alerts will relate to her route to work; the sports reports will be on her favorite teams and players; the news topics will relate to her interests; and the stock market prices will be on stocks that she is tracking.

The Internet subscription systems available or under development today can be divided into three broad categories. Unicast systems transmit notifications directly to subscribers over the Internet's existing mechanisms. Unicast systems are commonplace in today's commercial world. The email news updates sent out by Google and the New York Times, for example, use a unicast routing system. Single-identifier multicast systems send messages to discrete message channels to which customers with identical interests subscribe. Single-identifier multicast systems seem especially well-suited for distribution of on-line entertainment events. An entertainment provider can route a streaming video of an event to all subscribers subsumed under a single Internet address. The most popular current approach to subscription systems, called content-based multicast systems, forward messages based on the text content of the messages. Content-based systems rely heavily on intelligent routers that parse message content and use the results of such analysis to forward messages.

None of these systems may be adequate for large, complex applications in which millions of users sign up for notices using complex, overlapping subscription categories. Unicast systems can easily clog network routers with numerous copies of the same message [17]. With small numbers of subscribers, the problem of redundancy may be manageable, but with global scale systems, the problems can become overwhelming. Single-identifier multicast systems cannot handle complex and evolving subscription categories efficiently [1]. And content-based systems take a long time to process notification messages. Studies have shown that the amount of time that it takes a content-based router to match a notification with its subscriptions grows linearly with the number of subscriptions in its table [2]. As a result, the development of efficient matching algorithms for content-based systems has become a hot topic of research [2, 46, 6, 43, 3, 12].

What is needed is an alternative system that overcomes the problems of unicast,

single-identifier multicast, and content-based multicast systems. This thesis describes the development and evaluation of such a system. The Fast, Flexible Forwarding System (or F3) is designed specifically for large-scale, complex applications, and it uses distributed multicast mechanisms to support such applications. Applications that use F3 must pass messages through preprocessors before submitting the messages to the F3 network. These preprocessors identify relationships between messages, and encode them in data-structures called content graphs. Using content graphs, routers are able to match subscriptions with notifications efficiently without reading the application-level contents of messages. Using this approach, F3 is able to support complex, application-level interfaces without using complex, application-level machinery within the network itself.

This thesis also examines performance of F3, single-identifier, and content-based routing systems in a series of simulations. The scenarios used in the simulations involve three tasks: sports announcements, traffic alerts, and generic attribute-value notifications. These results show that the amount of time that it takes F3 to match subscriptions with notifications grows logarithmically, rather than linearly, with the number of entries in its table. However, it takes slightly longer to set up subscriptions in F3 than in other systems. Overall, these results suggest that F3 is a promising development in the area of Internet subscription systems.

The rest of this thesis presents the F3 subscription system in further detail. Chapter 2 describes the central problem addressed by this thesis, the design of Internet subscription systems, and analyzes previous approaches to these systems. Chapter 3 describes an alternative solution to this problem, the F3 architecture. Chapter 4 evaluates F3, both theoretically as well as through experimental comparisons to other systems. Chapter 5 puts F3 in the context of related work in subscription systems. Finally, Chapter 6 discusses future directions for work in this area and Chapter 7 concludes.

Chapter 2

Background of the Problem

A variety of Internet services now use subscription systems to forward information to interested parties. The Rebecca subscription system supports on-line trading [39]. Xfilter disseminates XML news documents [3]. DEEDS helps networked participants conference between remote sites [18]. Gryphon delivers scores for sports events such as the US Tennis Open and the Olympics [14]. Varied as these applications are, they provide only a hint of the range of subscription applications that we can expect to see in the future.

Can a single subscription system support a full range of Internet applications? What would an adequate subscription system look like? The main purpose of this chapter is to answer such questions. The chapter first looks at types of applications that can be helped by subscription systems. It then sets up criteria that subscription systems must meet to support these applications. The final section of the chapter uses research findings to determine how well current subscription systems meet these criteria.

2.1 Applications of Subscription Systems

One of the reasons that there has been fervent interest in subscription systems recently is that scientists have identified so many applications that may benefit from their eventual deployment. Examples of such applications include:

- *Remote monitoring.* These applications gather data from remote sites for immediate transmission to distant users. Examples of remote monitoring applications include a banking system that monitors a fleet of ATM machines; a traffic alert system; on-line auctions; web-log (or "blog") updates; intrusion detection sys-

tems; emergency alert systems; schedule services for planes, trains, and buses; monitors of gas pipelines and electric grids, and weather alert services.

- *Distributed data repositories.* These repositories store multiple copies of the same data, and one of the key challenges in managing these repositories is keeping all sets of the same data consistent. Using a subscription system, each repository holding a dataset can notify other holders of the set whenever a change in a particular piece of data occurs. Digital libraries, HTML document caches, and software distribution mirrors are all examples of distributed data repositories.

- *Business applications.* Businesses use a variety of applications to track their operations. Examples of business information that might be distributed through a subscription system include inventory updates and supply-chain data.

- *Multimedia applications.* A wide-area, shared whiteboard is an example of an interactive, multimedia application that might use a subscription system to share information. Users in different geographic areas can share their ideas on a wide-area whiteboard just as simply as they can when working together on the same whiteboard in a single room. A subscription system can help propagate any change made by a user to all other sites. Other examples of multimedia applications that could use subscription systems are massively multiplayer on-line games and on-line entertainment events.

These are just a few of the applications that may benefit from subscription systems. In fact, a subscription system can help any networked application whose operations are driven by the occurrence of unpredictable events.

2.1.1 Application Characteristics

This simple taxonomy provided above suggests that a subscription system infrastructure must support service providers with very different needs and requirements. Consider, for example, two remote monitoring systems: an intrusion detection system and a traffic alert system. An intrusion detector might request notifications about a small number of discrete locations. Subscriptions for traffic alerts might cover complex and overlapping traffic routes. Table 2.1 further illustrates dimensions in which subscription services vary and services that fall at the opposite ends of these dimensions.

Characteristic	Extremes	
Subscription complexity	A subscriber to an intrusion detection system might request notifications about a handful of conditions.	A subscriber to a traffic alert system can request notifications from a complex list of overlapping traffic routes.
Data format	A document repository system may distribute HTML documents	A shared whiteboard application may distribute multimedia, graphical data
Message size	A location-tracking system can fit an notification into a single, MTU-sized message.	A camera sensor might need to use multiple messages to carry a single notification.
Reliability	A stock-market ticker may be able to tolerate the loss of an occasional message.	An intrusion-detection system may not be able to tolerate the loss of even one message.
Privacy	An airline gives the general public access to flight schedule updates.	A patient gives only her doctor access to medical updates.
Geographic range	Subscribers to a building evacuation alert are often limited to a specific office or organization.	Subscribers to an electronic auction system may span an entire country.
Delays	Subscribers to AP news announcements can tolerate seconds of delay.	A system that automatically responds to virus detection notifications may only be able to tolerate milliseconds of delays.
Frequency	A ticket sales event may occur only once.	Radio broadcast events occur continuously.
Number of users	Only one or two people may subscribe to a particular car's alarm.	Hundreds of thousands of users may to subscribe to a particular news update.

Table 2.1: Characteristics of subscription system applications.

2.2 Evaluation Criteria

Based on the examples provided in Table 2.1, it is clear that future subscription systems must be able to support a wide variety of application characteristics. It is possible to group the characteristics provided in Table 2.1 under three broad criteria: flexibility, efficiency, and analytic ability. Flexibility refers to the ability of the system to support services with varying characteristics. Efficiency is the ability of a system to function economically. Analytic ability is the ability of the system to analyze relationships among categories of subscriptions and notifications.

A flexible subscription system adapts easily to different and changing needs of users and providers. A flexible system would adapt to each of the following:

- *Multiple data formats.* The system would support a variety of data formats. For example, it would support both public, uncompressed text data and private, compressed, audio-stream data.

- *Varying reliability requirements.* The system would support both reliable delivery notifications and less reliable notifications.

- *Changing user needs.* The system would adjust easily to changing user needs. For example, a content provider could change the names of subscription categories without having to make major revisions throughout the network.

Flexibility is thus a broad requirement. Subscription systems that have this quality would not only support a variety of subscription services but the services would evolve and grow naturally without a major overhaul of the services.

An efficient subscription system would operate with little or no waste of system resources. There are several costs that are important to consider:

- *Traffic loads.* The system should deliver messages without creating excessive amounts of traffic in the network.

- *Forwarding times.* Routers should be able to direct messages efficiently from incoming links to outgoing links.

- *Storage requirements.* The amount of storage required to maintain subscription should not be excessive.

If the cost of a system is excessive, no one will use it, no matter how attractive the system is in other respects.

A system with analytic ability can analyze relationships within and between subscription and notification topics. An analytic system can tell when a single notification is relevant to multiple subscription categories so that the system can ensure that each subscriber receives only one notification when an event occurs. Systems without analytic ability cannot perform the necessary analyses and can flood a user's computer with duplicate notices on the same event.

2.3 Subscription System Fundamentals

Figure 2-1 depicts the fundamental elements that characterize most subscription subscription systems. Subscribers, shown at the bottom of (a), submit requests for information updates to the subscription system in the form of subscription messages. Information providers, shown at the top of the figure, provide these information updates in the form of notification messages.

Subscribers and information providers interact with the subscription system through the subscription system's interface. This interface dictates the services that subscribers and information providers can request from the subscription system. Subscription services typically include such basic activities as subscribing, notifying, and unsubscribing. Services may also include requesting different qualities of service, such as reliable versus less reliable delivery.

Between subscribers and information providers lies the Internet and the subscription system itself. Subscription systems use a topology of one or more event routers to disseminate updates from notification sources to appropriate subscribers. Event routers are sometimes called brokers or servers in the literature.

Multi-router subscription systems, also called *multicast* subscription systems, typically use an overlay network of dissemination trees to distribute messages to subscribers. In such systems, each router typically maintains two separate tables, a *routing table* and a *notification forwarding table*, as depicted in Figure 2-1 (b). A system's routing table specifies what route messages should take to reach a particular destination in the network. Most subscription systems use standard approaches to routing, such as shortest-path first routing [38], to create these tables. It is important to note that routing tables do not store information about subscriptions, or which messages should be sent to particular destinations in the network. Routing tables only indicate how these destinations may be reached through the router topology.

Routers store information about subscriptions in notification forwarding tables, also depicted in Figure 2-1 (b). Forwarding tables indicate which router interfaces

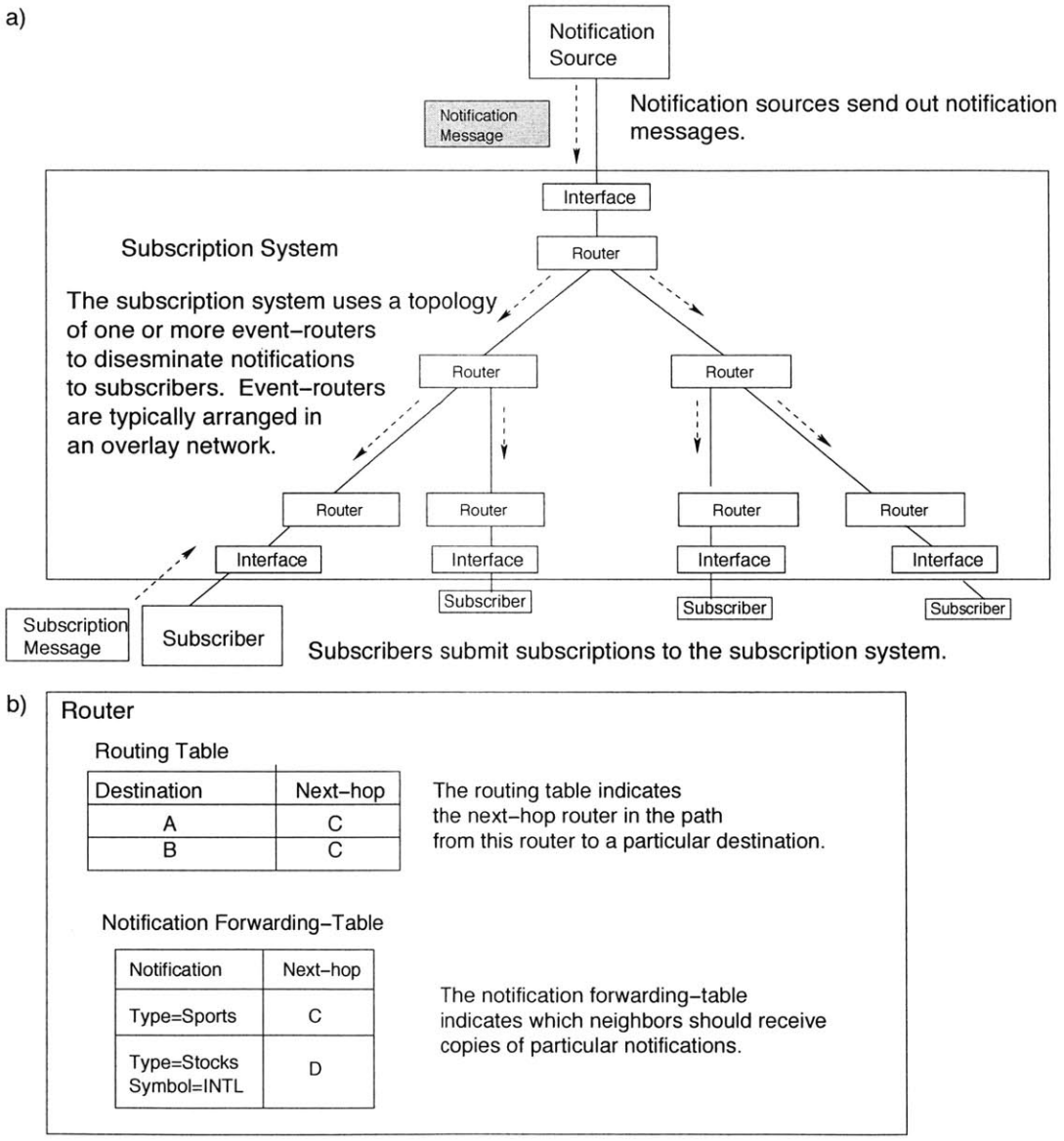


Figure 2-1: Overall subscription system architecture (a) and the architecture of an individual router (b).

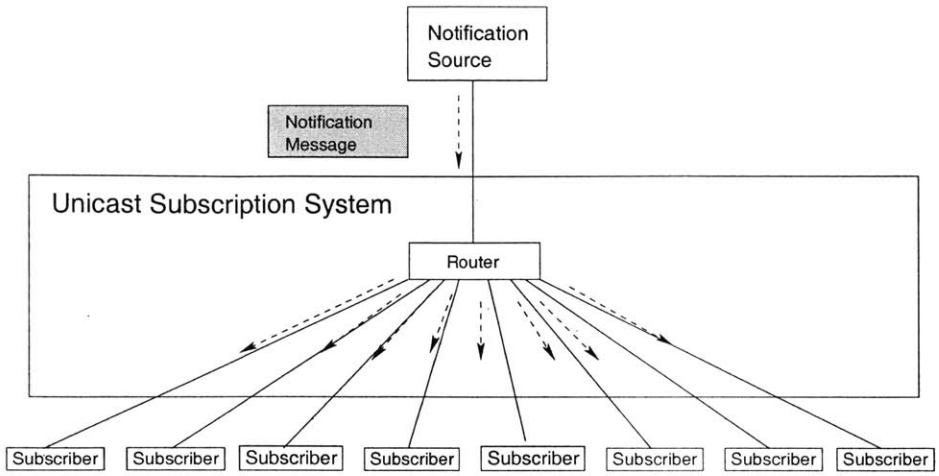


Figure 2-2: Unicast subscription systems. A single router handles all subscriptions and notifications. All messages are sent to and from the router using the existing unicast infrastructure.

should receive copies of particular notifications. When a router receives a notification message, it checks the notification message against entries in its forwarding table. When it finds a match, it receives a list of neighbors to which the message should be forwarded. It then sends a copy of the notification to each of these neighbors. The algorithm that a particular subscription system uses to store subscriptions and match notifications to subscriptions is called its *forwarding algorithm.*

2.4 Evaluation of Current Approaches

2.4.1 Unicast Systems

Unicast subscription systems are commonplace in today's commercial world. Ebay sends email notices to customers who ask to be notified about selected auction items. Amazon sends email notices to customers whose reading preferences are inferred from earlier purchases. Google and the New York Times send email alerts on news topics of interest to subscribers. And interest in unicast subscription systems may be growing. Several research groups are now working to develop general purpose tools to help unicast subscription providers. Relevant tools include HTTP-push systems [47], early Corba Event-Systems [34], Le Subscribe [46], Xfilter [3], Elvin4 [51], Keryx [8], GEM [36], and Yeast [31].

In the unicast approach, subscribers send messages directly to a single information provider, and the provider enters the subscriptions into a central subscription router,

sometimes also called a server or broker. This approach is depicted in Figure 2-2. Publishers send information for publication to the same central router. Upon receiving an item for publication, the provider looks up the topic of the publication in its database and locates the addresses of all subscribers who are interested in the topic. The provider then sends a copy of the publication directly to the Internet address of each subscriber. All of these transactions are handled on the Internet's existing unicast structure.

A recent proposal is to use virtual servers, composed of a cluster of individual servers, to handle unicast distribution of notifications. TIBCO and JEDI are examples of unicast systems that use this approach [59, 16]. The goal here is to reduce network delays that can easily develop with unicast messaging by distributing the workload of forwarding among several servers. Even though TIBCO and JEDI use multiple, distributed servers, these systems should still be considered as unicast approaches because their distributed servers are all located within a local area network. It should also be noted that network delays are not usually a significant concern in local area networks.

Unicast subscription systems are appealing because they use a simple, relatively mature technology. They easily meet two of the three criteria for sound Internet subscription systems. First, unicast services can be highly flexible. Unicast protocols support a wide array of features, such as congestion-control, reliability, and privacy, and when subscription services change and grow, an information provider needs to make changes only on a central server, not on special event routers distributed throughout the Internet. Second, unicast service providers can carry out complex analyses of subscriptions and notification topics in its database. Ensuring that each subscriber receives only one copy of a given notification is a simple operation in most database management systems. Overall, the only factor limiting analyses of subscription requests and flexibility of services in unicast systems is the ingenuity of the software designers who write database management systems for central servers.

Several researchers have noted, however, that unicast systems have one crucial failing [17, 5]. When a single notification must be delivered to a large number of subscribers, these systems can be inefficient and wasteful of resources. A stock-market system that sends real-time quotes to millions of users, for example, can easily clog the network with notification messages. Clogged messages are costly in terms of network bandwidth and delays. Overall, it is hard to envision a unicast system that would efficiently distribute real-time information to millions or hundreds of millions of users. Unicast systems must distribute as many copies of a message as there are subscribers

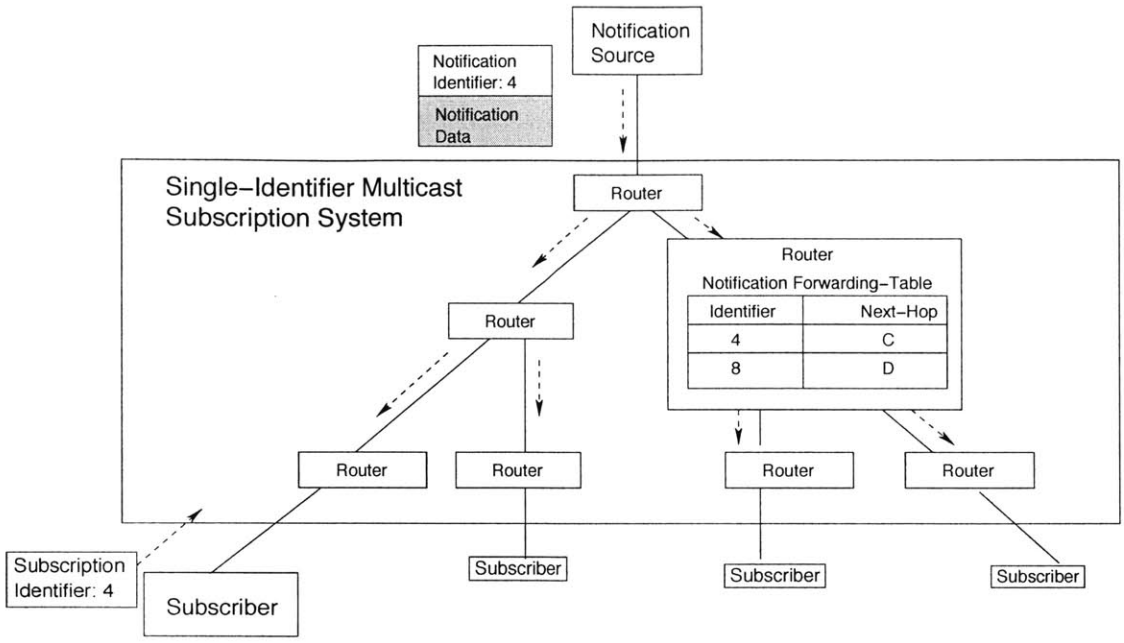


Figure 2-3: Single-identifier forwarding systems. All subscription and notification messages carry a single identifier. Forwarding tables map identifiers to router neighbors.

to the topic of the message. Sending millions of copies of a message through the network seems especially wasteful when one considers that each link in the network need carry only a single copy for the message to reach all subscribers.

2.4.2 Single-Identifier Multicast Systems

Network engineers originally developed IP multicast to support multimedia applications that deliver time-sensitive content to large numbers of subscribers. Internet television and radio are good examples of IP multicast applications. In IP multicast, routers typically set up overlay topologies of dissemination trees to send content from providers to subscribers, as depicted in Figure 2-3. Subscribers interested in a particular multimedia event send their subscriptions to the provider. When the provider is ready to broadcast the event to subscribers, it addresses each packet to a single IP multicast address, also called a channel. Upon receiving a packet, a router looks up the multicast address in its forwarding table and then forwards the packet based on the information it finds. Eventually the content reaches all appropriate subscribers. Theoretically, an IP multicast system can send only a single copy of a publication over any link in a network while disseminating the publication to millions of users.

Several Internet Subscription Systems have been proposed that build on the IP multicast model of communication. Examples of such systems include TIBCO [59], early versions of the Gryphon system [41], Herald [9], i3 [55], and Scribe [49]. These systems differ in many respects, including whether they operate in the Internet or in an overlay network or whether they use peer-to-peer networking in their topologies. They are all classified as single-identifier forwarding systems based on one trait they have in common. In each system, each notification carries a single, numeric identifier, and routers use these identifiers to forward messages.

Single-identifier subscription systems are able to support two kinds of interfaces: subject-based (or topic-based) and content-based interfaces. In a system that offers a subject-based interface, subscribers request information on a single subject, such as "stocks." Providers also place each notification into a single category, and a subscription matches a notification when the two subjects match. A subject-based interface can be built on top of a single-identifier forwarding system by mapping an individual subject onto a unique multicast identifier.

With content-based interfaces, subscribers request notifications based on their content, often in the form of attribute-value pairs, such as "Type=Baseball," "Game=Cubs vs. Marlins, Date=October 20." A subscription matches a notification when the content of the notification matches the content of the subscription. The advantage of a content-based interface over a subject-based interface is its fine-grained filtering of notifications. Although some researchers originally argued that single-identifier systems cannot support content-based interfaces, researchers at IBM have shown that it is possible [41]. The basic idea behind IBM's content-based system is that it assigns each content-based subscription to a single-identifier channel. When a subscriber submits a content-based subscription to this system, it instead submits a subscription for the identifier corresponding to the original subscription. Likewise, when a notification source sends out a notification, it sends the notification to each of the channels that match the notification.

Single-identifier multicast systems also meet two out of the three criteria for sound Internet subscription systems. First, single-identifier multicast systems offer users and providers a good deal of flexibility. These systems can disseminate data in virtually any format, and several algorithms are available for disseminating messages securely and reliably with these systems [21, 29, 37, 26]. In addition, changes in message format are not a problem in single-identifier multicast systems. Providers can make such changes without modifying network routers. Second, single-identifier multicast systems operate efficiently. The cost of forwarding a message in such systems is

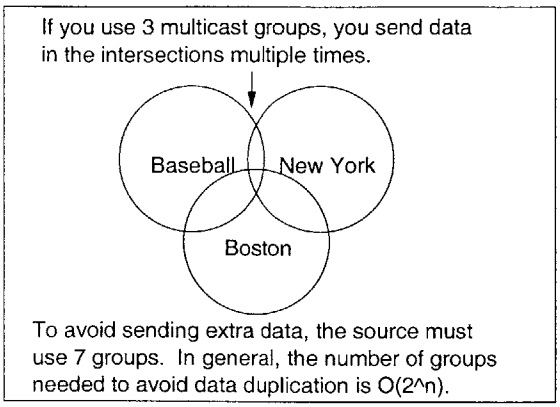


Figure 2-4: The IP Multicast channelization problem.

comparable to the cost of forwarding a message in a unicast system. In order to retrieve the list of interfaces to which a notification should be forwarded, a multicast router simply needs to look up a fixed-length integer address in a hash table.

Single-identifier multicast systems have problems, however, handling subscriptions on overlapping topics. The problem is often referred to as the IP multicast channelization problem. It occurs when subscribers request information from overlapping subscription categories, as illustrated in Figure 2-4. For example, three subscription categories currently available from the New York Times News Tracker are: "Baseball," "New York," and "Boston." If the News Tracker were to disseminate notifications for these categories using only three multicast identifiers, subscribers with interests in two or more of the categories would receive multiple notices when a news story was relevant to all three categories. Note that this problem exists in both subject-based and content-based versions of single-identifier multicast systems.

The News Tracker could solve this problem by assigning a separate multicast identifier to each of the seven disjoint sub-categories covered by the three larger categories. To eliminate all duplication, however, applications would need $O(2^n)$ multicast identifiers to disseminate n categories of notifications. Moreover, if the news tracker added another overlapping category, for example "Baseball Trades," subscribers would have to change their existing subscriptions to a new set of disjoint sub-categories. The News Tracker could try to optimize its use of multicast identifiers by assigning identifiers to only the most active, stable news categories. Unfortunately, this optimization problem has been shown to be NP-hard [1].

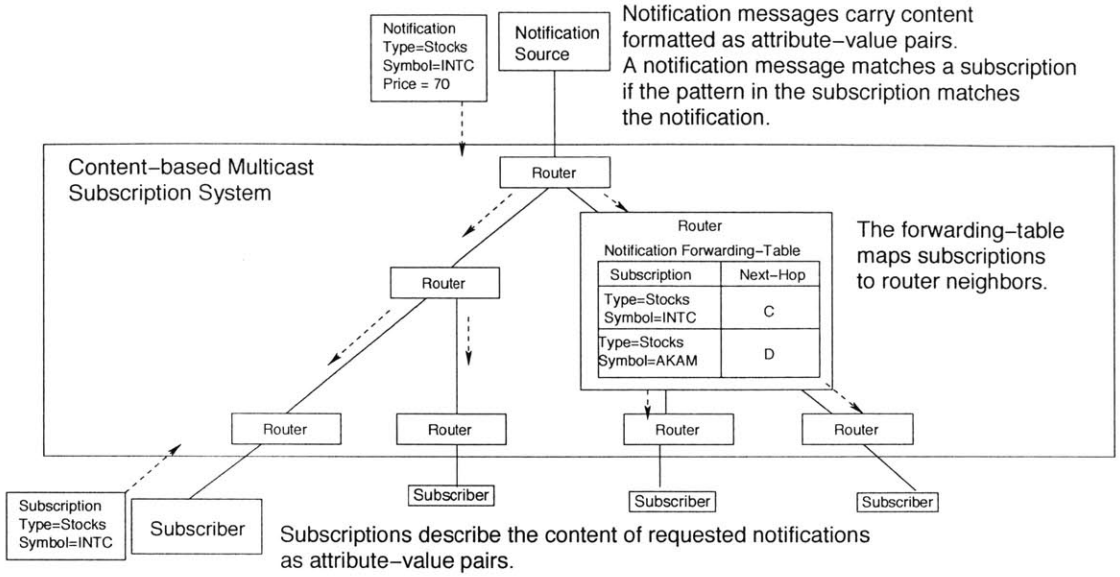


Figure 2-5: Content-based multicast subscription systems. Subscriptions and notification messages are formatted as attribute-value pairs. Routers use pattern-matching to forward notifications to the appropriate subscribers.

2.4.3 Content-Based Multicast Systems

Content-based multicast systems have recently become a hot topic in network research [11, 14, 45, 44, 39]. These systems resemble single-identifier multicast systems in many respects. Content-based and single-identifier systems, for example, use the same approaches to topology-formation and routing. Both systems are able to support a content-based interface, where subscriptions and notifications are expressed as attribute-value pairs. Content-based multicast systems, however, were designed to disseminate text documents, whereas single-identifier multicast systems were designed originally to disseminate multimedia information. Content-based systems therefore forward messages based on message text, rather than relying on a single-identifier that must be attached to each message.

Figure 2-5 depicts a typical content-based multicast architecture. Applications interact with a content-based system through a content-based interface. Unlike a single-identifier system, where content-based subscriptions and notifications must be mapped to individual identifiers before entering the network, in a content-based system, these messages are passed to routers unmodified. When a content-based router receives a notification, it checks the attributes and values of the notification against the subscriptions it has received, and it forwards the notification to the appropriate subscribers based on the results, as depicted in Figure 2-6. Current content-based

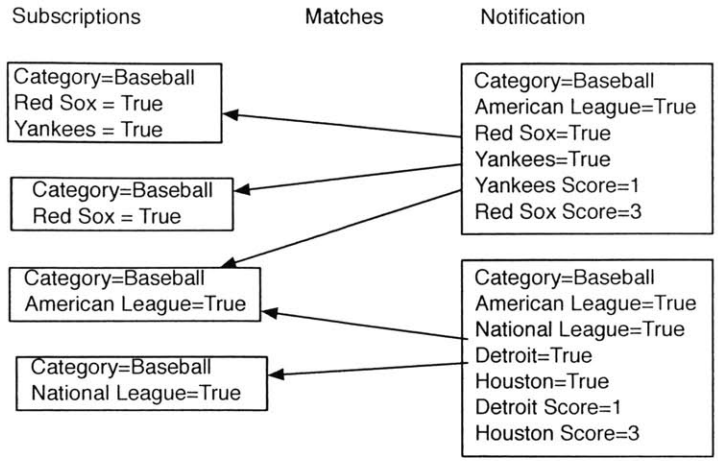


Figure 2-6: Content-based subscriptions and notifications formatted as attribute-value pairs.

forwarding systems include SIENA [11], Gryphon [14], PreCache [45] Hermes [44], and Rebecca [39].

Content-based forwarding systems get high marks on one of the three criteria laid out in the beginning of this section. These systems provide strong support for analysis of relationships between subscription and notification topics. By imposing structure on messages, content-based systems make it possible for routers to match complex, overlapping subscriptions with notifications while avoiding the IP multicast channelization problem. But content-based forwarding systems are less successful in the areas of flexibility and efficiency.

Flexibility

Reliable protocols. One weakness of content-based forwarding systems is the problem of implementing reliable, end-to-end protocols in such systems. In a typical reliable protocol, message senders attach sequence numbers to each message that they send out. Receivers then keep track of the message sequence numbers that they have received. If they notice that a particular sequence number is missing, they flag the message as lost and attempt to recover the data.

Unfortunately, this technique does not work in a content-based forwarding system. Figure 2-7 illustrates the problem. In this figure, there are two subscribers, one subscribed to Red Sox related news and one subscribed to all baseball news. The sender sends out three messages, one about a Red Sox game, one about a Cubs game,

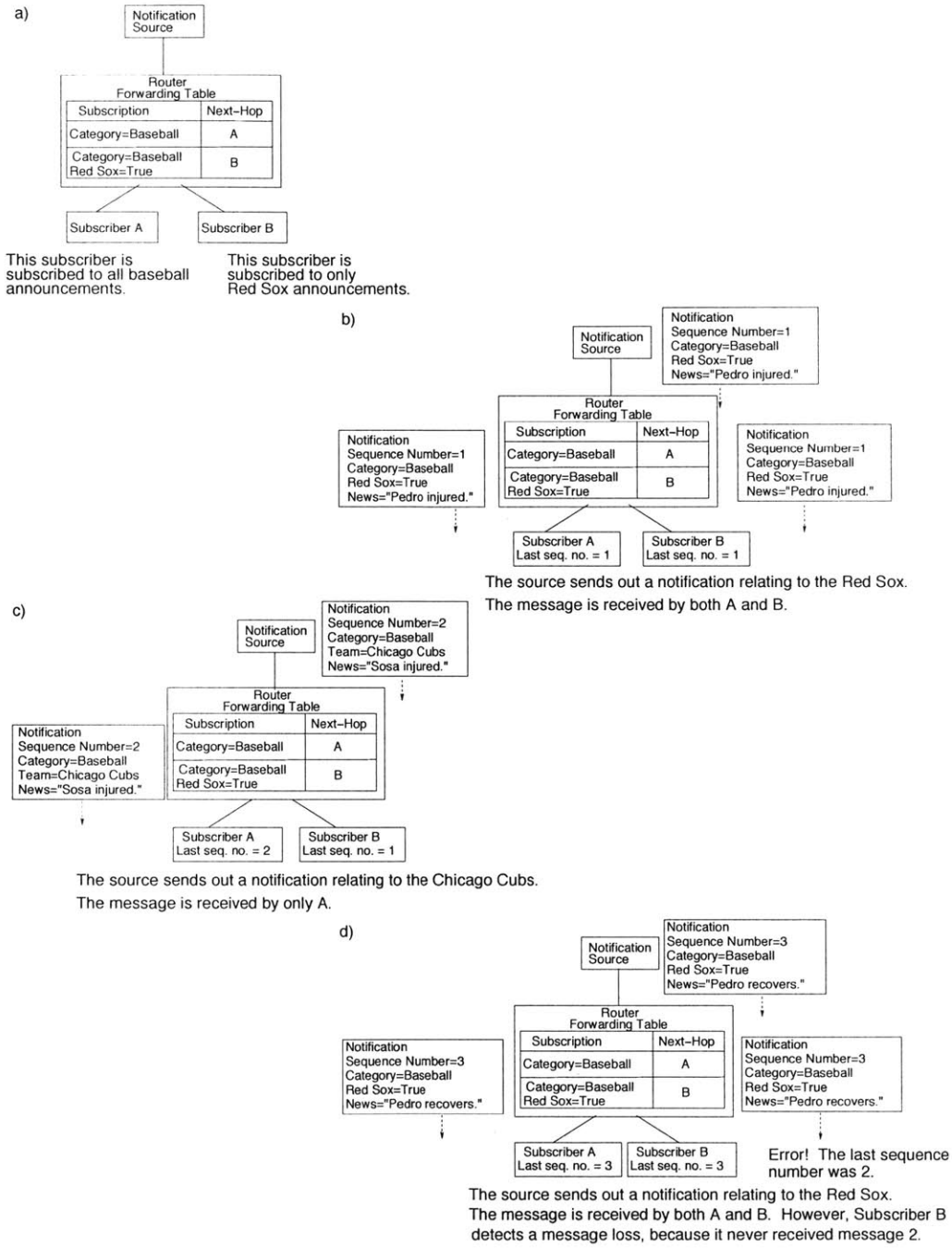


Figure 2-7: The problem with message loss detection in content-based forwarding systems.

and another message relating to the Red Sox. It assigns each of these messages a sequence number. The system correctly does not send the Cubs message to the Red Sox subscriber. However, the Red Sox subscriber detects the message as a loss because it is missing a sequence number.

The question of how to perform loss detection in these systems has not been previously addressed in the literature. A system that used reliable links between routers would only partially solve this problem. Though the links in the network would not lose messages, the routers themselves could lose messages due to failures or changes in the topology of the network. In general, the problem with end-to-end loss detection in a content-based routing system is that it is difficult for subscribers to differentiate between lost messages and messages that have been filtered out by routers.

New Services. Flexibility also emerges as an issue when subscription providers wish to modify and change their services. For example, current content-based routing systems can only match attribute-values for a few types, such as strings and integers. Subscribers to a network monitoring system may require information about IP networks, however. Augmenting a content-based system to match IP addresses with subnet masks would entail adding new machinery to every router in the subscription system, even though only one set of applications required it.

Other examples of application-level features that one might want to add to a content-based routing system are message encryption and compression. If message contents are encrypted or compressed, then every single router in the subscription system must be able to decrypt and decompress the message in order to forward it [48]. Aside from the obvious privacy concerns this approach raises, it adds significantly to the complexity of such a system. In general, the cost of maintaining content-based systems will be high because application-level features must be replicated on all the routers at which forwarding decisions are made. Unless the majority of applications requested a specific new feature, it is unlikely that the extra machinery for supporting the feature would ever be deployed.

Efficiency

Formatting costs. Content-based multicast systems are well-suited to applications that deal with data already structured using text-based, attribute-value pairs, but many applications do not handle such data. To use these systems, providers would

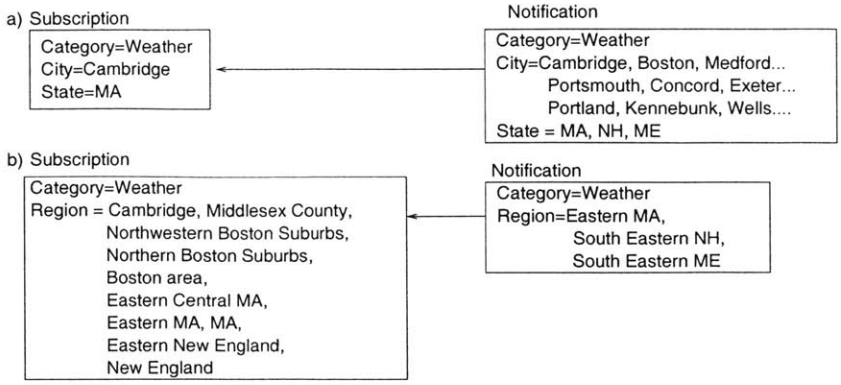


Figure 2-8: The attribute-value format can lead to long messages. To the left, a subscriber subscribes to weather alerts for Cambridge, MA. To the right, a notifier sends out a weather alert for parts of New England.

have to impose an attribute-value, text-based structure on their data, which can produce long, costly messages as a result. Figure 2-8 illustrates the problem. In this example, a subscriber requests weather alerts for Cambridge, MA, and the weather service issues a storm alert for parts of New England, including Cambridge. Part (a) and part (b) of Figure 2-8 illustrate two ways in which subscribers and the weather service can format their messages using attribute-value pairs. If each subscription specifies the name of a single city of interest, then the notification must list all cities covered by the storm, as illustrated in (a). Alternately, if the notification lists only the regions affected by the storm, then the subscription must list all possible regions containing the city of interest, as illustrated in (b).

The length of these unwieldy messages can affect system performance. First, these long messages take up a significant amount of space in notification forwarding tables. Second, as Gruber and his colleagues have shown, the longer the message, the longer it will take a content-based router to process the message [24].

Application-level processing costs. Compared to single-identifier multicast routers, content-based routers spend a significant amount of time performing application-level tasks. Whenever a content-based router receives a message, it must read the entire message. It must then parse application-level text expressions, check syntax, and possibly support other language features, such as type-checking and regular expression matching. In some cases, content-based routers spend more than half their time performing these application-level tasks [24, 53].

Matching complexity. Perhaps the most troubling aspect of content-based routing systems is the time that it takes to match notifications with subscriptions in these systems. Studies have shown that the amount of time that it takes many content-based routers to match a subscription with a notification grows linearly with the number of subscriptions it has received [43, 3, 6, 12, 2]. In other words, the more subscriptions the router has received, the longer it takes to process each notification.

One of the main reasons for using multicast systems over unicast systems is to reduce delays caused by excessive network traffic. Content-based multicast systems do not require excessive network traffic. If content-based routers take much more time to process notifications, however, one might question whether there is any cost advantage in using content-based multicast. For example, it takes some content-based multicast systems 100 milliseconds to process a single message. An equivalent unicast system might have to process 100 such messages, but it would take less than a millisecond to process them all. In this case, the unicast system is not only faster, but it requires no new infrastructure to deploy. The problem of notification processing time is sufficiently challenging that the development of efficient matching algorithms for content-based systems has now become a hot topic of research [2, 46, 6, 43, 3, 12].

2.4.4 Summary

Though it is clear that many applications need to disseminate time-critical data to large numbers of users, it is not clear that either single-identifier or content-based multicast adequately achieve this goal. Single-identifier addresses do not seem rich enough to express the complex relationships that exist between real-world subscription categories. Content-based multicast systems, which support application-level features at the router-level, are able to handle complex subscription relationships. However, content-based systems must sacrifice features such as multiple data formats, privacy, evolvability, and efficiency.

Chapter 3

The F3 Subscription System

F3 is an alternative to the subscription systems described in the preceding chapter. Like content-based routing systems, F3 is a multicast subscription system designed to handle complex subscription requests. Three features, however, distinguish F3 from content-based systems: *subscription-time processing*, which decreases notification processing times; *preprocessors*, which analyze incoming messages and attach routing information to them; and *content graph headers*, which F3 routers use in making forwarding decisions. This chapter gives the rationale behind these three features and also provides a detailed description of F3 architecture.

3.1 Subscription-Time Processing

A major weakness of content-based routing systems, described in the preceding chapter, is the relatively large amount of time that routers take to process notification messages. One way to reduce the time that routers take to process notifications is to perform extra processing steps when subscriptions arrive. For example, current content-based routing systems, such as SIENA, would perform more efficiently if they tested incoming subscriptions for equality. Figure 3-1 depicts results gathered from a SIENA content-based router in an experiment in which different numbers of subscribers signed up to receive messages on the same topic. As this figure illustrates, the time it takes the SIENA router to process a single notification grows linearly with the number of subscriptions it has received, even though all subscriptions are identical. This is because a router adds a record to its forwarding table each time a subscription arrives in SIENA. The router adds the record whether or not the new subscription is identical to subscriptions the router has already received. As a consequence, the size

of the forwarding table grows linearly with the number of subscriptions received. On receipt of a notification, the router must then check the contents of the notification against each record in the table.

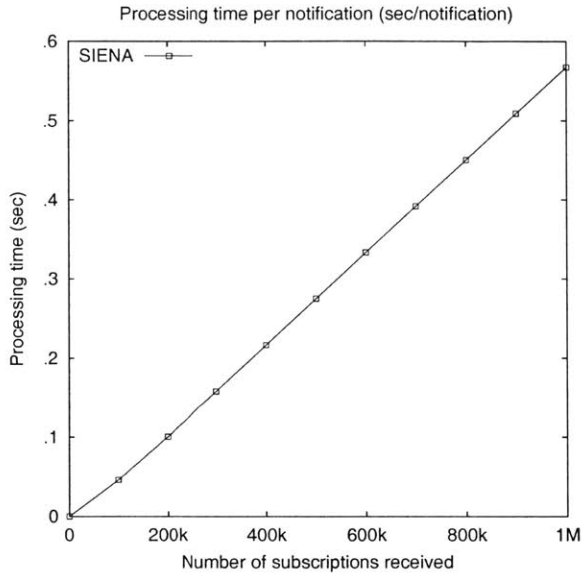


Figure 3-1: SIENA notification time versus number of subscriptions received. In this experiment, all subscribers subscribe to the same notification.

The router would function more efficiently if it tested incoming subscriptions for equality before notifications arrived. In the above example, identity checking would keep the number of entries in the forwarding table to one, no matter how many subscriptions for the same topic were received. The time it would take the router to process subscriptions would increase, because the router would have to check each subscription for equality with previous entries in the table. However, the time it would take the router to process these notifications would remain constant, rather than growing linearly.

Testing subscriptions for equality is just one way to reduce the amount of work that routers do when they receive notifications. A router might also sort attributes within each subscription, sort the subscriptions themselves, or create an index for quickly accessing entries in the table. Gough and Smith have shown that sorting content-based predicates at subscription time can, for example, significantly improve notification processing performance [23].

A system that sorts, analyzes, or indexes subscription requests will process subscriptions more slowly than current content-based routing systems do. However, the

extra time spent in subscription processing will reduce notification processing times. This is because subscribers will usually receive a number of notifications for each subscription request they submit.

3.2 Preprocessing

When a message travels through a content-based network, the network routers perform certain application-level tasks on the message over and over again in order to make routing decisions. The routing process is illustrated in Figure 3-2 (a). When a message enters a content-based system, it is passed to a router. The router parses the message. It then identifies attributes, attribute-types, values, and operators such as =, <, *, and so on. The router next formats this parsed information into an internal data-structure, usually optimized for efficient lookup in a multi-dimensional table. Once the router locates the data structure in its forwarding table, it passes the message along to one or more neighboring routers. The receiving routers then repeat the same process: They parse the message, identify features for forwarding, convert the features to a data-structure, look up the structure in table, and finally forward the message. Only the last two steps are essential router functions. The other steps–text parsing, identifying features for forwarding, and converting message features to data-structures–are application-level steps. Furthermore, unlike the last two forwarding steps, these steps produce the same result at every router along the network path.

An alternative to the content-based approach is depicted in Figure 3-2 (b). Here applications first submit messages to a special preprocessor outside of the network. The preprocessor accepts notification and subscription messages as input, performs all application-level processing steps, and produces formatted forwarding information as output. The preprocessor attaches the forwarding information to the headers of messages, and the formatted information serves as the sole basis for forwarding within the subscription system. Some applications may preprocess messages immediately before submitting them to the subscription system, and others may preprocess messages long in advance.

The idea of using a preprocessor in subscription systems is not a new one. An early version of the Gryphon system [6, 42], developed at IBM, also used a preprocessor to map content-based subscriptions, formatted as attribute-value pairs, to IP multicast identifiers. Because the problem of mapping subscriptions to IP multicast identifiers is inherently NP-hard [1], this early version of Gryphon used heuristics and advance

knowledge of subscription patterns to send notifications to subscribers. As a result, early Gryphon subscribers received false positives, or notifications that they had not requested. The Gryphon system has since moved away from this approach, however, and now uses content-based forwarding to disseminate notifications.

Though Gryphon no longer uses a preprocessor-based approach, such an approach is worth re-evaluating for two important reasons. First, preprocessors make subscription systems more flexible. With preprocessing, subscription services can expand their offerings to include such features as encryption, compression, and new attribute-types without making changes at all network routers. All the necessary changes can be made to a subscription service's preprocessors. Second, preprocessors increase the efficiency of the subscription system. With preprocessors, application-level analyses are completed once before the message begins traveling through the network and do not have to be repeated at every step along the message's path. Routers in the network are left to focus on the simple task of forwarding messages, greatly simplifying their design and making them more efficient. In general, preprocessors push the complexity of processing application-level information to the network endpoints and out of the network itself.

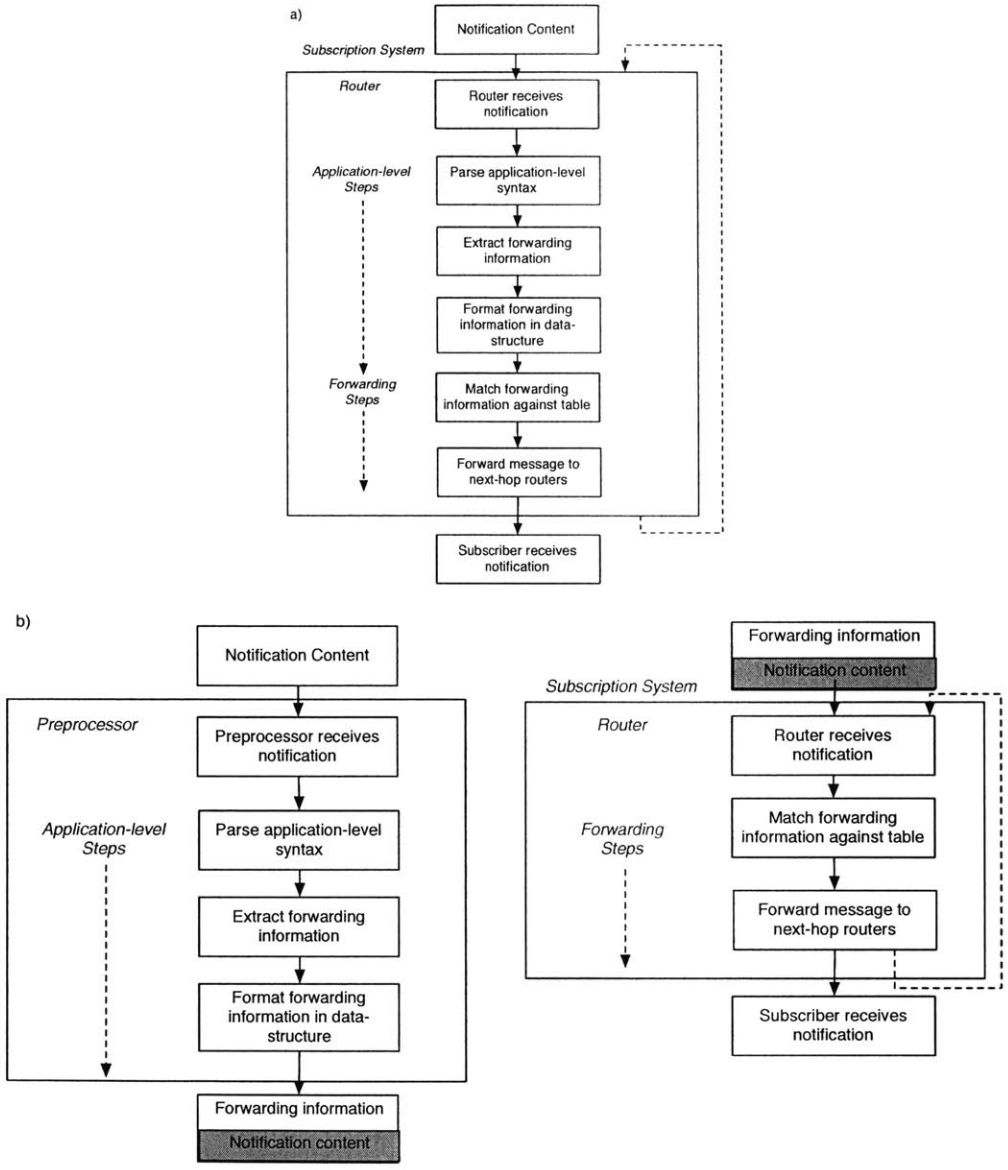


Figure 3-2: Content-based routers perform redundant tasks at every hop in the network (a). Using preprocessors, the subscription system can avoid redundant steps (b).

3.3 Content Graph Headers

How much information do routers need to forward messages to other routers? Designers of subscription systems have given very different answers to this question. Single-identifier multicast systems make forwarding decisions based on the limited information contained in a single address. Content-based systems make forwarding decisions based on their analysis of the full text of a message. Each of these approaches has its shortcomings. The information in single-identifier systems is too limited, and single-identifier systems run into crippling problems with complex, overlapping subscription problems. At the other extreme, content-based systems can put an intolerable burden on network resources because they require extensive analyses of message text at each router. F3 represents a compromise between single-identifier headers, which contain too little information, and content-based messages, which contain too much.

F3 rests on a simple observation about routing: The only information that a router needs to have to forward complex, overlapping notifications is information on overlap among subscriptions it receives. Routers do not need information about the contents of notifications. Instead, routers need to know only whether incoming notifications are covered by subscriptions it has already received. That is, routers do not need to infer from the content of Notification B (e.g., the latest price of Intel stock) that the notice is relevant to Subscription A (e.g., a request for prices of technology stocks). The routers need to know only that the topic of Notification B is relevant to the topic of Subscription A.

The relationships among any set of items can be represented in a content graph, and content graphs can therefore be used to show how topics relate to one another in any subscription system. Formally, a content graph is a directed, acyclic graph, or digraph that represents the partial-ordering between subscriptions. Each node in the graph represents a set of items. Each edge in the graph represents the relationship between two of these sets. Content graphs maintain the following invariant: If the set of items represented by Node A is a superset of the set of streams represented by Node B, then there exists a path in the graph from Node A to Node B.

Figure 3-3 (a) illustrates how a set of baseball topics might be arranged into a partial-ordering. Each node in the graph represents a subscription topic, such as "Category=Baseball" or "Team=Boston." Nodes at the top of the figure represent the most general topics, and nodes at the bottom represent specific categories. An arrow that points from one subscription topic in the figure to another indicates that

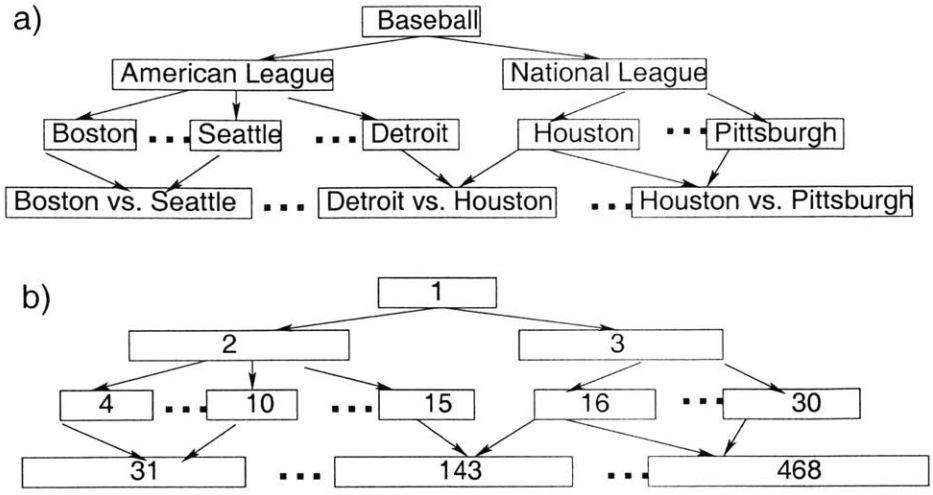


Figure 3-3: A partial-ordering of baseball announcement labels (a) and their corresponding content graph (b).

the first subscription topic covers a superset of notifications of the other. Figure 3-3 (b) depicts a content graph of the announcements. The application-level information about each topic has been removed and replaced with a unique identifier. The identifiers themselves are not meaningful; they can be rearranged or replaced with any other set of identifiers. All that matters is that the relationships among topics are preserved. Routers can make correct decisions about routing from this information about interrelationships alone.

A subscription system that uses content graph headers can support subscription services with a variety of application-level interfaces. These include:

- Content-based interfaces. As the example depicted in Figure 3-4 suggestions, preprocessors can use a simple set of rules to can automatically order attribute-value pairs [10], and this ordering can be used to sort subscriptions into content graphs.

- Numeric range expressions. Numeric range expressions can be partially ordered into a graph.

- Keyword ontologies. As the example depicted in Figure 3-3 illustrates, a hierarchical categorization of message labels can be directly translated into a content graph.

- Object-oriented interfaces. Using an object-oriented interface, preprocessors

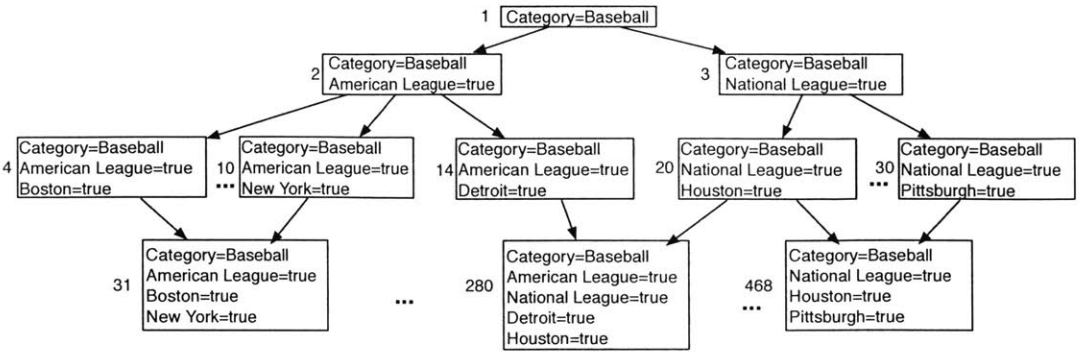


Figure 3-4: A partial-ordering of baseball subscriptions expressed as attribute-value pairs.

would assign nodes in a content graph to classes in an object-oriented hierarchy [19, 20]. Programs would then use the hierarchy to classify each message before submitting the message to the network.

- Application-defined interfaces. Applications can define their own functions for sorting subscription data. For example, an application could define a function to sort subscriptions containing calendar dates or geographic data.

There are some subscription relationships that cannot be represented with content graph edges. For example, graph edges do not represent partially-overlapping relationships between subscriptions. Furthermore, if the relationships between subscriptions change dynamically–sometimes one subscription covers another subscription and at other times it does not–then these subscriptions should not be represented as a graph. Nor can content graphs be used to represent the relationships between "active subscriptions", where each subscription is an executable program. In addition, the cost of representing subscription topics as a graph may just be too high in some circumstances. For example, though it is theoretically possible for preprocessors to arrange subscriptions containing regular expressions into a content graph, the computational complexity of doing so may be prohibitive. In all these examples, the content graphs corresponding to subscriptions would be completely flat, without edges to represent static relationships between subscriptions.

One feature of content graphs, however, make them very appealing: routers can use them to process complex notifications efficiently. The next chapter of this thesis provides documentation for the claim that content-graph forwarding systems can handle overlapping notification categories without the high memory or bandwidth requirements of single-identifier systems. It also documents the claim that content-graph systems process messages more efficiently than content-based forwarding systems do.

3.4 F3 System Architecture

Like other multicast subscription systems, F3 consists of an overlay topology of routers, which forward messages on the basis of information stored in forwarding tables. Unlike forwarding tables in other systems, F3 forwarding tables contain content graph data, which they receive from preprocessors. The rest of this section describes elements of F3 architecture in detail.

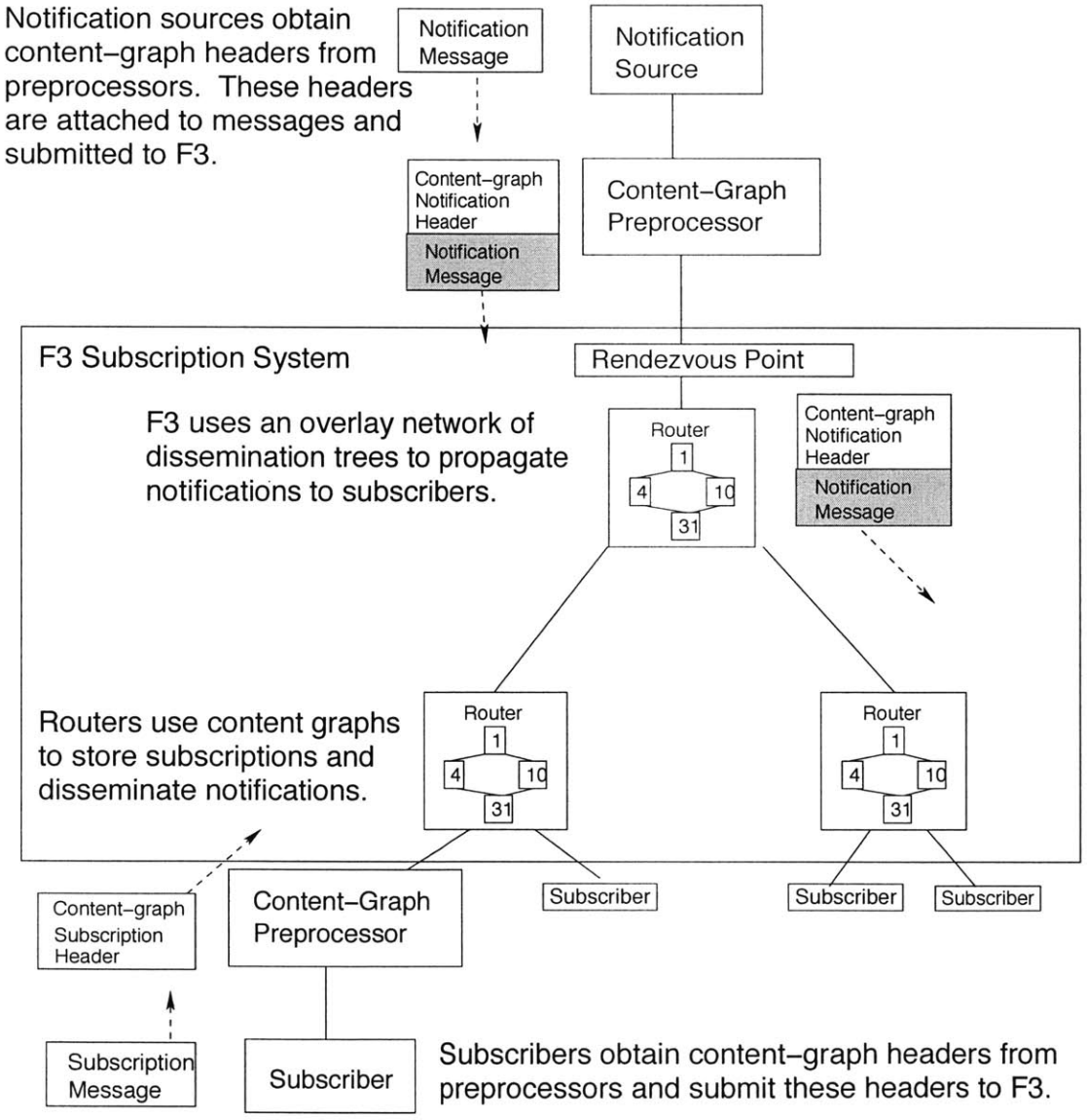


Figure 3-5: F3 system architecture.

3.4.1 Topology

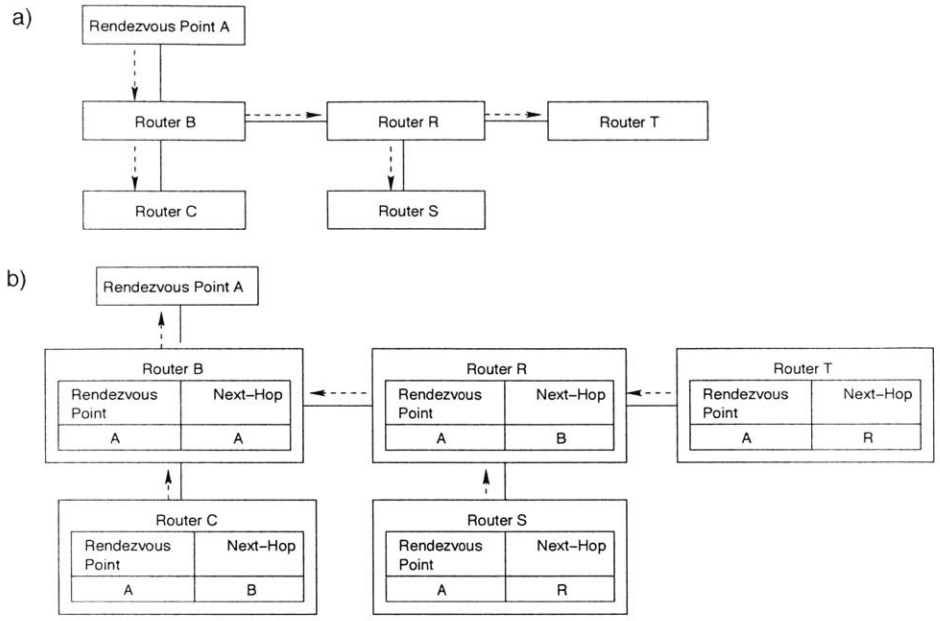


Figure 3-6: F3 dissemination trees. Messages flow downstream, away from the rendezvous point (a). Rendezvous forwarding table entries point upstream, toward the rendezvous point (b).

Like routers in other multicast systems [9, 55, 49, 44], F3 overlay routers are arranged into dissemination trees. Each dissemination tree is rooted at a single node, called a rendezvous point, which is identified by a unique address. Figure 3-6 illustrates a dissemination tree and its place in the F3 architecture. Each router in the overlay network maintains a special forwarding table, called its *rendezvous forwarding table*. The rendezvous forwarding table indicates the next hop on the path to a rendezvous point. In terms of the dissemination tree topology, entries in the rendezvous forwarding table point up the dissemination tree toward the rendezvous point, as illustrated in Figure 3-6. Notification messages flow down the dissemination tree, away from the rendezvous point.

3.4.2 Namespaces

Each rendezvous point in the F3 system is automatically allocated a namespace, and the address of the rendezvous point, also called the *namespace ID*, is used to identify the namespace. The content graph identifiers within a namespace are called *node IDs*.

These node IDs must be unique within the namespace, but two different namespaces may contain the same node IDs. The administrator for a given rendezvous point also manages its namespace. This administrator allocates node IDs to applications, and provides the preprocessors that analyze incoming messages. F3 routers do not handle any of these tasks.

3.4.3 Forwarding

One of the features that sets F3 apart from other systems is its use of content graphs to forward notifications. There are a number of ways to use content graphs in a forwarding algorithm, however. This section first presents a simple approach to forwarding using content graphs. It then presents successive refinements on this basic approach, and finally presents the forwarding algorithm used by F3.

Content-List Forwarding. Content-list forwarding is the simplest way to use content graphs in a subscription system. Figure 3-7 describes the preprocessor algorithm for content-list forwarding. For the purpose of this discussion, we assume that a single, central preprocessor stores the entire content graph for its namespace and handles all preprocessing for the namespace. A later section discusses other ways in which preprocessors may be designed. When a subscription arrives at the preprocessor, the preprocessor finds the node in its graph that exactly matches the subscription. It finds the identifier, or node ID, associated with that subscription in the content graph, and returns this node ID to the subscriber. When the preprocessor receives a notification, it finds all the nodes belonging to the *superset graph* for the notification. The superset graph for the notification consists of all the nodes in the graph that cover the notification. It adds a list of all the node IDs in the superset graph to the header of the message and returns the message to the subscriber. This approach is called content-list forwarding because the preprocessor appends a list to the header of each notification, listing all the content graph identifiers that match the notification.

Figure 3-8 describes the algorithm for setting up subscriptions and forwarding notifications using content lists at a router. In content-list forwarding, routers do not actually store any information about the structure of graphs. When a router receives a subscription from one of its neighbors, it finds the node ID corresponding to the subscription in its local table. In F3, a router's *neighbor* may be a subscriber or it may be another router in the F3 topology. It then makes a note that notifications matching the given node ID should be forwarded to the given neighbor. If the router has

never received a subscription for the node ID before, it also forwards the subscription upstream toward the rendezvous point for that subscription. When a router receives a notification, it finds all of the nodes specified in the header of the message in its table. If the router has received a subscription to any of these nodes from one if its neighbors, the router forwards a single copy of the notification to that neighbor.

Figure 3-9 illustrates how these algorithms work in the context of baseball notifications. In part (a) of the figure, the preprocessor assigns each subscription topic a unique, integer identifier. When a router receives a subscription, the router reads the identifier in the subscription header, and adds to its forwarding table an entry corresponding to that identifier, as depicted in (b) and (c). When a preprocessor receives a notification, as in (d), it creates a header that lists all the nodes in the graph that match the notification. Finally, when a content-list forwarder receives a notification, it checks all of the identifiers listed in the header of each message. If one of its neighbors has subscribed to a topic associated with any of these identifiers, the router forwards a copy of the message to that neighbor, as depicted in (e) and (f). Figure 3-10 further illustrates how these algorithms work in a network with multiple routers.

Content-list forwarding is appealing because of its simplicity. With this approach, routers do not have to maintain any structural information about content graphs–only preprocessors do. The main drawback of content-list forwarding is that content lists may have to contain many identifiers, one identifier for every subscription category that fits a given message. When a message is represented by many identifiers, it will take a correspondingly long time for a router to process the message.

Routers that store graphs. If routers store content graph information, it is possible to reduce notification-processing times by simplifying the headers of notifications that carry many node IDs. A router can determine from a content graph, for example, that an entire list of identifiers is subsumed by a single node ID. When a router receives a notification marked with a particular node ID, the router can infer that all the ancestors of this node also match the notification.

Figure 3-11 describes the algorithm that preprocessors use in this modified approach. When a preprocessor receives a notification, it returns a set of the lowest nodes in the graph that match the subscription. This set maintains the following invariants: a) all the nodes in the set match the notification and b) no node in the set is an ancestor of any other node. This set may contain multiple nodes (e.g. in the case where a notification is covered by multiple subscriptions, and the subscriptions

do not completely cover each other).

PreprocessSubscription(subscription s)
    Let t be the namespace for this preprocessor
    Find the node v in graph G_t that exactly matches s
    Return v and t

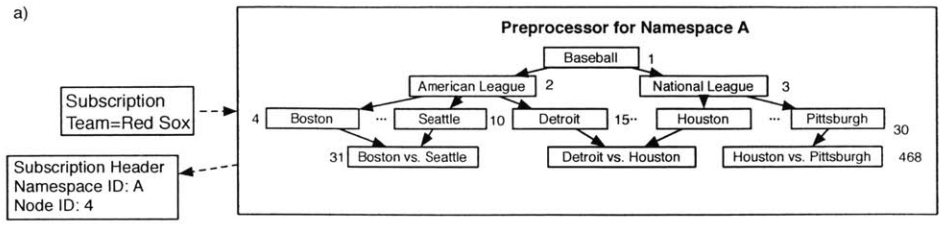
PreprocessNotification(notification n)
    Let t be the namespace for this preprocessor
    Find the set of nodes $V = \{v_1 \ldots v_j\}$ in graph G_t that cover n
    Return V

Figure 3-7: The content-list preprocessing algorithm.

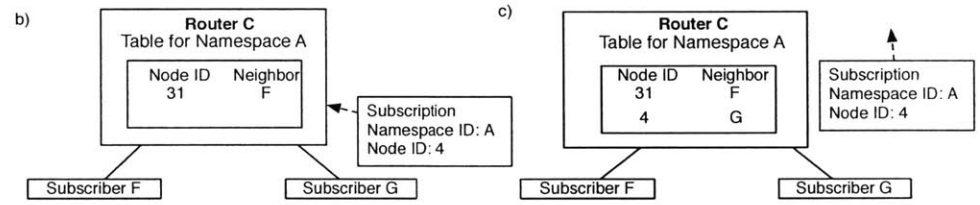
AddSubscription(subscription s, neighbor i)
    Let v be the node indicated in the header of s
    Let t be the namespace indicated in the header of s
    Lookup v in local table
    If v is not found, insert v in table for namespace t
    Add i to subscription list, i_v, for v
    If v's subscription list was previously empty,
        Forward s toward the rendezvous point for namespace t

RouteNotification(notification n)
    Let V be the set of nodes indicated in the header of n
    Let t be the namespace indicated in the header of n
    Let I be an initially empty set of neighbors
    Foreach node v in V
        Add all neighbors in the subscription list for v, i_v, to I
    Foreach neighbor i in I
        Forward a copy of n to neighbor i

Figure 3-8: The content-list forwarding algorithm.

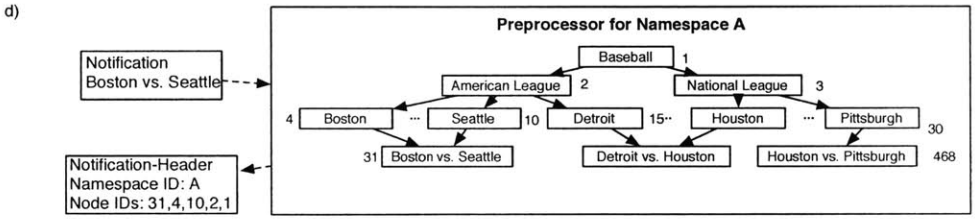


Subscribers first submit subscriptions to preprocessors, which return the node ID corresponding to the subscription.

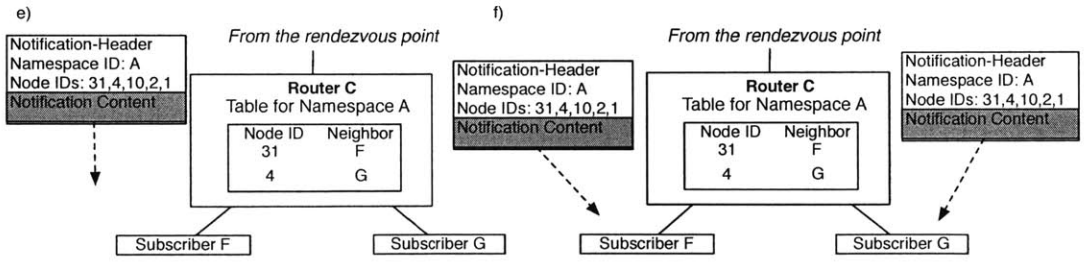


The subscriber submits the preprocessed subscription to a router.

The router updates its forwarding table with the new subscription and forwards the message toward the rendezvous point.



The notification source submits the notification to a preprocessor, which returns a list of all the notification IDs that match the notification.



The notification source submits the notification along with the header to the rendezvous point, and the message is disseminated by routers.

The router looks up all the identifiers in the notification header, and forwards the message to the appropriate subscribers.

Figure 3-9: An example of content-list forwarding.

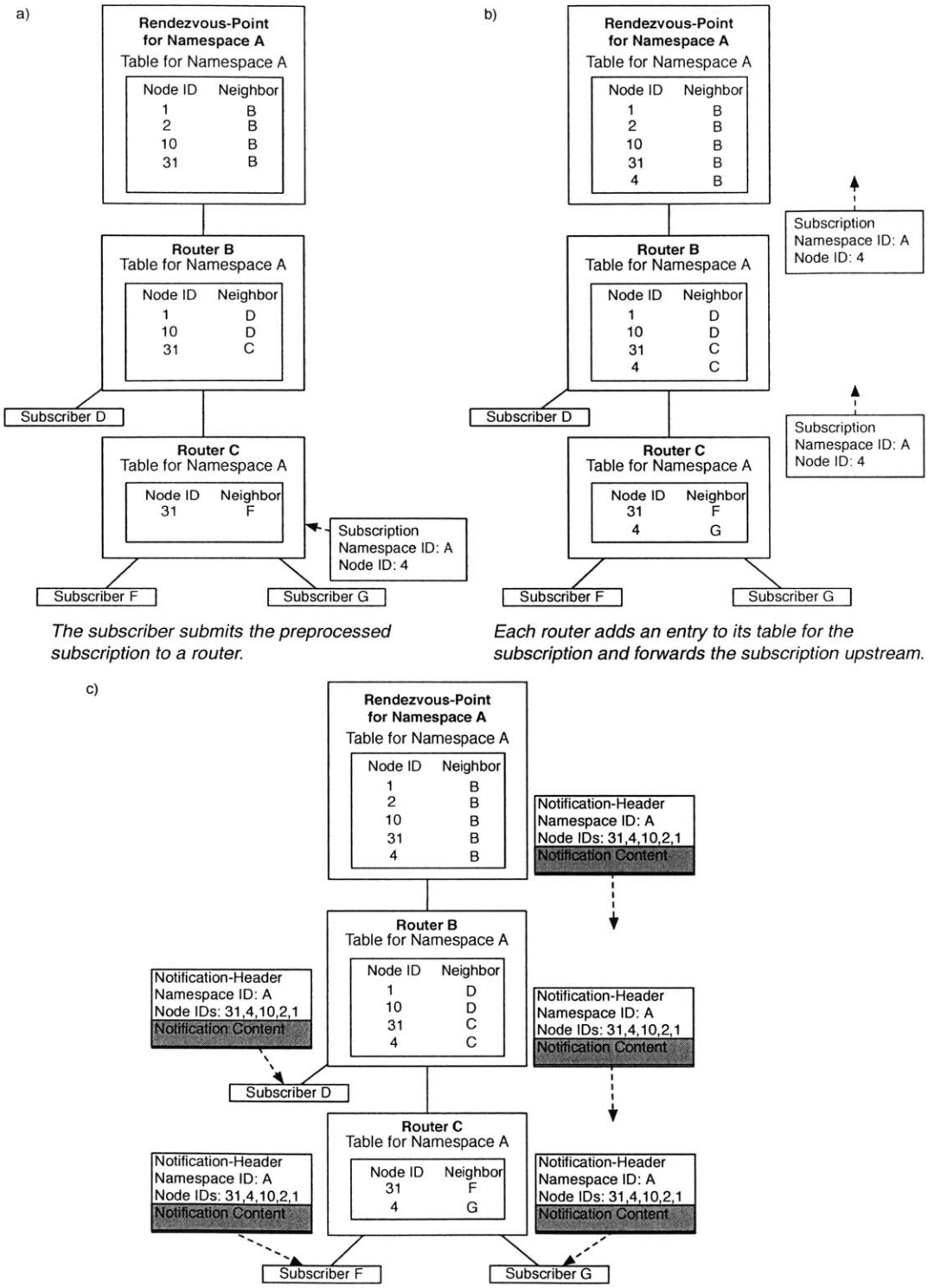


Figure 3-10: An example of content-list forwarding with multiple routers.

Figure 3-12 describes the algorithm for forwarding messages with content graph data. For the purposes of this discussion, it is assumed that every router maintains a copy of the entire content graph. Later sections of this chapter will provide more detail on ways in which routers obtain and store portions of graph data. Routers process subscriptions almost exactly as they do in content list systems. When the router receives a notification in a content graph approach, however, it uses the structure of the graph to forward the notification. Specifically, the router finds the superset graph of all the nodes indicated in the header of the notification. If one of the router's neighbors holds a subscription to a node in the superset graph, the router forwards a copy of the notification to that neighbor.

Figure 3-13 illustrates how this algorithm handles baseball announcements. Preprocessors return a single identifier for each subscription message; the identifier corresponds to the subscription in the content graph, as depicted in (a). When a router receives a subscription message from a neighbor, it looks up the node ID contained in the message, and adds a subscription for that neighbor to its graph, as depicted in (b) and (c). When a preprocessor receives a notification, it creates a header that identifies the lowest node in the graph that matches the notification, as depicted in (d). Finally, when a router receives a notification message, it finds the node in its content graph corresponding to the message. It also traverses the graph to find all of the node's ancestors. If any of the router's neighbors hold a subscription to a topic represented by one of these nodes, the router forwards a copy of the message to that neighbor, as depicted in (e) and (f). Figure 3-14 further illustrates how these algorithms work in a network with multiple routers.

One advantage of this approach over a content-list approach is its shorter notification headers. However, routers that use this approach must store additional information about the structure of graphs. Both approaches take about the same amount of time to process notifications. In both approaches, the amount of work that the router performs when it receives a notification depends on the number of nodes in the graph that match the notification.

```
PreprocessSubscription(subscription s)
    Let t be the namespace for this preprocessor
    Find the node v in graph G_t that exactly matches s
    Return v and t

PreprocessNotification(notification n)
    Let t be the namespace for this preprocessor
    Let V be the lowest nodes in G_t that cover n
    Return V and t
```

Figure 3-11: The preprocessor algorithm for routers that store graphs.

```
AddSubscription(subscription s, neighbor i)
    Let v be the node indicated in the header of s
    Let t be the namespace indicated in the header of s
    Add i to subscription list, i_v, for v in graph G_t
    If subscription list for v was previously empty
        Forward s toward the rendezvous point for namespace t

ForwardNotification(notification n)
    Let V be the set of nodes indicated in the header of n
    Let t be the namespace indicated in the header of n
    Let I be an initially empty set of neighbors
    Find all the ancestor nodes A_V for nodes V in graph G_t
    Foreach node a in A_V + V
        Add all neighbors in the subscription list for a, i_a, to I
    Foreach neighbor i in I
        Forward a copy of n to neighbor i
```

Figure 3-12: The forwarding algorithm for routers that store graphs.

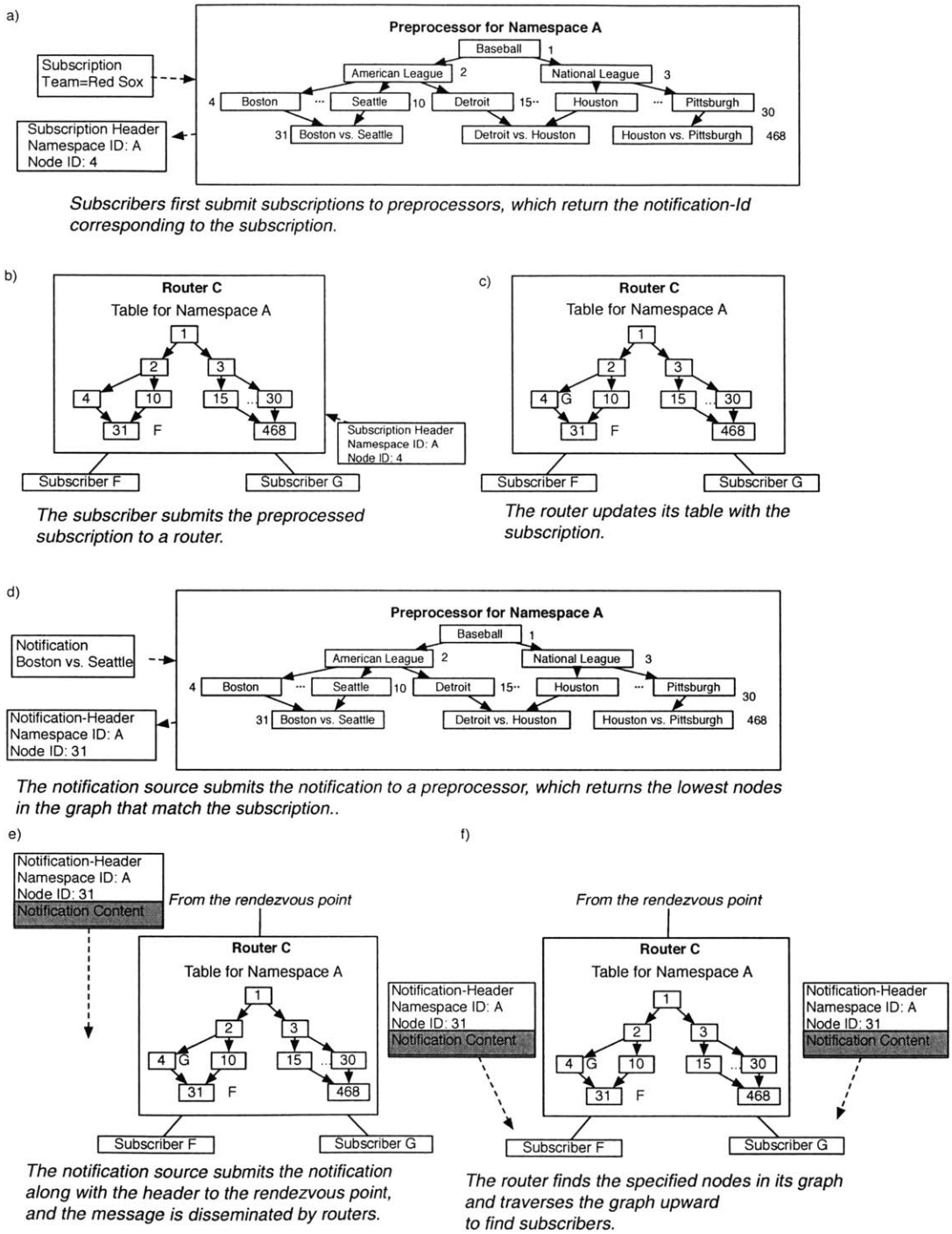


Figure 3-13: An example of a system where routers store graphs.

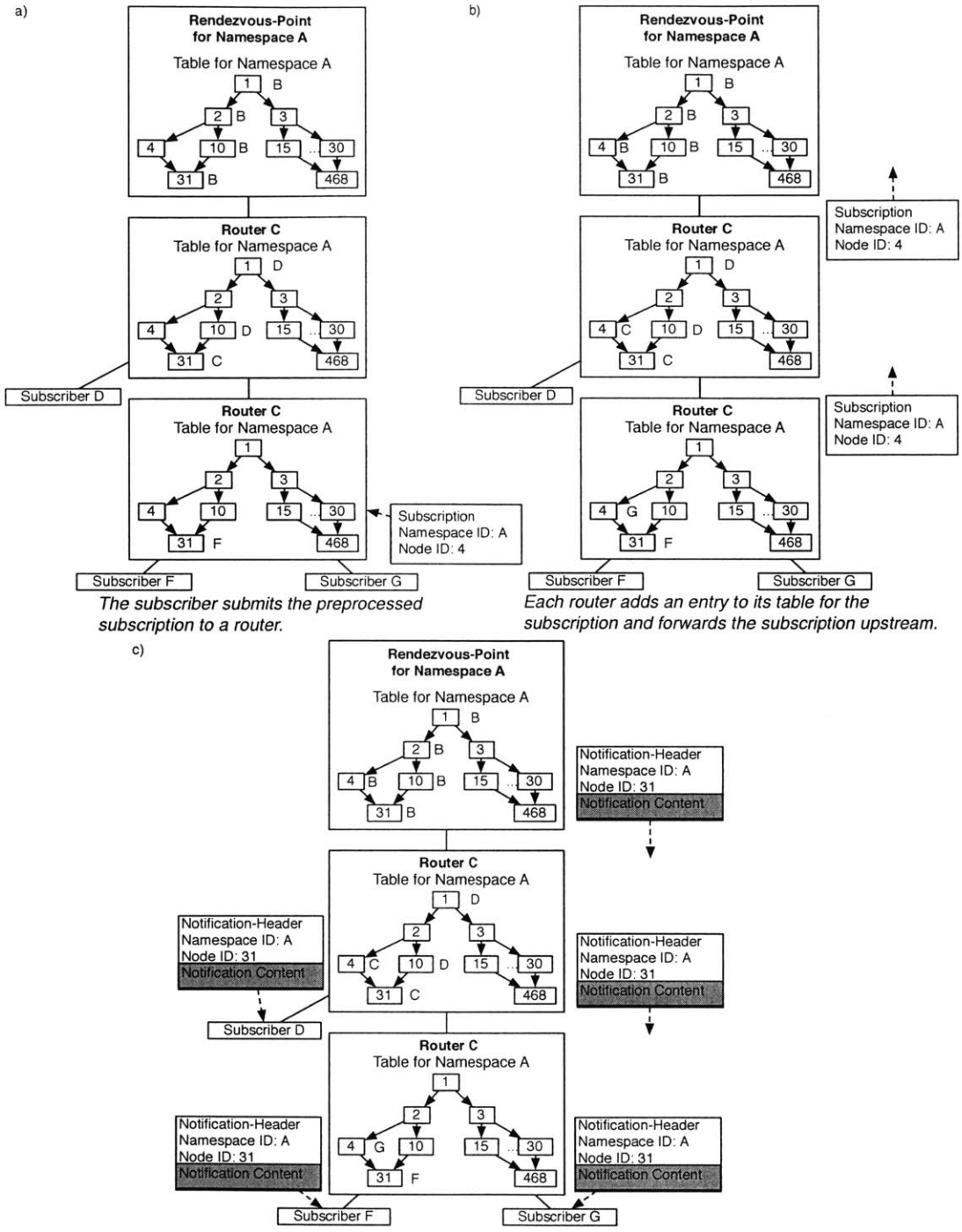


Figure 3-14: An example of a multiple router system where router store graphs.

Minimal graphs. Routers would be more efficient if their forwarding tables contained only information that was necessary for correct forwarding. The router in Figure 3-13 (b) maintains a large table. The table contains information for all the nodes in the content graph, even though the router has received a subscription request for only one of the topics represented in this graph. This not only wastes router storage space, but it slows router performance. The router will waste time processing graph nodes that do not correspond to subscriptions. Routers would be more efficient if they maintained graph information for only those subscriptions that they actually received.

Figure 3-15 illustrates an algorithm for forwarding messages using minimal graph information. The algorithm for preprocessors is the same as before. This algorithm assumes that routers have obtained information about a graph's structure. Specific methods for sending such information to routers will be discussed later. When the router in Figure 3-15 receives a notification, it finds the superset graph for the notification, just as before. It collects a list of neighbors that hold subscriptions to nodes in the superset graph, and it forwards a copy of the notification to those neighbors. When a neighbor does not hold a subscription to one of the nodes listed in the notification header, the router changes the notification header of the message. Specifically, it finds the lowest set of nodes in the superset graph for which the neighbor holds a subscription. The router then changes the header of the notification to specify this set of nodes. If the router did not change the header of the message, the downstream neighbor would receive notifications for node IDs for which it had no subscribers.

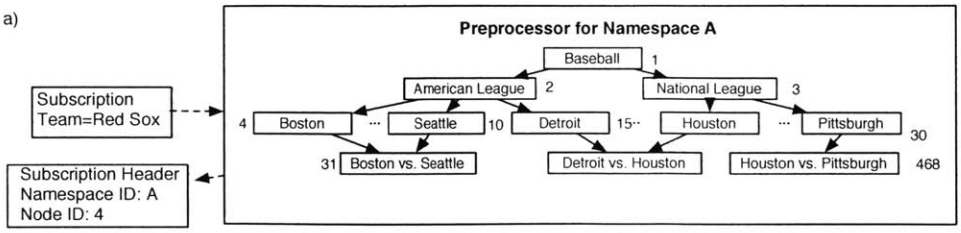
Figure 3-16 illustrates how these algorithms work in the context of baseball subscriptions. In this example, we assume that routers start with the appropriate graph information for their subscription area. When the router receives a subscription, as depicted in (b) and (c), it adds the subscription to its forwarding table and adds graph information to the table corresponding to that node. When the router receives a notification, as depicted in (e) and (f), the router again finds the node in its graph corresponding to the notification. The router forwards a copy of the notification to subscribers of the given node or of any of the node's ancestors. In part (f), the router changes the node ID in the header of the notification. If the router forwarded a message marked with Node ID 31 to Subscriber G in this example, the subscriber would drop the message because it signed up for Node ID 31. The router therefore changes the node ID in the header of the notification to be 4, which corresponds to G's original subscription. Figure 3-17 further illustrates how these algorithms work in a network with multiple routers.

This approach limits the number of routers that must be updated when graph nodes are added to a content graph. When a new content graph node is added to a particular namespace, only the rendezvous point for that namespace must be updated immediately. Routers in the network do not need to be updated with new graph information. This is because routers store graph information only for subscriptions that they have received. Only when a router receives a subscription for a new graph node does it retrieve graph information about that node from the rendezvous point. Section 3.4.6, which discusses how routers receive graph information, elaborates on this topic.

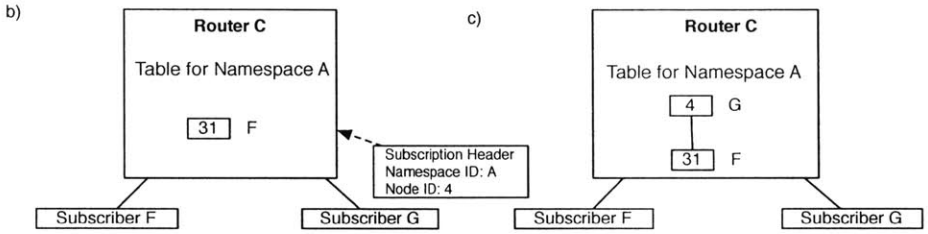
AddSubscription(subscription s, neighbor i)
    Let v be the node indicated in the header of s
    Let t be the namespace indicated in the header of s
    If v is not yet in G_t, get graph information for v and insert v into graph G
    Add i to subscription list, i_v, for v
    If v had no previous subscribers,
        Forward s toward the rendezvous point for namespace t

ForwardNotification(notification n)
    Let V be the set of nodes indicated in the header of n
    Let t be the namespace indicated in the header of n
    Let I be an initially empty set of neighbors
    Let H be an initially empty set of notification headers
    Find the set of ancestor nodes A_V for nodes V in graph G_t
    Foreach node a in $A_V + V$, perform a bottom-up, breadth-first graph-traversal
        Add the subscriber list for a, i_a, to I
        Foreach neighbor i indicated in subscriber list for a, i_a
            Let h_i be the header for neighbor i in H
            If a has no descendants in h_i, add a to the header h_i
    Foreach neighbor i in I
        Forward a copy of n to neighbor i, replacing the header with h_i

Figure 3-15: The forwarding algorithm for routers that minimize graphs.

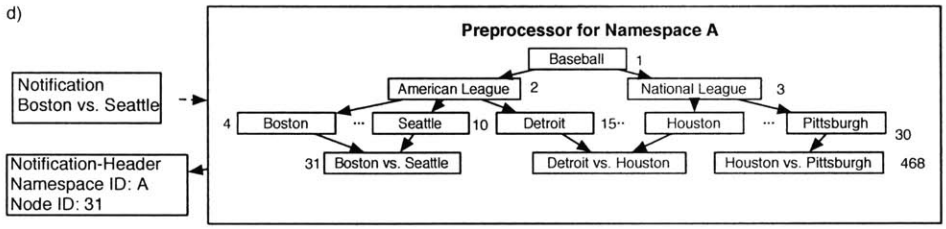


Subscribers first submit subscriptions to preprocessors, which return the notification-Id corresponding to the subscription.

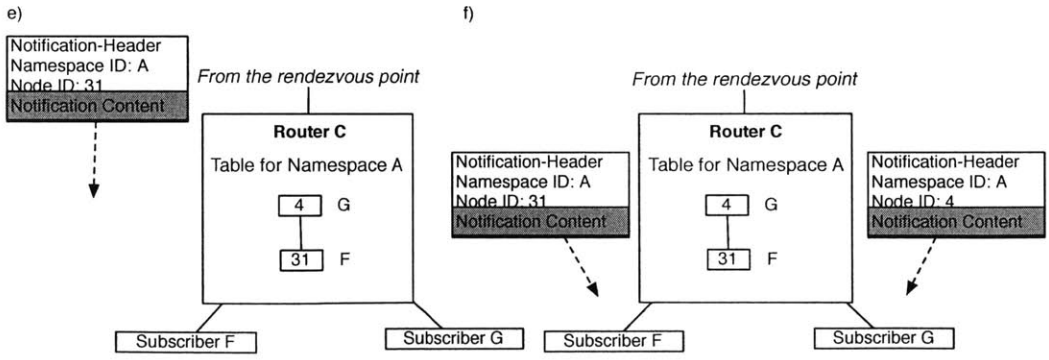


The subscriber submits the preprocessed subscription to a router. The router forwarding table only contains graph information for subscriptions that it has previously received.

The router adds the subscription to its table. It also adds edges to the forwarding table that correspond to the original content graph.



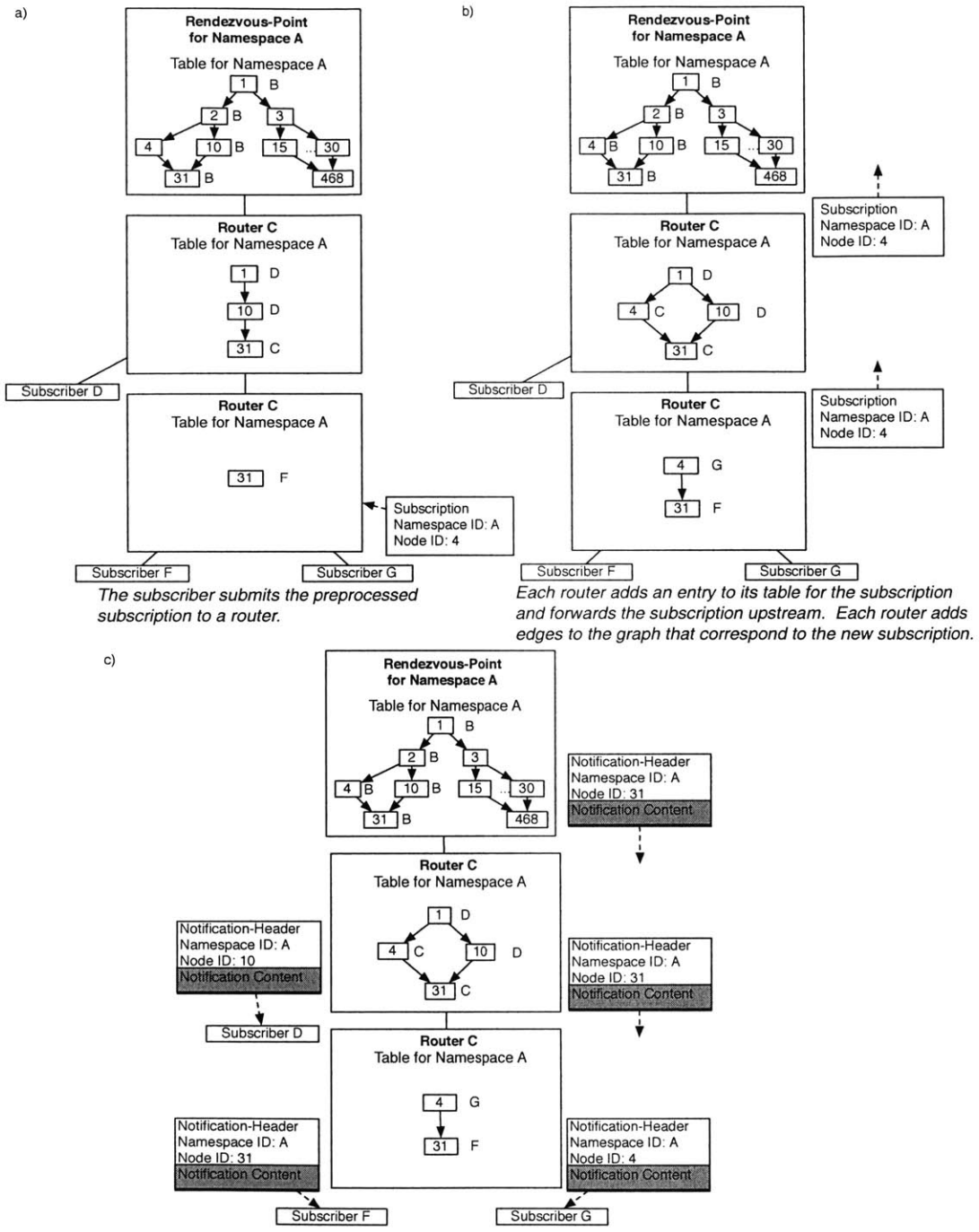
The notification source submits the notification to a preprocessor, which returns the lowest nodes in the graph that match the subscription.



The notification source submits the notification along with the header to the rendezvous point, and the message is disseminated by routers.

The router finds the given node in its graph and traverses the graph upward to find subscribers. The router changes each header to reflect the original subscription

Figure 3-16: An example of a system that minimizes content graphs.



The subscriber submits the preprocessed subscription to a router.

Each router adds an entry to its table for the subscription and forwards the subscription upstream. Each router adds edges to the graph that correspond to the new subscription.

Each router looks up the node IDs in the notification header. It forwards the notification to a neighbor if that neighbor is subscribed to the given nodes or any of their ancestors. It also changes the node IDs listed in the header of each notification to correspond to the original subscription.

Figure 3-17: An example of a multiple router system that minimizes content graphs.

F3 forwarding. Routers can also reduce notification processing times by determining forwarding information when subscription requests arrive at the routers. In the content-graph algorithms described so far, routers find information about superset graphs when they receive notifications for particular sets of node IDs, and they use this information in forwarding the notifications. In the F3 forwarding algorithm, on the other hand, each router pre-computes as much forwarding information as it can when it first receives a subscription.

In the F3 forwarding algorithm, each node in the router's graph is annotated with *output labels*. Each output label contains pre-computed information that the router would normally gather while processing notifications. Each output label indicates two things. First, it indicates which neighbors should receive copies of a notification marked with the given node ID. Second, it indicates what the headers of outgoing notifications should contain. Output labels obey the following rules:

- If neighbor i holds a subscription to node v in the graph, then the output label, $o_{v,i}$, for node v and i is v.
- If neighbor i does not hold a subscription to node v in the graph, then the output label, $o_{v,i}$, specifies the lowest nodes in the supergraph for v for which i holds a subscription.

The first algorithm in Figure 3-18 specifies the steps that F3 routers take to set up subscriptions using output labels. When a router in this figure receives a subscription for a given neighbor, i, it finds the node in its graph that matches the subscription node, v, and adds the subscription to v, as before. Next, the router checks the output labels of v's descendants to see whether they are affected by the new subscription. The router first finds the current output labels for v, $o_{v,i}$ and then finds the set of all of v's descendants, the set D_v. For each descendant, d, in D_v, the router checks the descendant's current output labels, $o_{d,i}$. If none of the nodes listed in $o_{d,i}$ is in v's descendant set, D_v, then the router adds a new output label to d's output labels, $o_{d,i}$, for v and i. It also removes all of v's output labels, $o_{v,i}$, from $o_{d,i}$. Finally, the router changes the output labels for v and i to indicate a single node, v. Figure 3-19 provides an example of how a graph's output labels change as a series of subscriptions are added to the graph.

The second algorithm in Figure 3-18 specifies the steps that F3 routers take to forward notifications. When the router receives a notification, it finds the corresponding output labels for each of the nodes in the header of the notification. Specifically, for each node in the header of the notification, v, and each neighbor, i, the router

finds the output label $o_{v,i}$. It creates a header for neighbor i, h_i, by adding the nodes in $o_{v,i}$ to h_i. Once the router has finished processing all the output labels for the given nodes, the router processes all of the headers that it has created. If the router has a header, h_i, for one of its neighbors, i, it forwards a copy of the notification to neighbor i with the header h_i.

Figure 3-20 illustrates how the F3 algorithm works with baseball announcements. In Figure 3-20 (b), the router starts with a single subscription from Subscriber F for Node ID 31. When the router subsequently receives a second subscription from Subscriber G for Node ID 4, depicted in (c), it adds an output label for Node ID 4 to its graph. This output label indicates that Subscriber G should receive notifications for Node ID 4, and that the header of the notification should be marked with Node ID 4. The router also traverses the descendants of the given node. For each descendant, it checks to see whether an output label for the given subscriber exists and, if not, it creates the output label as before. In this example, the router adds an output label to Node ID 31, indicating that Subscriber G should receive copies of this notification, and that the notification should be marked with Node ID 4.

When the router subsequently receives a notification, as depicted in Figure 3-20 (e) and (f), it finds the given node ID in its content graph. Instead of traversing the graph, the router looks at the output labels annotated on the given node. For each output label, it creates a copy of the notification, modifies the notification header as indicated by the label, and forwards a copy of the notification to the specified subscriber. Figure 3-21 further illustrates how the F3 algorithm works in a network with multiple routers.

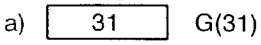
F3 forwarding tables can be slightly larger than forwarding tables in other content-graph forwarding systems because F3 graph nodes are annotated with extra subscription information. F3 routers also process subscriptions somewhat more slowly than other content-graph systems do. The longer subscription-processing times are usually offset by much quicker notification-processing times in F3.

AddSubscription(subscription s, neighbor i)
    Let v be the node indicated in the header of s
    Let t be the namespace indicated in the header of s
    If v is not yet in graph G_t, get graph information for v and insert v into graph G_t
    If i is not on the subscription list for v, i_v, add i
    AdjustDescendantLabelsIfNecessary(v,i,t)
    If v had no previous subscribers
        Forward s toward the rendezvous point for namespace t

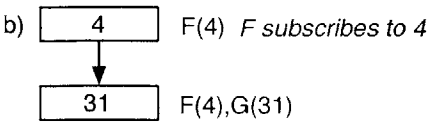
AdjustDescendantLabelsIfNecessary(node v, neighbor i, namespace t)
    Let $o_{v,i}$ be the output labels for node v, neighbor i
    Let D_v be the set of all descendants for node v in graph G_t
    Foreach node d in D_v
        Let $o_{d,i}$ be the output labels for descendant d, neighbor i
        If node of the nodes in $o_{d,i}$ is in the set of v's descendants, D_v
            Subtract $o_{v,i}$ from $o_{d,i}$
            Add v to $o_{d,i}$
    Set $o_{v,i}$ to contain the node v

ForwardNotification(notification n)
    Let V be the set of nodes indicated in the header of n
    Let t be the namespace indicated in the header of n
    Let I be an initially empty set of neighbors
    Let H be an initially empty set of notification headers
    Foreach node v in V
        Add the subscriber list for v, i_v, to I
        Foreach neighbor i indicated in subscriber list for v, i_v
            Let h_i be the notification header for neighbor i
            Let $o_{v,i}$ be the set of output labels for node v and neighbor i
            Add $o_{v,i}$ to h_i
    Foreach neighbor i in I
        Forward a copy of n to neighbor i, replacing the header with h_i

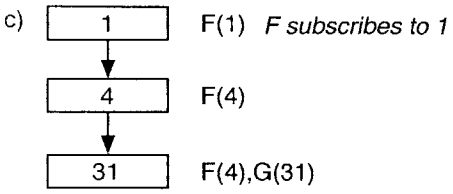
Figure 3-18: The F3 forwarding algorithm.



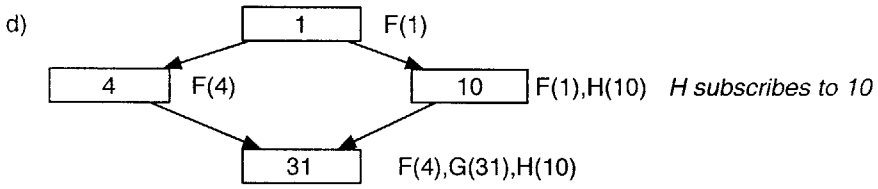
Initially, the graph has one subscription for Neighbor G for node 31.



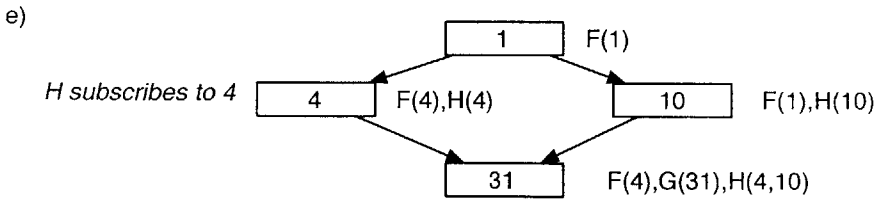
Neighbor F subscribes to node 4. The router adds a label to node 31, indicating that messages matching 31 should be sent to Neighbor F, marked with Node ID 4.



Neighbor F subscribes to node 1. Because all of node 1's descendants already have labels for Neighbor F, the router does not add any labels to these descendants.

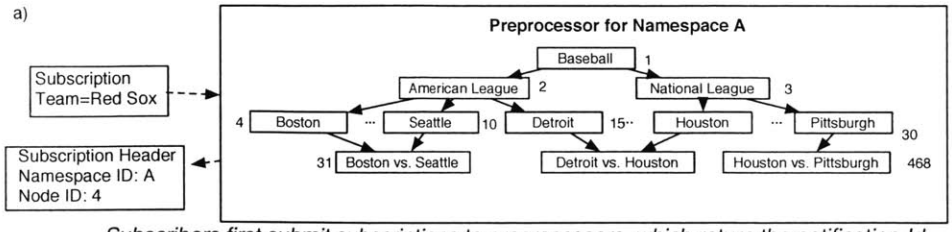


Neighbor H subscribes to node 10. The router first adds node 10 to the graph and updates the labels for the new node by adding a label for Neighbor F with Node ID 1. The router then adds a label to node 10 for H. Finally, the router adds a label to node 31, indicating that messages matching 31 should be sent to Neighbor H, marked with Node ID 10.

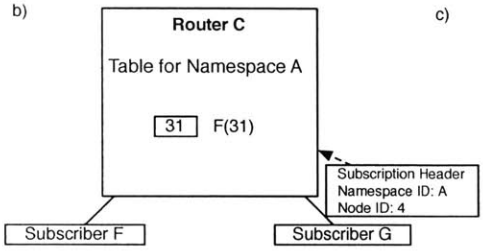


Neighbor H subscribes to node 4. The router adds a label to node 31, indicating that messages matching 31 should be sent to neighbor H, marked with Node ID 4. When the router receives a message marked with Node ID 31, it will send the message to Neighbors F, G, and H. Note that the message that the router sends to Neighbor H will be marked with two node IDs, 4 and 10.

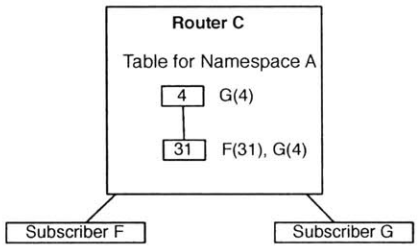
Figure 3-19: How a graph's output labels change as subscriptions are added to the graph.



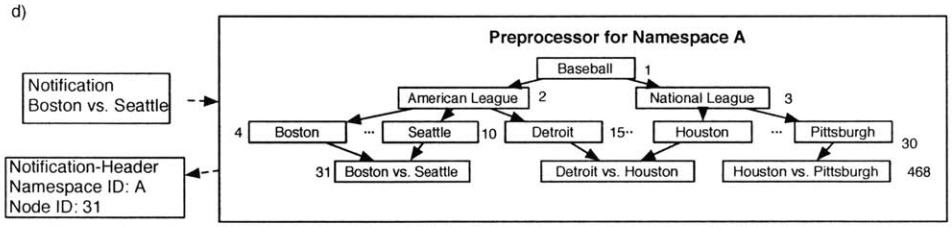
Subscribers first submit subscriptions to preprocessors, which return the notification-Id corresponding to the subscription.



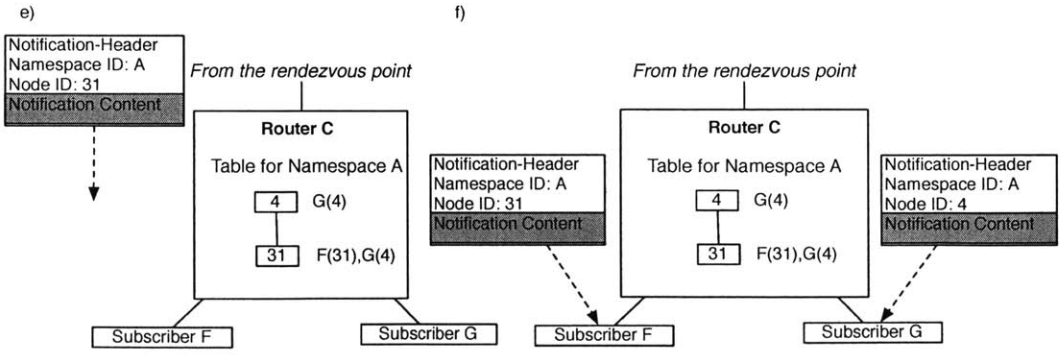
The subscriber submits the preprocessed subscription to a router. The router forwarding table only contains graph information for subscriptions that it has previously received.



The router adds the subscription to its table. It adds edges to the forwarding table corresponding to the original content graph. The router traverses the graph downward, and modifies the output labels of descendants, as necessary. Output-labels indicate which interface a notification should go to and what the header should look like.



The notification source submits the notification to a preprocessor, which returns the lowest nodes in the graph that match the subscription..



The notification source submits the notification along with the header to a router.

The router finds the given node in its graph. it does not traverse the graph. It outputs a copy of the message to the listed subscribers. he router uses the given output-labels to change the header of the notification.

Figure 3-20: An example of F3 forwarding.

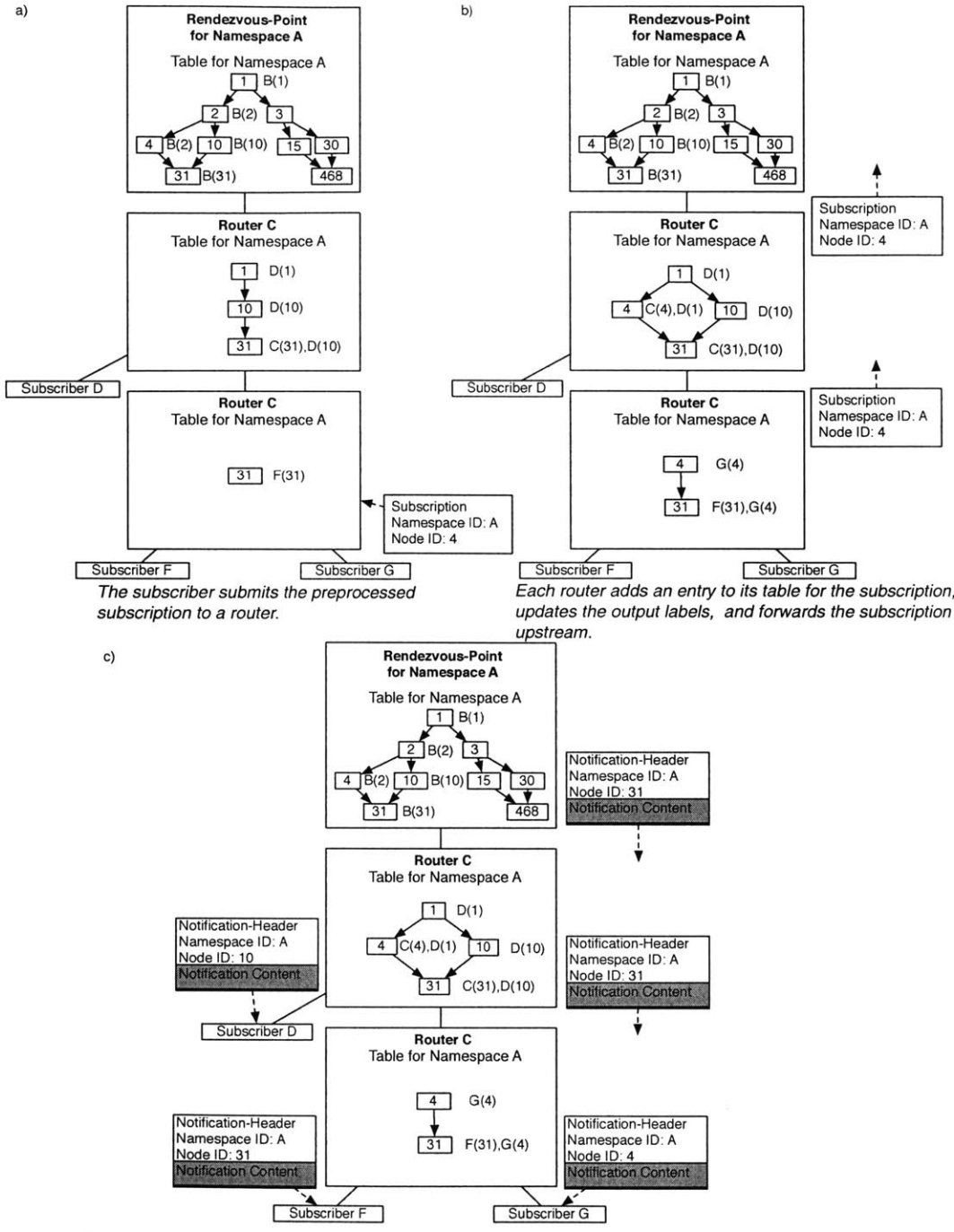


Figure 3-21: An example of a F3 forwarding using multiple routers.

3.4.4 Preprocessors

F3 preprocessors take message content as input and produce content graph node identifiers as output. These preprocessors are not part of the F3 network proper. Administrators for each namespace are responsible for setting up preprocessors to provide identifiers. Administrators are also responsible for maintaining consistency between the content graphs used by preprocessors and the content graphs used at rendezvous points.

Though the specific needs of an application will determine its preprocessor design, there are three aspects of preprocessor design that are worth discussing. The first is the number of physical processors that make up the preprocessor. A centralized preprocessor is a single preprocessor through which all subscriptions and notifications must pass. Centralized preprocessors are appealing because of their simple design. The problem with using a centralized preprocessor is that the preprocessor may become a bottleneck in the system because all messages must pass through the preprocessor. A distributed system uses multiple, distributed processors. These preprocessors may be co-located within a local area network or they may be scattered around the wide area network. In the extreme case, each subscriber and notification source would run a dedicated preprocessor for its namespace on its local machine. The advantage of the distributed approach is that a number of preprocessors share a load that would otherwise fall on a single preprocessor. The disadvantage with multiple preprocessors comes when the structure of the graph is changed for any reason. This is because it is more difficult to coordinate changes to the graph among multiple processors than it is with a single processor.

The second aspect of preprocessor design that is worth discussing is the method of content graph generation. A static preprocessor generates its entire content graph before it receives any subscription messages. Static preprocessors can process messages relatively quickly, because they do not spend time creating graph nodes in response to these messages. There are certain applications that cannot use a static preprocessor. For example, for an attribute-value application with a hundred possible attributes and values, the number of possible subscriptions would be 100^{100}. For such applications, it makes sense to use a dynamic preprocessor. A dynamic preprocessor generates its content graph in response to the subscription messages it receives. For example, in Figure 3-3, if the preprocessor received a new subscription for "Boston OR New York", the preprocessor would insert a new node into the graph ; the node would be a child of "American League" subscriptions and a parent of both "New York" and

"Boston" subscriptions, as depicted in Figure 3-22. The preprocessor would create a new identifier for the subscription and return this identifier to the subscriber. As mentioned previously, the preprocessor must also ensure that changes to the graph are sent to the appropriate rendezvous point in the network. The advantage of such dynamic preprocessors is that they have to maintain content graph nodes only for existing subscriptions. The disadvantage of dynamic preprocessors is that they must potentially sort and insert the subscriptions they receive. They can therefore take more time to process subscriptions than static preprocessors.

It is important to note that a preprocessor can share some combination of the above traits. Some preprocessors may retrieve statically generated content graph identifiers from a locally stored table. A centralized preprocessor may use web pages to elicit dynamic subscription information from subscribers. It is even possible, though more challenging, for a distributed preprocessor to generate graphs dynamically in response to subscription requests. So long as the administrator for the namespace ensures that the preprocessor graphs and the rendezvous point graph are kept consistent, any of these preprocessor designs will work with F3.

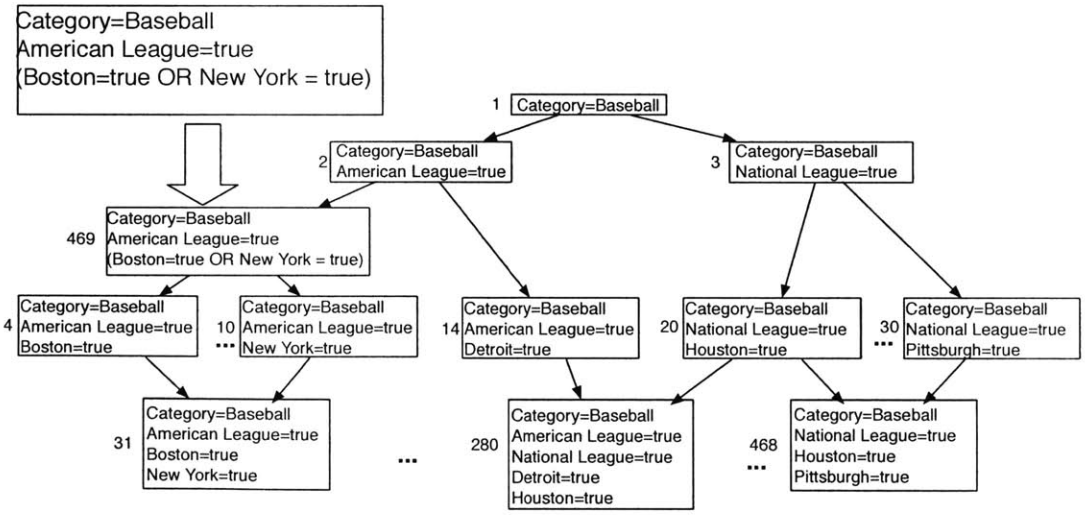


Figure 3-22: Dynamic preprocessors take new subscriptions and insert them into graphs on-the-fly. Here, the preprocessor inserts a new subscription for "Boston or New York" into the content graph.

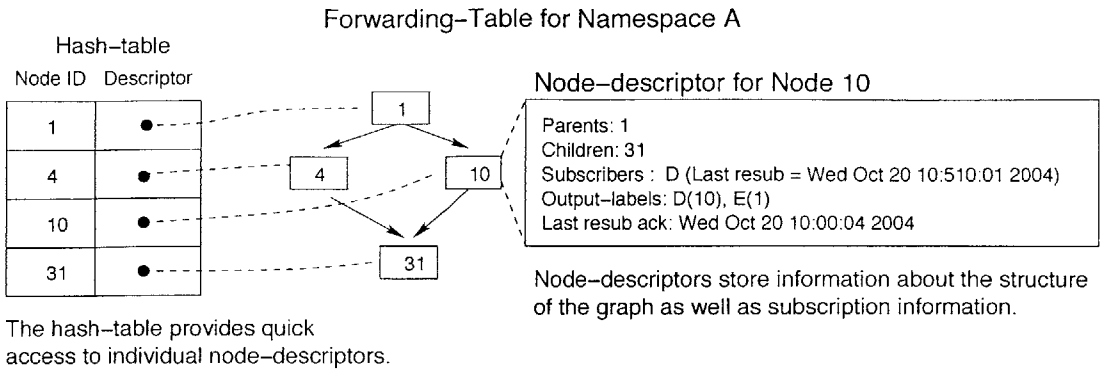


Figure 3-23: An F3 notification forwarding table.

3.4.5 Notification Forwarding Tables

Each F3 router maintains a forwarding table for each namespace. These forwarding tables are perhaps the key element in F3 functioning; they play a central role in almost every F3 activity. Figure 3-23 illustrates a typical table. The central component of the table is a content graph in which nodes are represented by node-descriptors. Each node-descriptor contains several pieces of information:

- Pointers to the parents and children of the node in the content graph. Together, the pointers in all node-descriptors make up the structure of the content graph.
- A list of downstream neighbors requesting notification messages for a given node. Along with each neighbor's name is a timestamp, which gives the last time the router received a resubscription request from the neighbor. This timestamp is used to determine when a downstream neighbor has dropped a subscription.
- A list of output labels for the node. When a router receives a notification for a particular node ID, it uses these output labels to determine which neighbors should receive a copy of the notification. Each output label includes two things: the name of a neighbor to whom the notification should be sent and the node IDs that should be included in the notification header for that neighbor.
- A timestamp that indicates the last time that the router received a resubscription acknowledgment from the router's upstream neighbor. This timestamp is used to determine when an upstream neighbor has dropped a subscription.

The forwarding table also contains a hash-table, as illustrated at the right of Figure 3-23. This hash-table maps node IDs to node-descriptors in the content graph. Routers

use this hash-table to quickly look up node-descriptors in the content graph.

3.4.6 Graph Setup

Each F3 rendezvous point maintains a copy of the content graph for its identifier namespace. These content graphs may be static representations of a subscription topic area, input by the namespace administrator initially and changed only when the administrator undertakes a major revision of the subscription service. Or the content graphs may change and grow as subscribers make new requests for new subscription topics. Whatever methods a namespace administrator uses to construct the content graphs, it is the administrator's responsibility to ensure that an up-to-date copy of the graph is always available at the namespace rendezvous point.

When F3 routers receive subscriptions, they send requests for content graph information upstream toward the relevant rendezvous points. The rendezvous points, in turn, send information about content graphs downstream to routers. The algorithm that routers use to request and disseminate graph information is given in Figure 3-24. When a router receives a subscription request for a node that is not already in its graph, it adds the node to a list of nodes that are pending graph information. It then forwards the subscription to its upstream neighbor, requesting information about the node. If a router receives a subscription that requests graph information from one of its neighbors, and it also has no information about the node, it will propagate the request upstream toward the rendezvous point. It will also make a note that any information it receives should eventually be sent downstream. When the router is ready to send information about the node to one of its neighbors, it finds the appropriate parent and child nodes for that node and neighbor in the graph. The parents of the node are the lowest ancestors of the node to which the neighbor has subscribed. Likewise, the children of the node are the highest descendants of the node to which the neighbor has subscribed. The router then sends a graph-setup message to the neighbor, specifying the parents and children of the node. Upon receiving a graph-setup message, a router adds the node to its graph with edges to the specified parents and children. Finally, it adds any pending subscriptions to the node using the standard procedure for adding subscriptions to the graph.

Figure 3-25 illustrates an example of this process. In Figure 3-25 (a), Router C has only one subscription recorded in its forwarding table at the beginning; the subscription is for messages on the topic marked with Node ID 31 of Rendezvous Point A. Router C then receives a subscription from Subscriber G for messages on

the topic marked with Node ID 4 of this namespace. As depicted in (b), Router C makes a new entry in its forwarding table for Node ID 4 and forwards the subscription upstream. Because C lacks additional information about Node ID 4, it also sets a flag on the forwarded subscription and makes a note in its forwarding table that this information is pending. When Router B receives the subscription, it takes the same steps that Router C took, until the subscription finally reaches Rendezvous Point A.

When Rendezvous Point A receives the subscription, it sends downstream a graph-setup message that contains the requested edge information. This process is depicted in (c). The message first reaches Router B; the message gives this router the parent- and child-edges that it should add to Node ID 4 in its graph. Upon receiving this message, B adds the edge information to its table, updates its output labels for Node ID 4, and marks the node as no longer pending. Router B then sends a message to Router C, indicating the edges that it should add to its graph for Node ID 4. Router C updates its graph and the process is complete.

It is important to note that F3 could use other methods to disseminate graph information to routers. Subscribers could, for example, obtain content graph information from preprocessors and then distribute this information to routers along with their subscription messages. The advantage of this particular approach is clear; when a new node is added to the content graph for a particular namespace, only the rendezvous point for that namespace must be immediately updated. Routers in the network do not have to be updated because they only store graph information for subscriptions that they have previously received. When routers on the path to the rendezvous point receive a subscription for the new graph node, they propagate requests upstream for information about the node's graph, as described above. The rendezvous point then propagates graph information about the node back down the path to the subscriber, and all the routers along the path then update their tables.

```
AddSubscription(subscription s, neighbor i)
    Let v be the node indicated in the header of s
    Let t be the namespace indicated in the header of s
    If v is not yet in graph G_t
        Add v to list of nodes that is pending graph information
        Add a subscription for i to node v
        If the subscription is marked "graph-request"
            Make a note that i is "graph-request-pending" on v
        Send a subscription for v upstream, marked "graph-request"
    else
        If the subscription is marked "graph-request", SendGraphSetup(v,i,t)
        Continue with subscription algorithm as before...

SendGraphSetup(node v, neighbor i, namespace t)
    Let P be the empty set of parent nodes for v
    Let A_v be the set of all ancestors for node v in graph G_t
    Foreach node a in A_v
        Add a to P if i is subscribed to a and a has no descendants in A_v
    Let C be the empty set of child nodes for v
    Let D_v be the set of all descendants for node v in graph G_t
    Foreach node d in D_v
        Add d to C if i is subscribed to c and c has no ancestors in D_v
    Send a Graph-setup message downstream, containing v, t, P, and C

ReceiveGraphSetup(node v, namespace t, parents P, children C)
    Find v in the list of nodes pending graph information
    Add v to graph G_t
    Add P to the parent list for v
    Add C to the child list for c
    For all neighbors i subscribed to v
        AddSubscription(v,i)
        If i is marked "graph-request-pending" on v
            SendGraphSetup(v,i,t)
    Remove all marks from v
```

Figure 3-24: The algorithm that F3 routers use to set up graphs.

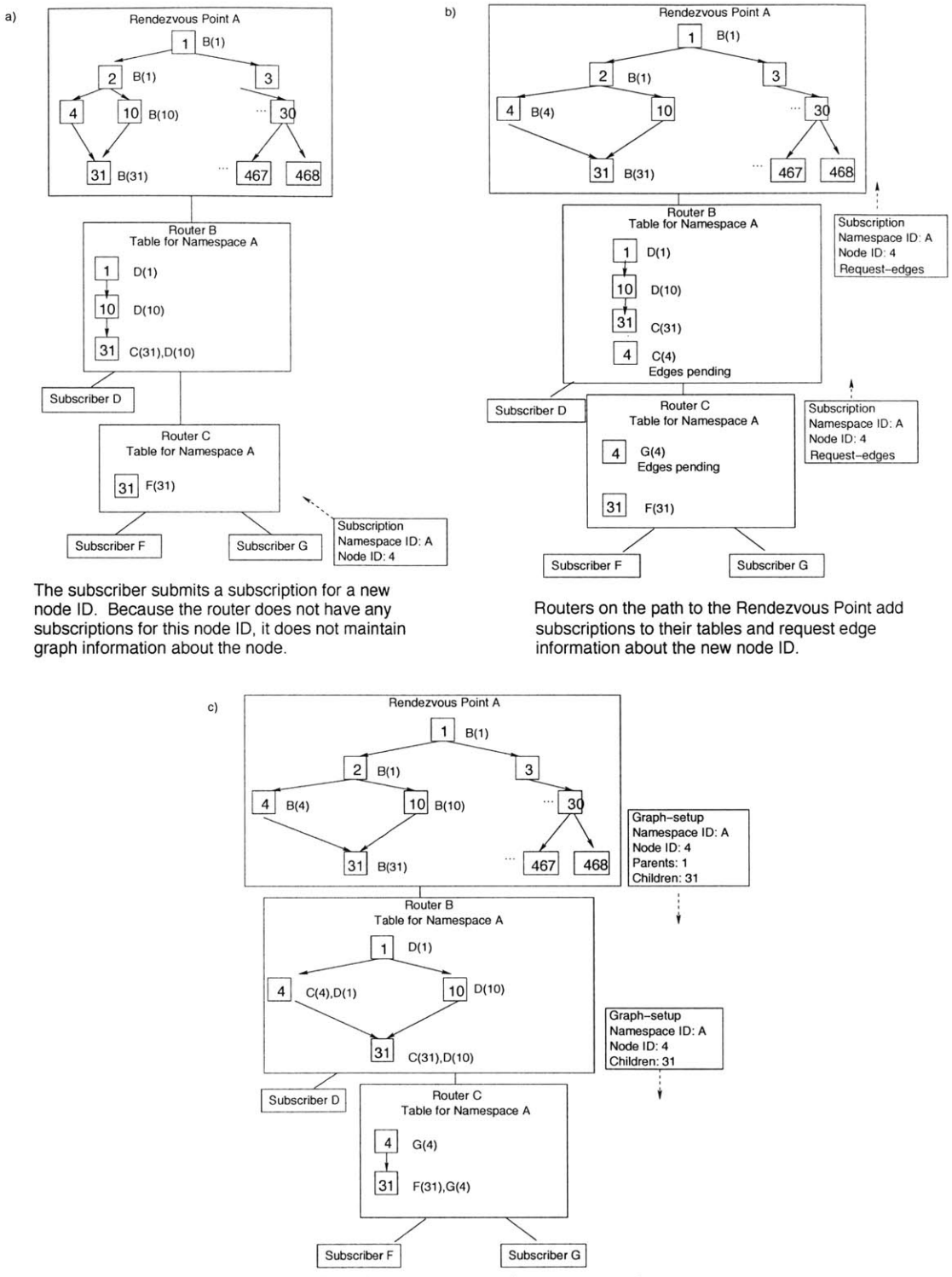


Figure 3-25: How routers obtain graph information from rendezvous points.

3.4.7 Resubscription

F3 uses periodic resubscription and subscription acknowledgment to ensure consistency of state in all routers. Figure 3-27 illustrates this process. The subscriber or router that maintains subscriptions for a particular rendezvous point periodically sends a resubscription message upstream toward the rendezvous point, as depicted in (a). This resubscription message identifies the nodes in the graph for which the router maintains an active subscription. When the upstream router receives the resubscription message, it looks for nodes in its graph that correspond to the message, and the router updates the resubscription timestamps for the neighbor.

Routers also send periodic resubscription acknowledgments downstream, as depicted in Figure 3-27 (b). If a router maintains an active subscription for a downstream neighbor, it will send a resubscription acknowledgment message to that neighbor. This will identify all the nodes in the router's graph for which the neighbor currently holds subscriptions. When the downstream neighbor receives one of these messages, it looks for the nodes in its graph that correspond to the message identifier, and updates the resubscription-acknowledgment timestamps for the given neighbor.

SendResubscription(namespace t)
    Let V be the set of all active nodes in graph G_t
    Send a resubscription message upstream containing V

ReceiveResubscription(neighbor i, nodes V, namespace t)
    Foreach node v in V
        Update the resubscription timestamp for subscriber i on node v in graph G_t

SendResubscriptionAcknowledgment(namespace t)
    Foreach neighbor i
        Let K_i be the set of all nodes to which neighbor i is subscribed in G_t
        Send a resubscription acknowledgment to i containing K_i

ReceiveResubscriptionAcknowledgment(nodes K, namespace t)
    Foreach node k in K
        Update the resubscription acknowledgment timestamp for node k in graph G_t

Figure 3-26: The resubscription algorithm.

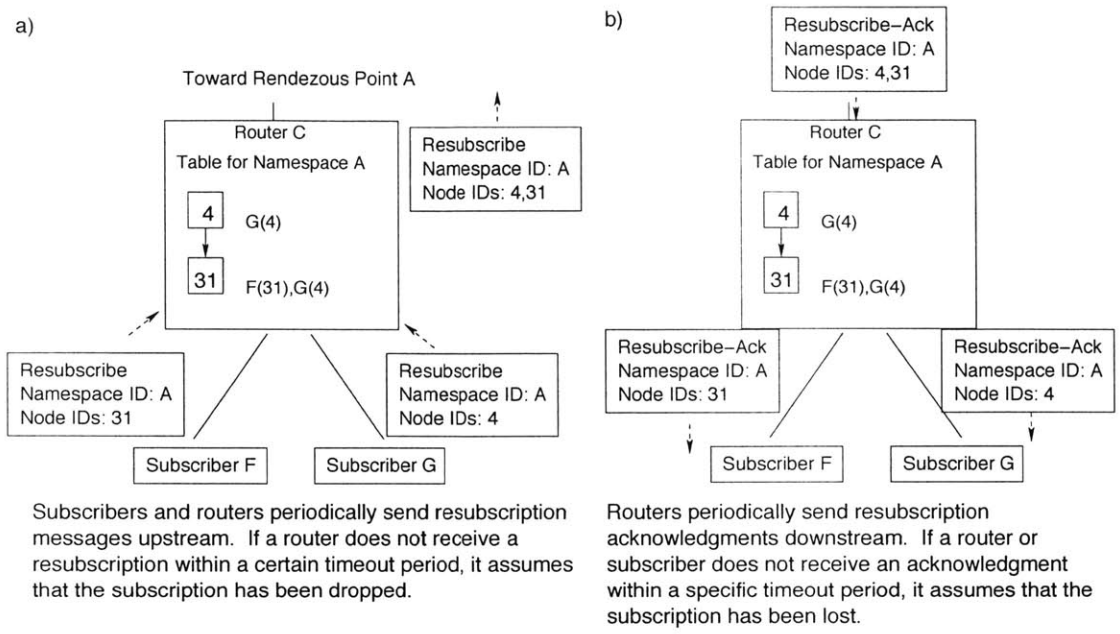


Figure 3-27: An example of resubscription in F3.

PeriodicCheckGraph(namespace t)
    Foreach node v in graph G_t
        If the resubscription acknowledgment timestamp has expired,
            mark the upstream subscription to v as interrupted.
        Foreach neighbor i in the subscription list i_v for v
            If the resubscription timestamp for i has expired,
                mark the downstream subscription to v from i as interrupted.

Figure 3-28: The algorithm that routers use to check subscriptions periodically.

Routers periodically check their graphs to determine whether any subscriptions have expired. For each node in each graph, each router checks to see whether a downstream subscriber has sent a resubscription message recently. If a neighbor has not sent a resubscription message recently, the router considers the subscription to be interrupted. The router also checks each node in the graph to see whether the upstream neighbor has sent a resubscription-acknowledgment recently. Again, if the router has not received an acknowledgment recently, it will consider the subscription to be interrupted. If F3 cannot recover an interrupted subscription, it will drop the subscription.

3.4.8 Unsubscription

To remove a subscription from F3, a subscriber must submit an unsubscription message to the same router that it used to create the subscription. Figure 3-29 presents the detailed algorithm. Each unsubscription message must contain the same node IDs and namespace ID that was contained in the original subscription. When the router receives the unsubscription message, it locates in its content graph the nodes specified in the message, and it removes the subscription from those nodes. The router also fixes the labels for these nodes and the nodes of any descendants, if necessary. If the router finds that any nodes listed in the unsubscription message have no more subscribers, the router marks the node for removal from its graph. It also sends an unsubscription message that identifies these nodes to its upstream neighbor. The upstream neighbor continues the same process. Figure 3-30 illustrates this process for baseball notifications.

```
RemoveSubscription(subscription s, neighbor i)
    Let t be the namespace indicated in the header of s
    Foreach node v indicated in the header of s
        Remove i from subscription list, i_v, for v
        Let D_v be the set of all descendants for node v in graph G_t
        Foreach node d in D_v + v, perform a top-down, breadth-first traversal
            Let o_{d,i} be the set of output labels for node d and neighbor i
            If v is in the set o_{d,i}, remove v from o_{d,i} then MergeParentLabels(d,i)

MergeParentLabels(node v, neighbor i)
    Foreach parent p of node v
        Let o_{p,i} be the set of output labels for node p and neighbor i
        Add o_{p,i} to o_{v,i}
```

Figure 3-29: The unsubscription algorithm.

3.4.9 End-to-End Message Loss Detection

One advantage of F3 over content-based multicast systems is that F3 applications can detect lost messages using sequence numbers. Figures 3-31 and 3-32 illustrate one approach to such loss-detection. The rendezvous point first annotates each node in its content graph with sequence numbers, as illustrated in Figure 3-31. Before a subscriber submits a subscription to F3, it first sends a message to the rendezvous point, identifying the node ID for that subscription, as shown in (a). The rendezvous

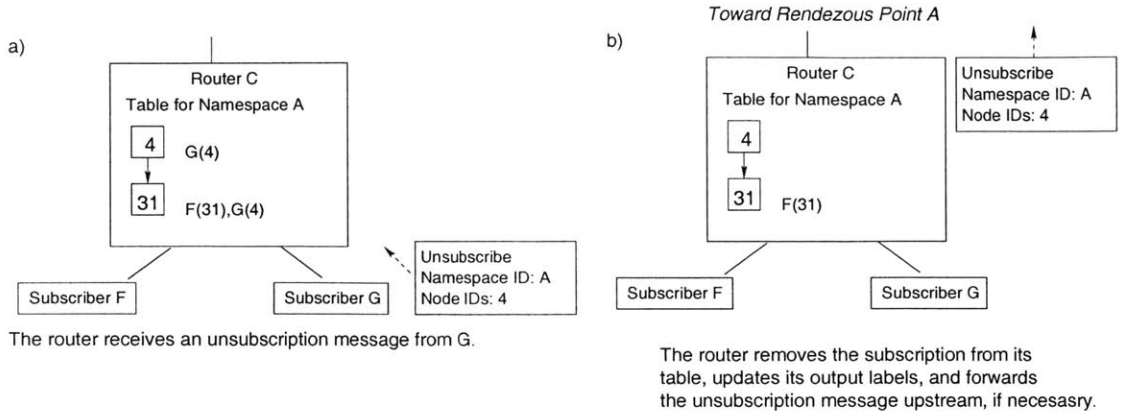


Figure 3-30: An example of unsubscription in F3.

point sends back a message that lists the sequence number for the node ID, as well as the sequence numbers for all of the node's ancestors. The subscriber stores these node IDs and their corresponding sequence numbers in a local table, as shown in (b).

When the rendezvous point receives a notification message, it finds the lowest node in its graph that matches the message, as depicted in Figure 3-32 (a). The router at the rendezvous point then increments the sequence number on the relevant nodes. The router attaches a sequence number header to the message, listing this node and its corresponding sequence number. Finally, the router attaches a standard F3 notification header onto the notification message and submits the message to F3.

F3 then forwards the notification to the appropriate subscribers, as depicted in Figure 3-32 (b). F3 reads F3 headers in order to forward notification but does not read the sequence number headers. Upon receiving a notification, each subscriber strips off the F3 header and reads the sequence number header of the message. It checks the sequence number for the given node ID against its local table. If the subscriber detects a break in the sequence numbers for any given identifier, it marks the message as lost and attempts to recover the message. Otherwise, the subscriber simply updates the sequence numbers in its local table.

It is important to note that the above approach is an end-to-end algorithm for loss detection. F3 routers do not participate in the detection of message losses or recovery from these losses. The purpose of presenting this algorithm here is to illustrate the important difference between F3 and content-based systems. Unlike content-based forwarding systems, F3 is compatible with loss detection algorithms that use sequence numbers.

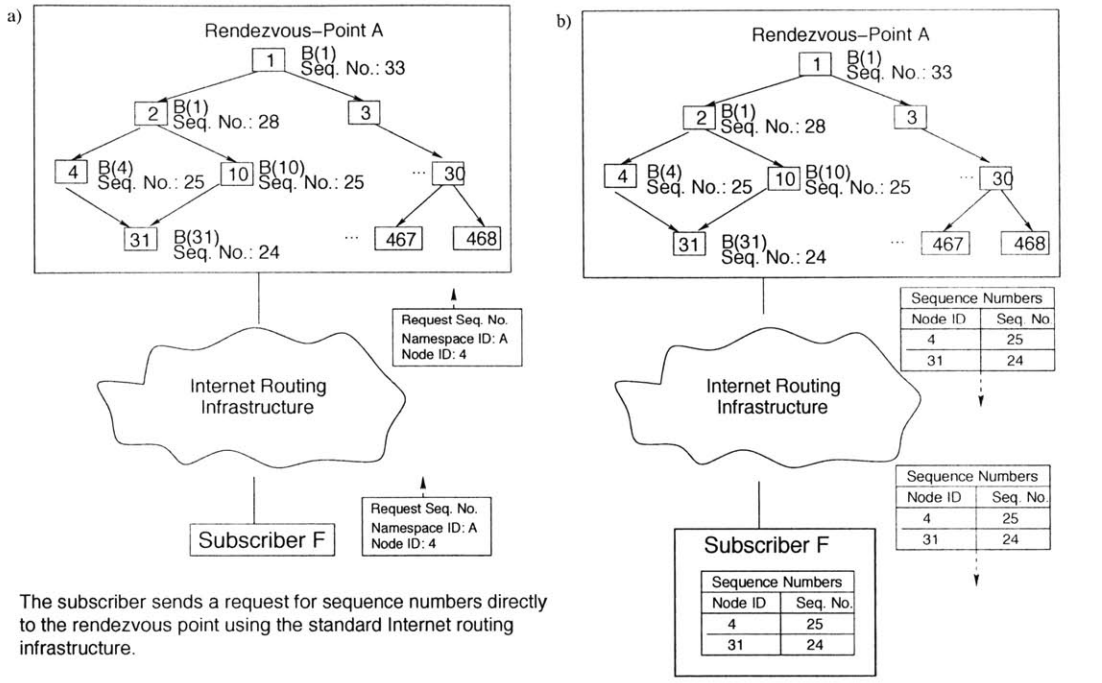


Figure 3-31: An approach to sequence numbering in F3. The subscriber first sends a request for sequence numbers directly to the rendezvous point directly through the Internet (a). The rendezvous point maintains sequence numbers for every node in its graph. Upon receiving the request, the rendezvous point sends a table directly back to the subscriber (b). This table specifies the sequence number for the requested node, as well as the sequence numbers for that node's descendants. The subscriber stores a copy of the table it receives.

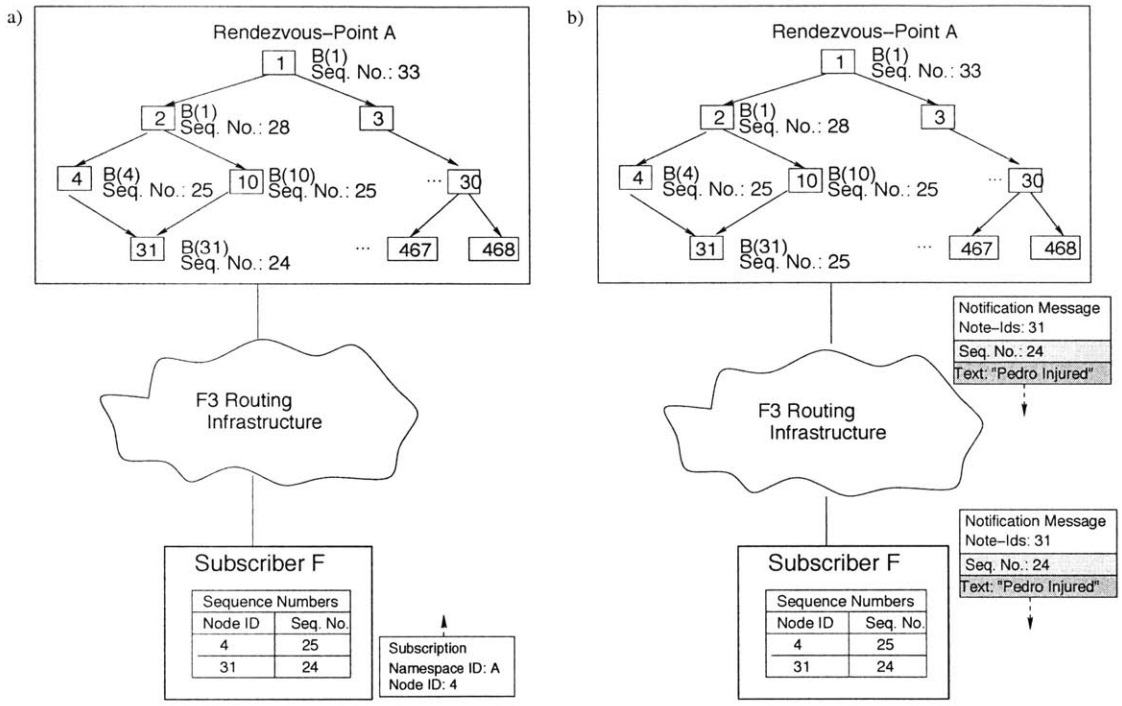


Figure 3-32: An approach to loss-detection in F3. The subscriber first submits a subscription to F3, and F3 sets up the subscription as before (a). When the rendezvous point receives a notification message, it adds a header to the message, indicating the sequence numbers of the node IDs in the message header (b). It also increments the sequence numbers of the node IDs in its table. F3 then delivers the message to the appropriate subscribers as before. Subscribers detect losses by noting gaps in the sequence numbers they receive.

3.5 Current status of F3

3.5.1 ns Simulator

A simulated version of F3 was constructed as part of this dissertation project. This version of F3 runs on the ns simulator [52]. Developed as a variant of the Real network simulator in 1989, ns has evolved into a public-domain simulator that is widely used in network research. Maintained and updated by a large user base and a small group of developers at the Information Sciences Institute at the University of Southern California, it supports simulation of TCP, routing, and multicast protocols over wide and wireless networks.

Three simulated applications were also developed to run in the nsenvironment. These applications disseminate three types of announcements, including baseball notices, traffic announcements, and messages formatted using attribute-value pairs. These simulated applications have been used for comparisons of the performance of F3 to other systems in network topologies of 1200 nodes [32], which are presented in the next chapter.

3.5.2 Router Software

A working version of the F3 router software was also implemented as part of this dissertation project. The router software currently runs on a standard PC in the Linux operating environment. It is implemented in C++, and comprises approximately 6000 lines of code. There are four basic modules that make up an F3 router:

- Routing daemon. The routing daemon executes the main control loop of the F3 router. It sets up network connections to neighboring routers and listens for messages from these neighbors. When the routing daemon receives a message, it passes the message to the F3 state machine module for processing.

- F3 state machine. The F3 state machine implements the core functionality of the F3 router. The state machine processes all incoming messages as described previously in this chapter. It also manages all the state associated with the router. This state includes a routing table and multiple notification forwarding tables.

- Routing table. The F3 routing table stores information about the F3 network topology. F3 routing tables are currently configured through static configuration files.

- Notification forwarding table. Each notification forwarding table stores content graph information for a single namespace in the F3 network. Notification forwarding tables provide an neighbor for constructing graphs, traversing graphs, traversing them, storing subscriptions, and removing them. Notification forwarding tables also provide an neighbor for managing a graph's output labels.

The F3 router software currently runs as a user process. A version of F3 running within the Click modular router environment is also under development [30].

3.5.3 Preprocessor Software

Two preprocessors have been developed for F3 and others are under development. The first is an extensible preprocessor written in C++ and runs on a single processor. The preprocessor creates content graphs dynamically, adding new nodes to its graph as it receives subscriptions. The base preprocessor manages generic SubscriptionObject C++ objects. SubscriptionObjects are objects that provide a core set of functions for manipulating subscriptions and creating graphs. First, they provide functions for converting formatted message data into SubscriptionObjects. Second, they provide functions for testing their relationships to other SubscriptionObjects. Using these functions, the base preprocessor can sort SubscriptionObjects into a partially-ordered content graph.

Programmers can extend the functionality of this base preprocessor by creating new kinds of SubscriptionObjects. For example, a programmer could define a new class of SubscriptionObject, called a NumericRangeObject, for preprocessing numeric range expressions. To define this new class, the programmer would have to define functions for converting formatted message data into NumericRangeObjects. The programmer would also have to define functions for testing the relationships between NumericRangeObjects.

This base preprocessor has been extended to create an attribute-value preprocessor. This preprocessor takes subscriptions and notifications formatted as XML attribute-value pairs and produces F3 message headers as output. With this preprocessor, applications can access the F3 system through the same attribute-value neighbors that are provided in content-based routing system. To implement this preprocessor, a new class of SubscriptionObject, called an AttributeValueObject, was defined. AttributeValueObjects use the Xerces XML parser, implemented by Apache, to convert XML message data into AttributeValueObjects.

3.5.4 RSS News Dissemination Service

An actual news dissemination service also currently operates on F3. This news service disseminates news updates from a variety of news sources, including the BBC. The news stories submitted to F3 are formatted in RSS, a standardized XML grammar for news announcements. Using the RSS grammar, a news publisher can specify the title, story description, and URL associated with an individual news announcement. F3 uses a modified version of RSS in which publishers can also specify the general topic areas–such as "Business" or "Arts"–associated with each announcement.

Subscribers reach the dissemination service through the F3 news player, an interactive web page that is implemented as a Macromedia Flash program. Figure 3-33 provides a screenshot of the news player's subscription page. On this page, subscribers can sign up to receive updates on a variety of topics from a news source. The news player then formats the subscription as an RSS template and sends the subscription to a remote preprocessor, which is specifically designed to handle RSS news announcements. The preprocessor processes the subscription and returns the appropriate F3 header to the news player. The news player then submits the preprocessed subscription to a nearby F3 router, and F3 sets up the subscription as described previously.

When an F3 news source publishes a story, it submits the story to the same preprocessor described above. This preprocessor returns the appropriate header to the F3 news source, the news provider appends this header to the story, and submits the entire message to F3 for distribution to the appropriate news players. When a news player receives the story, it updates its notification page with the new story. Figure 3-34 contains a screenshot of the news player notification page. The news player also pops up a small newsgram window containing the headline and summary of the story. A popup newsgram appears at the bottom right-hand side of Figure 3-34.

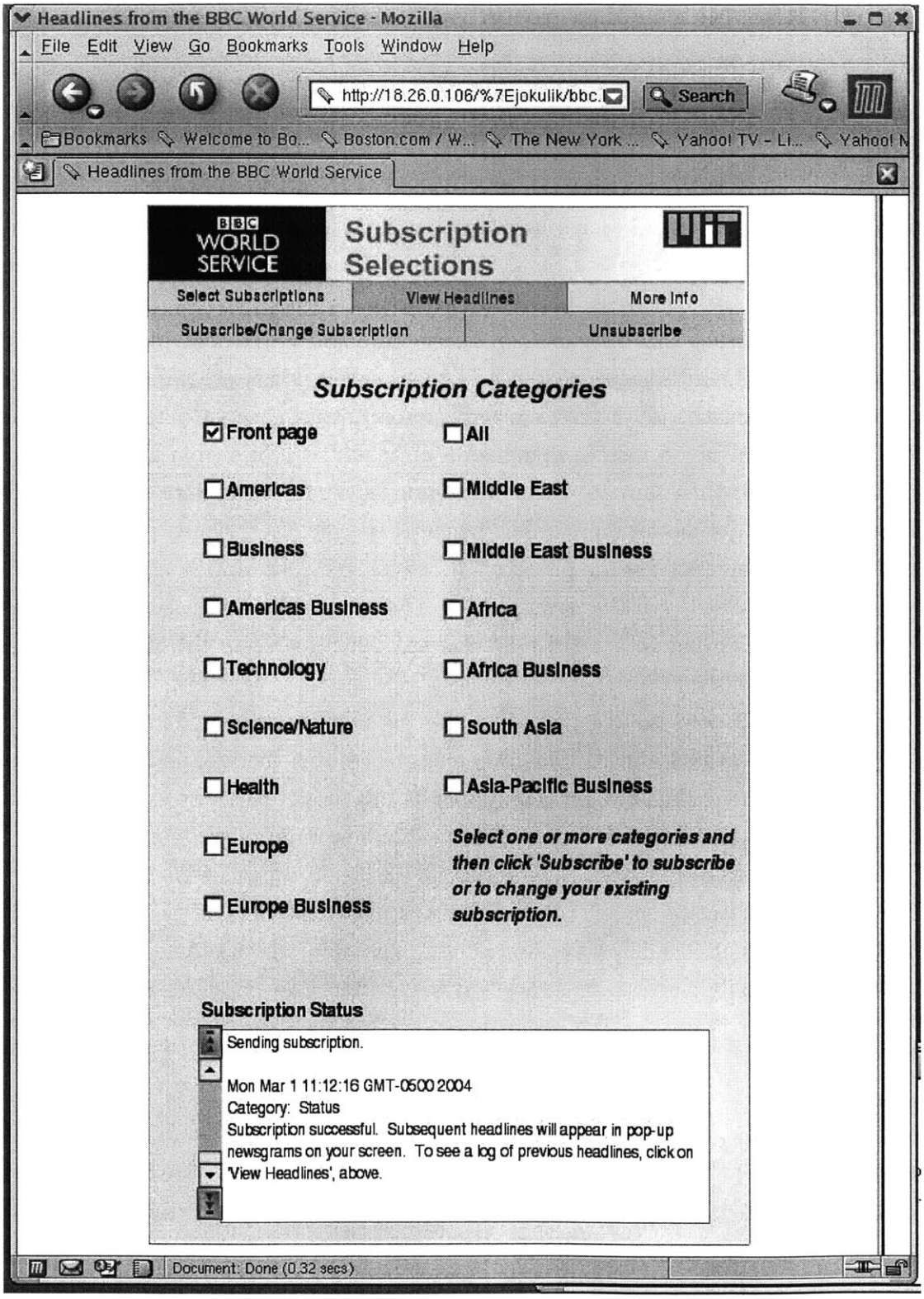


Figure 3-33: Screenshot of the F3 news player subscription page.

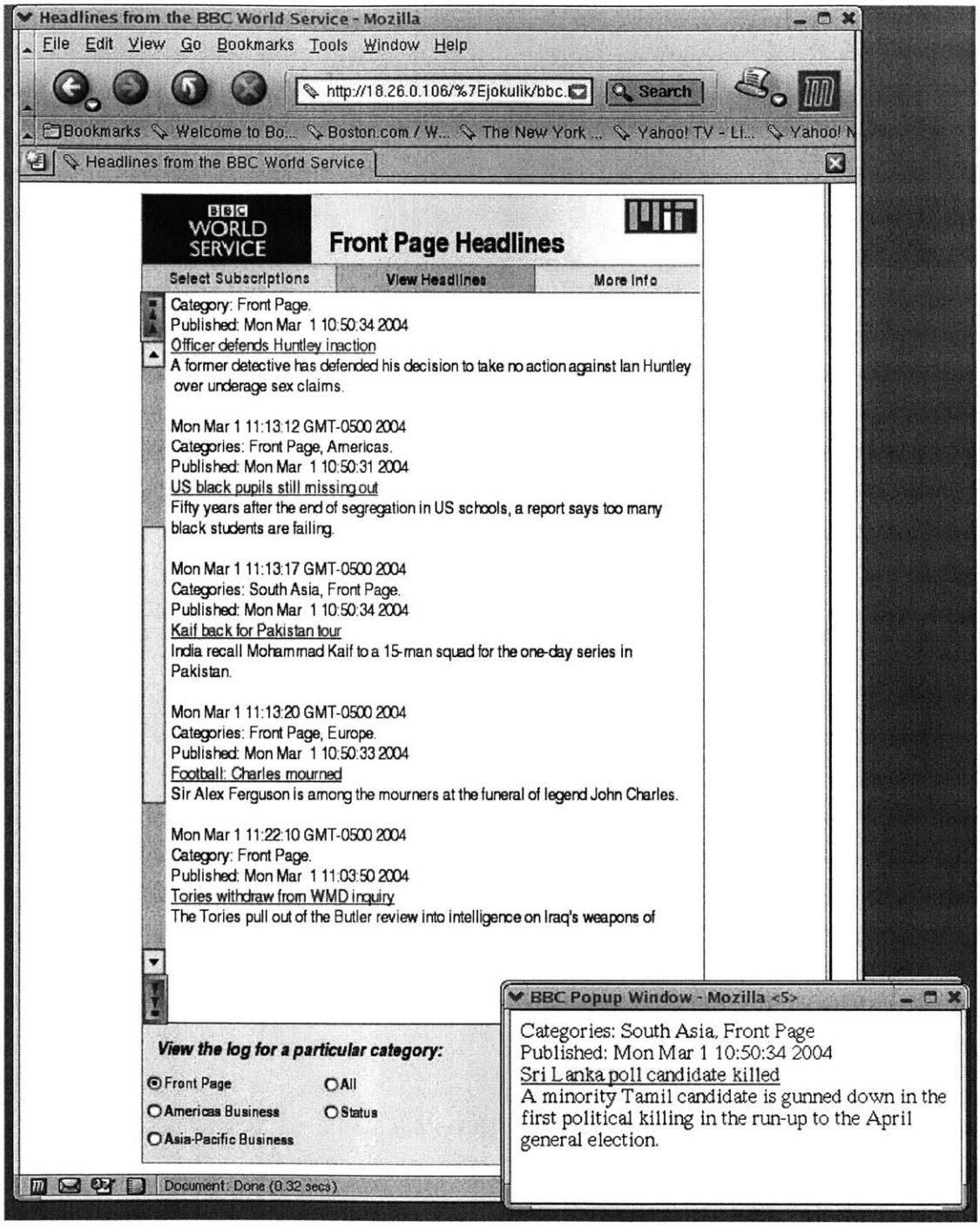


Figure 3-34: Screenshot of the F3 news player notification page

Chapter 4

Experimental Studies

How well does F3 actually perform? Does it process subscriptions and notifications as quickly as competing systems do? Does it overload the network with unnecessary traffic? What factors affect its performance? What performance measures best show its strengths and weaknesses? This chapter presents results from four studies designed to answer such questions. Studies 1 through 3 directly compared performance of F3 and competing subscription systems. Study 4 compared performance of F3 and a modified version of F3. A theoretical study of these results is provided in Appendix A.

4.1 Study 1

It is well-known that unicast and multicast subscription systems can flood networks with multiple copies of the same messages. My goal in carrying out this study was to examine F3 message production and, more specifically, to compare F3 network traffic to traffic in unicast and multicast systems. To make the necessary comparisons, I set up a simulated network and made observations on traffic in unicast, multicast, and F3 systems in varying conditions.

4.1.1 Method

The simulator used in this study was the ns network simulator [52]. The simulations were carried out on a 1200-node, transit-stub network created using the GT-ITM topology generator.

Subscription Systems

This study compared performance of four approaches: one unicast approach, two single-identifier multicast approaches, and F3.

- Unicast. Subscribers send subscription requests directly to a notification source. The notification makes a copy of the notification for each of the relevant subscribers and addresses each copy with the subscriber's Internet address. The notification is forwarded to the subscriber using the standard unicast routing mechanisms.

- Single-identifier multicast with overlapping identifiers, or *overlapping ID multicast.* When a subscriber signs up for a category of notices, the service attaches a single predetermined identifier to the subscription. Identical subscription topics receive the same identifier, and different subscription topics receive different identifiers, even when a new subscription used overlaps or is a subset or superset of an existing category. The identifiers do not convey anything about interrelationships among subscription categories. Figure 4-1 provides an example of this type of multicast system. A subscription for results on all Red Sox games would receive one identifier; a subscription for results on both Red Sox and Tigers games would receive another identifier; a subscription for results on any baseball game would receive still another. Notifications (e.g., the score for a Red Sox vs. Tigers game) would carry only a single identifier. When a notification was relevant to more than one topic (as in the example in Figure 4-1), the source would have to generate multiple copies of the notification, one for each subscription topic.

- Multicast with disjunctive identifiers, or *disjunctive ID multicast.* In applications running on these multicast systems, the subscription area is first broken into disjoint subcategories, and each subcategory is assigned a unique identifier. When a subscriber submits a subscription, the subscription service must identify all disjoint subcategories that are covered by the subscription. Each subscription is then represented by a string of subcategory identifiers. Each notice, however, carries a single identifier that indicates the topic of the notification. An example of this type of system appears in Figure 4-2. In the example, a subscriber has signed up for notifications on all game results. The subscription is translated by the subscription service into a string of identifiers

that cover all possible baseball games. Each notification carries only a single identifier that specifies results for a particular game.

- F3. This system is described fully in Chapter 3. Figure 3-20 illustrates the forwarding-algorithm used by this system.

Though neither of the single-identifier multicast approaches simulated is very sophisticated, in both approaches, both approaches are able to assign identifiers to the topic space. Recall that the general problem of selecting an optimal set of identifiers for a single-identifier multicast system is NP-hard [1].

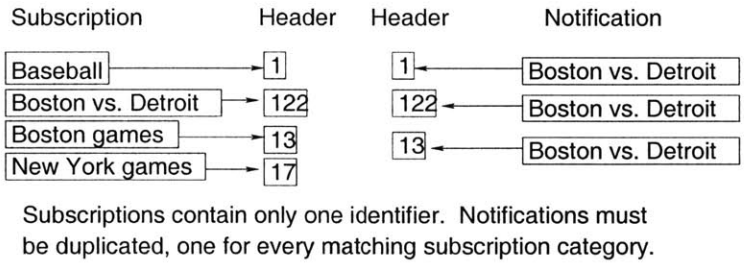


Figure 4-1: Baseball messages and their corresponding headers using a single identifier for each overlapping subscription category.

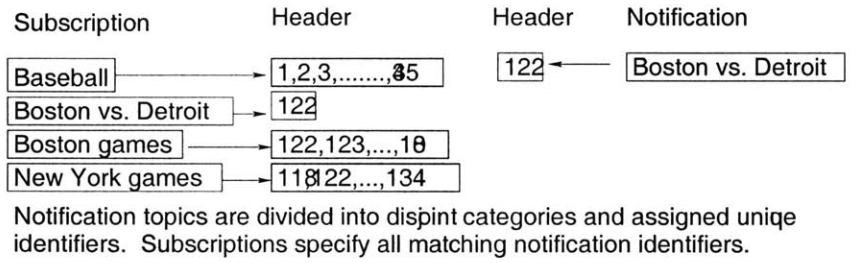


Figure 4-2: Baseball messages and their corresponding headers using a single identifier for each disjunctive subscription category.

Simulation Scenarios

Three scenarios were developed for this study. In each, a subscriber at each node of the network signed up for notifications from a single source, and the network source sent back relevant notifications in response. Due to memory requirements of the simulation software, the three simulations were small in terms of numbers of number

of subscribers. The scale of the simulations is adequate, however, for revealing the real differences in performance of unicast, multicast, and F3 systems.

The first scenario modeled performance of a baseball-news service. Subscribers in the simulated service were randomly assigned baseball teams to follow. Total number of teams in the league was 30, and 40% of subscribers were assigned one team to follow; 50% were assigned two teams; and 10% were assigned all teams. Each notification from the source covered a game between a specific pair of teams, and all pairs were equally likely to be the subject of a notification.

The second scenario modeled performance of a traffic alert system that covered a single highway with 20 exits. Subscribers were assigned alerts on sections of the highway between two randomly selected exits (e.g., Exits 12 through 14), and each notification was relevant to a randomly selected stretch of highway (e.g., congestion between Exits 5 through 12). Put another way, each subscription and notification contained 20 possible attributes (one for each exit) and two values (selected or not). A notification was relevant to a subscription when it covered any portion of highway specified in the subscription. This scenario is of particular interest because traffic subscriptions contain numeric range expressions, which can produce relatively deep graphs. In this scenario, the maximum depth of the graph is 20, or the total number of possible exits. Figure 4-3 depicts an example of a content graph for traffic alerts.

The third scenario modeled performance of a generic subscription service that responded to subscriptions formatted as attribute-value pairs. Each subscription consisted of five attributes, each with four possible values. Values were randomly assigned to subscriptions, and the values had the same probabilities of occurrence. Individual attributes could also contain a wildcard. The probability of occurrence of a wildcard as the value of an attribute was .3. Notifications also consisted of four attributes, each of which had four possible values. Probability of occurrence of any specific value was .3. The probability that a given attribute would appear in a notification was .7. This scenario closely resembles those used in measuring performance of such attribute-value systems as SIENA, READY, Gryphon, LeSubscribe, and Xfilter [12, 24, 2, 46, 3]. Figure 4-4 depicts an example of a content graph for generic attribute-value pairs with two attributes and two values.

For each scenario, the source sent out 1000 notifications. Routers in the network distributed these notifications using a shortest-path dissemination tree. The simulations did not contain any network losses or queuing delays. Each scenario ran 20 times using the same topology, each time with a different, random notification source. Characteristics of the three scenarios are summarized in Table 4.1.

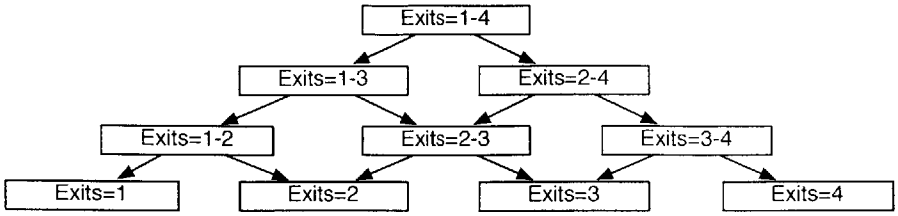


Figure 4-3: An example of a content graph for traffic alerts covering four highway exits.

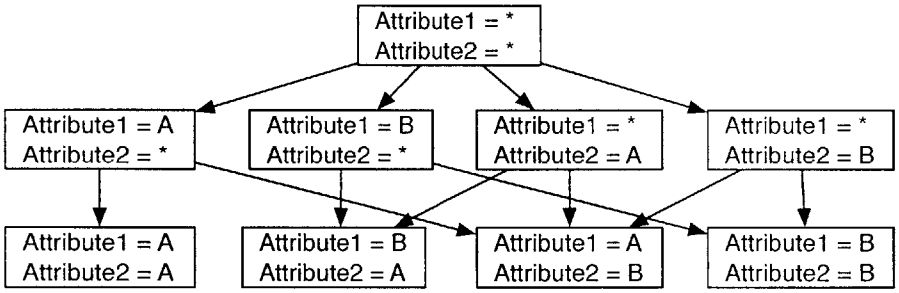


Figure 4-4: A content graph for generic attribute-value pairs with two attributes and two values.

Performance Measures

Four outcomes were measured in this study: (a) number of notifications per link; (b) number of identifiers stored per routing table; (c) number of identifiers contained in each subscription header, and (d) number of identifiers contained in each notification header. The results were tabulated in two ways: average results on each measure for the whole network and average result for the most heavily loaded router in the network.

4.1.2 Results

The results of this study appear in Table 4.2. The pattern of results is similar at typical router and heavily loaded routers, but performance differences of the systems were more pronounced at heavily loaded routers. The problems of unicast and multicast systems were also apparent with each simulation scenario, but the traffic simulation brought out the problems of unicast and multicast systems most clearly.

The top panel of the table contains results on notifications per link. It is clear

that unicast systems overloaded the system with notifications. At typical routers, unicast systems produced between 2 and 5 times as much traffic as F3 produced. At heavily loaded routers, the amount of traffic for unicast systems was between 4 and 130 times as much as for F3. Overlapping-Id multicast systems also produced too many unnecessary messages, as much traffic 3 times as many messages as F3 at typical routers and as much as 60 times as many at heavily loaded routers.

The next panel of the table contains results on the number of identifiers per subscription. Overlapping-Id multicast and F3 contained only one identifier on each subscription, whereas disjunctive ID multicast contained long strings of identifiers on subscription messages. On the third panel of the table, which contains results on number of identifiers per table, the results for disjunctive ID multicast are again anomalous. Disjunctive ID tables have far more identifiers than other systems do at both typical and heavily loaded routers.

The final panel of the table contains results on identifiers per notification. The multicast systems contained only one identifier per notification–as required by definition. F3 produced more identifiers per notification, but it is notable that the number of identifiers in F3 was also very small.

4.1.3 Conclusions

The simulations in Study 1 were small in scale, but they were large enough to show the crippling weaknesses of unicast and overlapping ID multicast systems. Even in simulations with a small number of subscription topics and a small number of subscribers, unicast and overlapping ID multicast systems produced far too many messages. Only one conclusion can be drawn. In larger applications with millions of subscribers, these systems would flood a network with unnecessary and redundant messages. Redundant messages are costly in terms of network bandwidth and delays. Because performance improvements of an order of magnitude are possible, no further consideration will be given to unicast and overlapping ID multicast systems in the remaining studies in this chapter.

The performance of disjunctive ID multicast systems was more acceptable. This system did not produce redundant copies of messages, as unicast and overlapping ID multicast methods did. One possible concern about disjunctive ID multicast, however, is the length of the identifier strings in subscriptions. In the simple scenarios used in this study, the identifiers were between 20 and 120 times as long as the identifiers in other systems. The length of the identifiers was not unmanageable, however, in these

small-scale simulations. Further study is needed to determine whether the length of identifiers might be a serious problem in large-scale applications.

The performance of F3 in these small-scale simulations was very good. F3 did not produce redundant copies of messages, and it did not attach long strings of identifiers to subscriptions. Identifiers on notifications were longer in F3 than in other systems. However, the identifiers were still very short. Further study of F3 performance definitely seemed warranted.

Characteristic		Baseball	Traffic	Att. Val.
S	The maximum number of subscribers per router	1200	1200	1200
i	The maximum number of interfaces per router	6	6	6
N	The total number of possible subscription topics	465	190	3125
d	The maximum depth of the graph	3	20	5
e	The maximum degree of a node in the graph	29	20	5

Table 4.1: Characteristics of 3 scenarios used in Study 1.

Format	Typical Router			Most Heavily Loaded Router		
	Baseball	Traffic	Att. Val.	Baseball	Traffic	Att. Val.
Notifications per link						
Unicast	909.1	3105.1	100.1	66,560.6	128,336.2	3898.2
Overlapping-Id	312.7	1870.0	85.3	1299.8	61,423.5	3543.9
Disjunctive ID	245.5	619.5	52.7	1000.0	1000.0	1000.0
F3	245.5	619.5	52.7	1000.0	1000.0	1000.0
Ids per subscription						
Overlapping-Id	1.0	1.0	1.0	1.0	1.0	1.0
Disjunctive ID	50.8	118.4	21.8	50.7	118.5	22.0
F3	1.0	1.0	1.0	1.0	1.0	1.0
Ids per table						
Overlapping-Id	4.5	3.9	16.9	361.2	187.6	955.4
Disjunctive ID	103.6	145.8	183.4	465.0	210.0	3102.5
F3	4.5	3.9	16.9	361.2	187.6	955.4
Ids per notification						
Overlapping-Id	1.0	1.0	1.0	1.0	1.0	1.0
Disjunctive ID	1.0	1.0	1.0	1.0	1.0	1.0
F3	1.0	1.5	1.3	1.2	4.2	3.7

Table 4.2: Performance of 4 Internet Subscription Systems under Three Simulation Conditions

4.2 Study 2

My goal in carrying out this study was to determine whether F3 works more quickly than competing systems do. Slow processing times are a known weakness of content-based subscription systems, and network engineers have recently been working to improve these processing times. It seemed especially important therefore to compare the performance of F3 and content-based systems. Slow processing times may also be a problem in disjunctive ID multicast systems, so I also included disjunctive ID systems in this study.

4.2.1 Method

A working router was implemented in C++. The router ran on the RedHat Linux 8.0 operating system on a 2.8 GHz Intel processor with 1.2 GB of memory. F3 and disjunctive ID multicast systems were measured on this router. Version 1.6.8 of the SIENA fast-forwarding software was obtained from the University of Colorado [40], and it was run on the same platform and hardware for this study.

Subscription Systems

The three subscription systems studied in this experiment were F3, disjunctive ID multicast, and the SIENA content-based forwarding approach [13, 12]. SIENA, which uses a fast-forwarding algorithm to match attribute-value subscriptions to notifications, is one of the oldest and best known content-based forwarding systems. Its performance is representative of the current state-of-the-art in content-based forwarding systems [12], and its published performance is comparable to those used in other content-based forwarding systems, such as Le Subscribe [43, 46], Xfilter[3], and READY [24]. It is also the only one of these systems whose software is publicly available for experimentation. This study did not examine unicast and overlapping ID systems because Study 1 established the crippling weakness of these systems in large and complex applications.

Simulation scenarios

The three scenarios developed for Study 1 were used in this study. Because this study included 1 million, rather than 1000, subscribers, the complexity of each scenario was accordingly increased for the study. In the fist scenario subscribers could select from 100 baseball teams, rather than the 30 teams available in the previous study. In

the second study, subscribers could request traffic alerts for a highway system of 60 exits, rather than 20. This scenario was of particular interest for investigating F3's performance in handling deep graphs created by numeric range expressions. In this case, the graph had a height of 60. Finally, in the third scenario, subscribers could request notifications with 6, rather than 5, attributes and values.

In each scenario, a source sent out sent out 1000 notifications. Subscription processing times were measured continually at a single router (with ten neighboring routers) as the number of subscribers increased from 1 to 1 million. Performance time was also measured when the router was forwarding the notifications to subscribers. Each subscription was forwarded to the router from another router randomly chosen from a group of four neighboring routers. Each scenario was run 20 times, and in each repetition of a scenario, subscriptions were forwarded to the target router from randomly chosen, neighboring routers. Characteristics of the three scenarios are summarized in Table 4.3.

Performance Measures

Three outcomes were measured in this study: (a) the size of the forwarding table, (b) the total time that it took the router to process subscriptions, and (c) the total time for processing notifications.

4.2.2 Results

Table 4.4 shows subscription- and notification-processing times and table sizes for the three simulations run with 1 million subscribers in the system. The top panel of the table contains results on table size. The results show that the content-based SIENA system produces much larger table sizes than disjunctive ID and F3 systems do. In fact, the forwarding tables set up by SIENA in the three scenarios were enormous. For the traffic alert simulation, for example, SIENA's forwarding table occupied 439 MB of memory, whereas the F3 forwarding table occupied only 471.1 KB. In other words, SIENA's forwarding table was approximately 1000 times the size of the forwarding table of F3.

The middle panel of the table contains results on per-subscription processing times. The results show that disjunctive ID systems take large amounts of time to process subscriptions. For the baseball problem, for example, disjunctive ID time-requirements were about 100 times as large as the time-requirements of F3. Subscription processing times for F3 and SIENA were similar.

The bottom panel of the table contains results on per-notification processing times. The results for disjunctive ID and F3 systems are similar, and they are strikingly different from the results for SIENA. The long processing times for SIENA messages undoubtedly result from the enormous size of SIENA forwarding tables. The difference in processing times for SIENA and other systems is staggering: For the attribute-value problem, F3 required more thar three seconds per notification, whereas F3 required less than 10 microseconds per notification. The ratio of the two processing times is almost 300,000 to 1.

Figures 4-5– 4-7 illustrates how each measurement grew with the number of subscriptions received by each router. As the graphs at the left side of Figure 4-5 illustrate, the size of the SIENA forwarding table grows approximately linearly as the number of subscribers increases from 1 to 1 million. This linear growth is very different from the growth in table sizes for F3 and disjunctive ID systems. The pattern of growth for these systems can be seen in the panels at the right side of the figure. These panels present a magnified view nearly of portions of the panels at the left. The magnified views show that the growth in size of tables in F3 and disjunctive ID systems stops long before one million subscriptions are reached. In the attribute value simulation, for example, growth in table size slows at about 100,000 subscriptions. There are only 15,625 unique subscriptions in this simulation, and a router that has received 100,000 subscriptions has entered every possible subscription in its forwarding tables. SIENA forwarding tables, however, continue to grow long after growth of other forwarding tables stops. This example illustrates the general point that SIENA tables continue to grow, even when no new, unique subscriptions are being added to SIENA forwarding tables.

Figure 4-6 illustrates patterns of growth in per-subscription processing time. The patterns can be seen most clearly in the magnified views presented in the panels at the right of the figure. Particularly notable are changes in subscription processing time for disjunctive ID systems and F3. The per-subscriber processing time for disjunctive ID systems spikes with a relatively small number of subscribers and ultimately drops to dramatically low levels. The factors that produce this curve are not hard to understand. Subscriptions in disjunctive ID systems can contain many identifiers; a single attribute-value subscription with all wildcards, for example, expands to a subscription of $5^6 = 15,625$ disjunctive identifiers. Routers in disjunctive systems fill up their forwarding tables with identifiers after receiving a thousand or so subscriptions, and they do not have to make many entries for subsequent subscribers. The total cost of inserting ten thousand subscribers and a million subscribers is roughly the same, and

therefore the per-subscriber cost decreases with the number of subscribers.

The cost of processing subscriptions is higher for the F3 forwarding system than for SIENA. This is because F3 does a lot of initial work to set up content graphs. Once content graphs are set up for initial subscribers, it takes F3 much less time to process subsequent subscriptions. These results also show that the amount of time that it takes F3 to process subscriptions depends upon the characteristics of the graph. The baseball scenario involves the shallowest graphs, and F3 processes subscriptons in this scenario very quickly. The traffic scenario involves the deepest graphs, and F3 processes subscriptions in this scenario more slowly. Even in this scenario, where the graph has a depth of 60, the maximum amount of time it takes F3 to process subscriptions is less than a disjunctive Id system. For large numbers of subscribers, the extra time that F3 needs for handling subscriptions is significantly less than the time needed by disjunctive ID systems.

As the graphs at the left side of Figure 4-7 illustrate, per-notification processing times for SIENA increase linearly with increases in the number of subscriptions in the system. This linear growth is very different from the growth in per-notification processing time for F3 and disjunctive ID systems. However, the linear growth is very similar to the growth in size of forwarding tables in SIENA with increasing subscriptions. The similarity in growth for the two measures is undoubtedly related. Processing times grow at a staggering rate in SIENA with increasing subscription bases because table sizes increase dizzyingly with increases in the subscription base. These results are consistent with previous studies that show that notification processing times for content-based forwarding systems grow linearly with the number of subscriptions in the system [12, 6, 3, 43, 46]. In contrast, the time that it takes F3 or disjunctive ID system to process notifications grows logarithmically as number of subscriptions in the system increase.

4.2.3 Conclusions

The bottom line is that the F3 forwarding system avoids the pitfalls of content-based and disjunctive ID systems. Although F3 processes subscriptions more slowly than SIENA does, F3 subscription processing is faster than subscription processing in disjunctive ID systems. Most important, F3 processes notifications quickly, even when there are many subscriptions in the system. In this respect, F3 contrasts sharply with content-based systems. Content-based systems process notifications quickly only when there are a few subscriptions in the subscription base. With many subscriptions,

notification processing proceeds at a snail's pace in content-based systems.

Characteristic		Baseball	Traffic	Att. Val.
S	The number of subscribers	1–1M	1–1M	1–1M
i	The number of interfaces	10	10	10
N	The maximum number of subscription topics	5151	1830	15,625
d	The maximum depth of the graph	3	60	6
e	The maximum degree of a node in the graph	99	60	6
m	The maximum number of attributes per message	2	60	6
a	The number of possible attributes	101	60	6
v	The number of possible values	2	2	5

Table 4.3: Characteristics of 3 scenarios used in Study 2.

System	Baseball	Traffic	Att. Val.
Table size (kB)			
SIENA	54,384.92	439,536.4	76,859.3
Disjunctive ID	1322.9	472.7	4004.2
F3	1356.7	471.1	4532.7
Processing time per subscription (usec/subscription)			
SIENA	3.9	28.4	14.7
Disjunctive ID	826.0	1481.2	290.7
F3	5.5	78.7	138.0
Processing time per notification (usec/notification)			
SIENA	18,668.0	1,678,498.2	3,037,973.0
Disjunctive ID	19.9	14.9	27.5
F3	10.4	10.3	10.8

Table 4.4: Performance of 3 subscription systems forwarding notices to a million subscribers

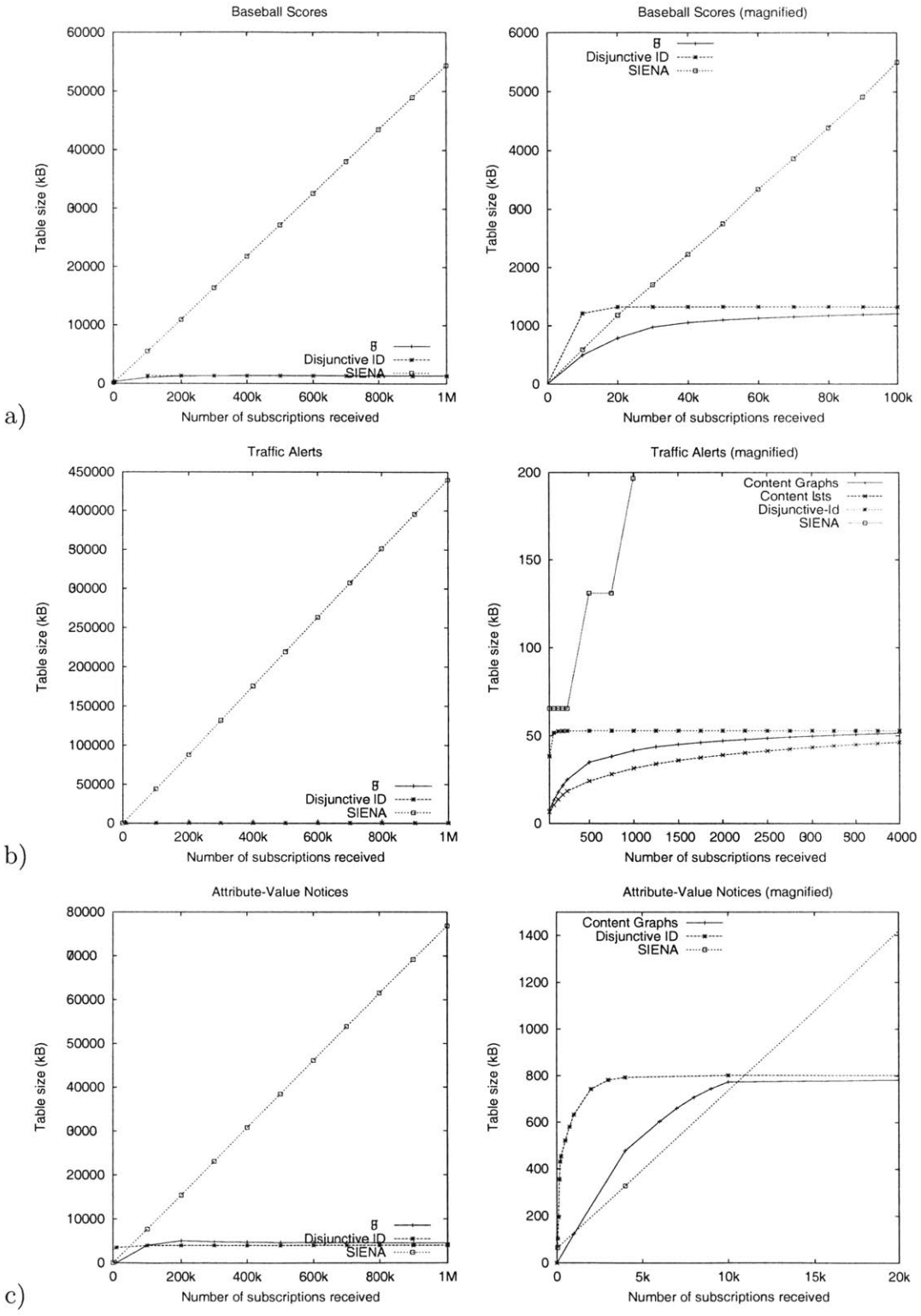


Figure 4-5: Table sizes (kB) of 3 subscription systems when forwarding notices.

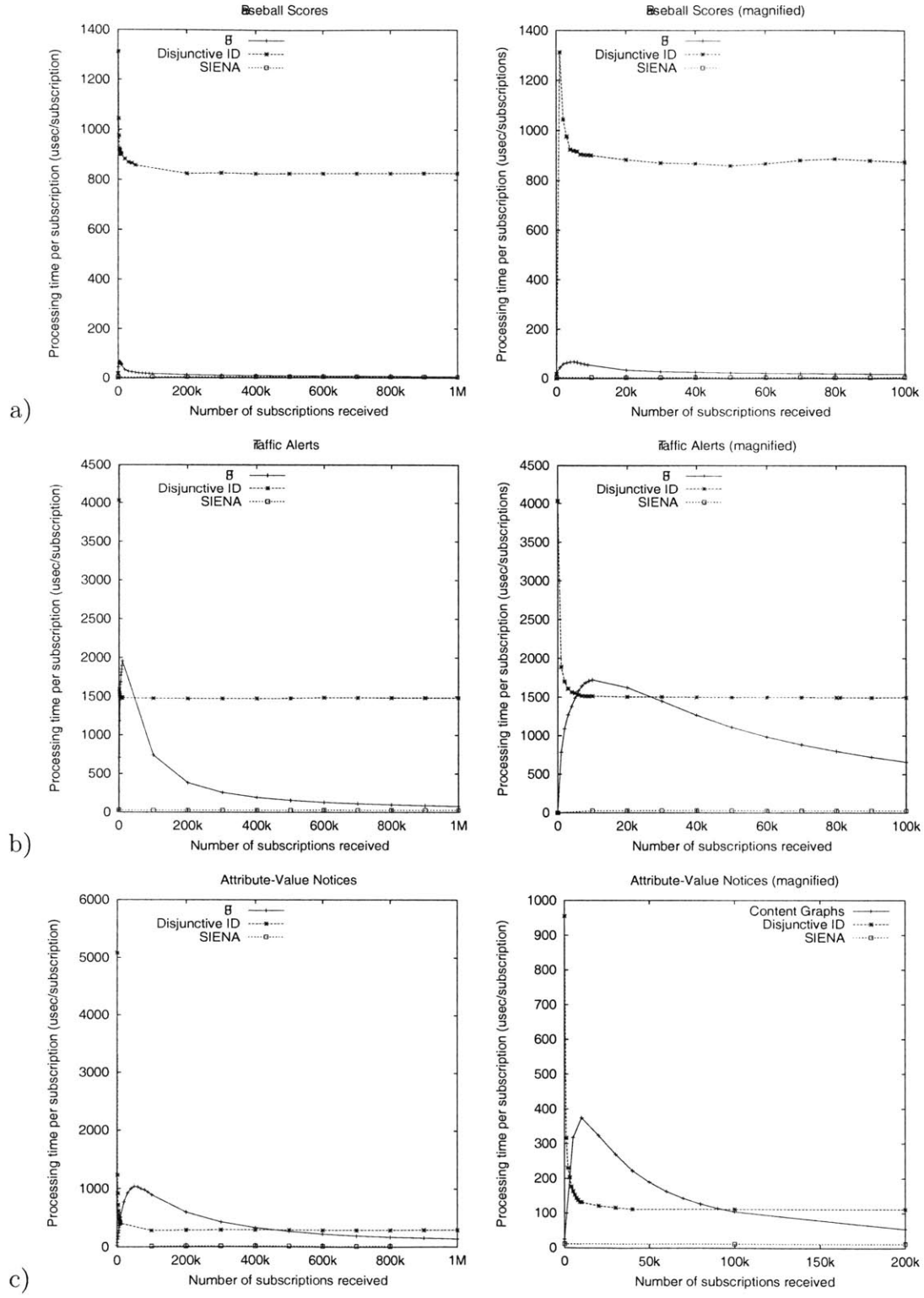


Figure 4-6: Processing times per subscription (usec/subscription) for 3 subscription systems when forwarding notices.

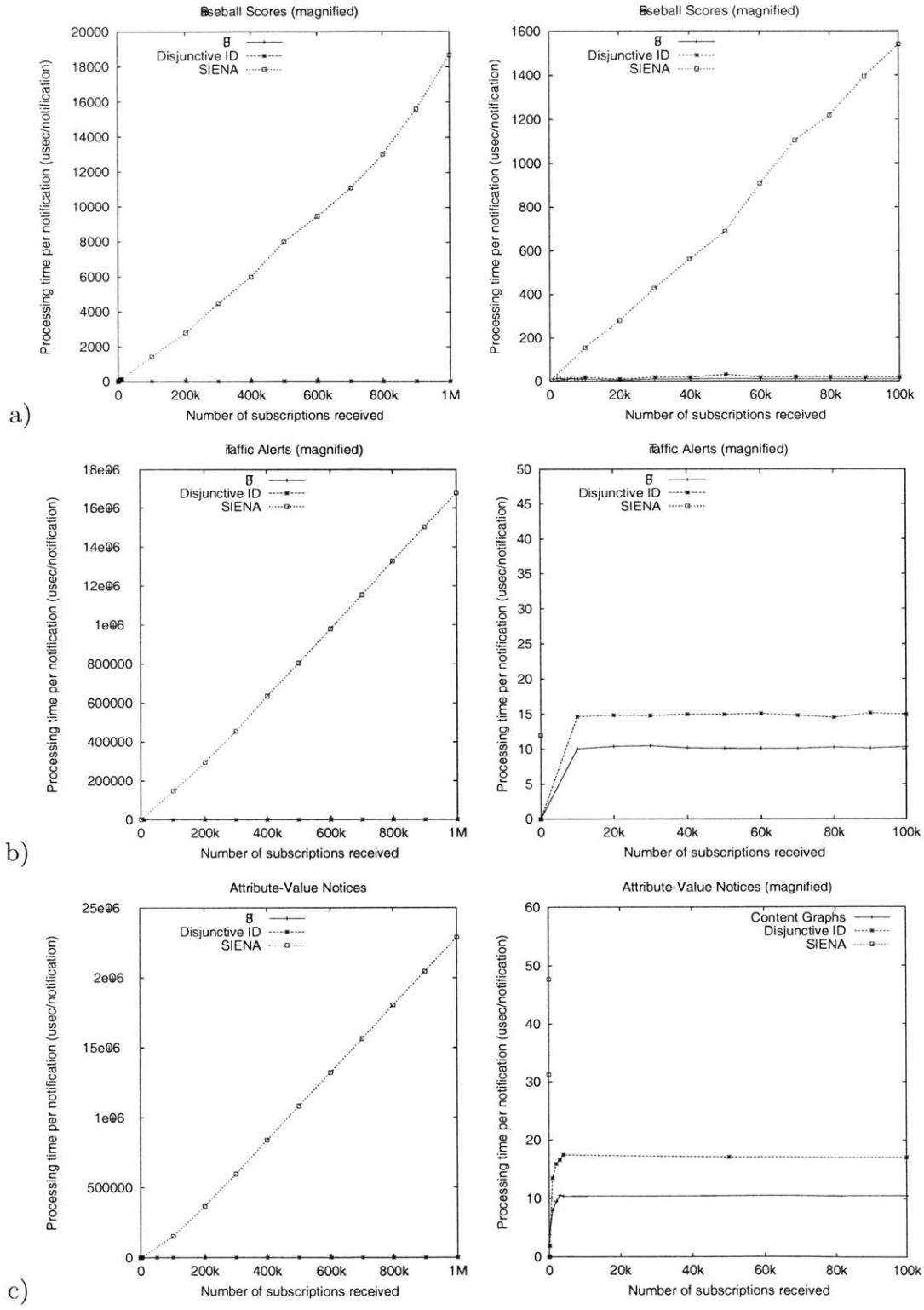


Figure 4-7: Processing times per notification (usec/notification) for 3 subscription systems when forwarding notices.

4.3 Study 3

Is the performance of F3 marginally superior to the performance of disjunctive ID systems, or is F3 greatly superior to disjunctive ID systems? Study 2 established that F3 outperformed disjunctive ID systems in one important area: subscription processing. But the size of gain available from F3 varied with scenario. For baseball and traffic alerts, F3 improved performance by more than an order or magnitude. In the generic attribute-value simulation, performance improvement was more modest.

This study was designed as a follow up of Study 2. It examined subscription processing times for F3 and disjunctive ID systems systematically. I varied the complexity of subscription topics and observed the effects of this variation on processing times of both F3 and disjunctive ID systems.

4.3.1 Method

The two subscription systems compared in this study were F3 and disjunctive ID multicast systems. These two systems were implemented on the working router developed for Study 2. Implemented in C++, the router ran on a RedHat Linux 8.0 operating system on a 2.8 GHz Intel processor with 1.2 GB of memory.

Simulation scenario

The simulation used in this study was similar to the generic attribute-value simulation used in Studies 1 and 2. The simulation modeled performance of a subscription service that responded to subscriptions formatted as attribute-value pairs. In Studies 1 and 2, the number of attributes used in this scenario was set at six and the number of values was set to five. In this study, the number of attributes varied between 7 and 10 in the experimental conditions. The number of values per attribute was set to 3 in all conditions. Values were randomly assigned to subscriptions, and all values were equally likely to occur. Individual attributes could also contain a wildcard. The probability of occurrence of a wildcard as an attribute-value was .3. Characteristics of this scenarios are summarized in Table 4.5.

Performance Measures

The three outcomes measured in this study were the same as those in Study 2: (a) the size of the forwarding table, (b) the total time that it took the router to process subscriptions, and (c) the total time for processing notifications.

4.3.2 Results

Figure 4-8 shows change in subscription-processing time, notification-processing time, and table size as a function of number of subscription attributes. These results are also summarized in Table 4.6. Each of the panels of Figure 4-8 shows increasingly divergent performance as the number of attributes increases from 7 to 10. In the top panel, forwarding table sizes grow from 500 KB to nearly 5000 KB for disjunctive ID systems, whereas the size of F3 forwarding tables remains constant at about 200 KB as number of attributes increases. The curves in the middle panel of the figure show results for per-subscription processing. The growth in subscription processing time is similar to the growth in table size for both disjunctive ID systems and F3. Subscription times rise sharply for disjunctive ID systems as subscription attributes increase; per-subscription processing times drop slightly for F3. With 10 subscription attributes, per-subscription processing times for disjunctive ID systems are more than 20 times the size of F3 times. The bottom panel of the figure gives per-notification processing times. The figure shows that per-notification processing time increases for disjunctive ID systems, whereas it decreases for F3.

The decreased efficiency of disjunctive ID systems with increases in subscription complexity is not hard to understand. In disjunctive ID systems, the total number of identifiers in a given subscription increases exponentially with the number of attributes in the subscription. Specifically, when a subscription contains n attributes, a single, wild-card subscription expands to 3^n disjoint identifiers. All 3^n identifiers must be inserted into the disjunctive ID table, and therefore the size of the disjunctive ID forwarding table grows exponentially with the number of attributes in the subscriptions. In contrast, the size of the F3 forwarding table remains constant. This is because the size of an F3 forwarding table depends primarily on the number of unique subscriptions in the table. This number is a constant in these experiments.

The long identifier lists in disjunctive ID systems affect subscription processing times. Because the number of disjoint identifiers per subscription grows exponentially with linear growth in the number of attributes, the time it takes disjunctive ID systems to process identifiers also grows exponentially with linear increases in number of attributes. In contrast, the amount of per-subscription processing time for F3 actually decreases with increases in the number of attributes. This decrease in per-subscription processing time is due to the decrease in subscription overlap with increasing numbers of attributes and a constant number of subscriptions. The less subscriptions overlap, the fewer edges in a graph, and the less time it takes F3 to process each subscription.

Finally, per-notification processing times increase for disjunctive ID systems but decrease for F3 with increases in the number of attributes per subscription. The increase in processing times for disjunctive ID systems is due to the exponential growth of table sizes with growth in attributes. The time it takes F3 to process notifications actually decrease with the number of attributes in each subscription. This decrease in processing time occurs because of a decreasing subscription overlap with decreasing numbers of attributes per subscription. As the number of attributes increases, each notification is destined for fewer subscribers, and it takes F3 less and less time to process each subscription.

4.3.3 Conclusion

This study showed that simulation conditions can affect simulation results. With simple subscription areas, disjunctive ID systems may seem competitive with F3. When subscription areas become more complex, however, the performance of disjunctive ID systems begins to deteriorate. Forwarding tables expand to enormous lengths, subscription processing becomes sluggish, and notification processing deteriorates. In contrast, F3 processing continues to be efficient as subscription areas become more complex. Its tables remain small, its subscription processing continues to be efficient, and it continues to process notifications quickly.

S	The number of subscribers	1000
r	The number of interfaces	10
N	The number of possible subscription topics	4^7–4^{10}
d	The depth of the graph	7–10
e	The degree of a node in the graph	3
m	The number of attributes per message	7–10
a	The number of possible attributes	7–10
v	The number of possible values	3

Table 4.5: Characteristics of scenario with varying numbers of attributes.

Subscription System	Attributes			
	7	8	9	10
Size of Forwarding Table (kB)				
Disjunctive ID	414.8	911.1	2235.2	4249.9
F3	142.6	132.7	127.2	120.0
Subscription Processing Time (usec/subscription)				
Disjunctive ID	165.3	308.6	516.0	932.3
F3	67.3	37.8	31.0	24.6
Notification Processing Time (usec/notification)				
Disjunctive ID	11.3	12.9	14.0	14.8
F3	6.4	5.1	4.7	3.9

Table 4.6: Performance of F3 and disjunctive ID systems when forwarding notices. The subscriptions in this scenario correspond to subscriptions with varying attributes.

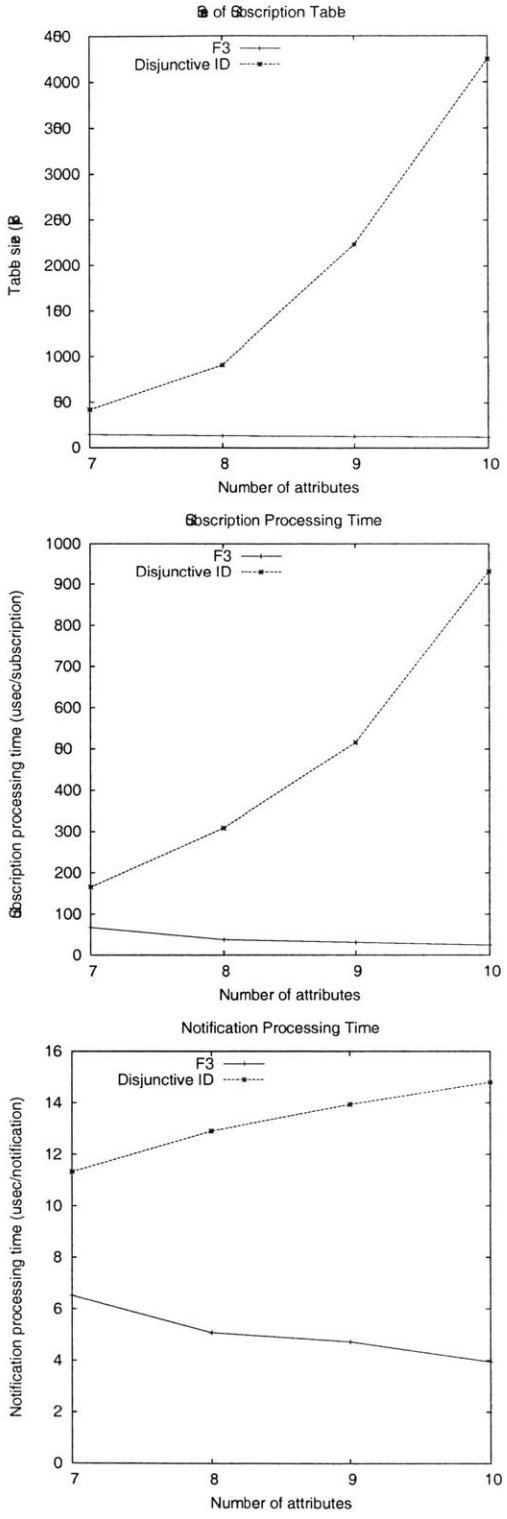


Figure 4-8: Performance of F3 and disjunctive ID systems when forwarding notices. The subscriptions in this scenario correspond to subscriptions with varying attributes.

4.4 Study 4

How important are content graphs to a subscription system like F3? Would F3 perform adequately if its message headers did not include content-graph representations of relevant subscription topics but instead contained only an unordered list of the relevant topics? This study provided empirical answers to these questions. The study examined performance of a modified version of F3 in which notification headers contained no information about interrelationships among subscription topics. My purpose in carrying out this study was to determine the importance of content graphs for efficient functioning of subscription system like F3.

4.4.1 Method

The working router developed for Studies 2 and 3 was used in this study. Implemented in C++, the router ran on a RedHat Linux 8.0 operating system on a 2.8 GHz Intel processor with 1.2 GB of memory.

Subscription systems

The two subscription systems examined in this study were F3 and a modified version of F3, called a content-list system. Content-list forwarding was described in Section 3.4.3 of Chapter 3. Content-list forwarding is a variant of content-graph forwarding, except that routers do not store edge information indicating the relationship between subscription identifiers. Whenever a source sends out a notification, it attaches a list of identifiers of subscription topics for which the notification is relevant. A router forwards a copy of the notification to one of its neighbors if that neighbor is subscribed to any of the identifiers listed in the header of the message. By comparing the performance of list-based systems to content-graph systems, we can measure the benefits that graph systems achieve by storing edge information at every router.

Simulation Scenarios

Two scenarios were developed for this study. Both scenarios called for systematic manipulation of topic overlap in the underlying content graph of the subscription area. Other aspects of each scenario were held constant. That is, in each condition of a scenario, number of subscribers, number of attributes, number of values, and wildcards were set at constant values.

The first scenario employed content graphs of varying degrees. In this scenario, there were 1000 subscribers. The number of nodes in the content graph was set at 100. The depth of the graph was set at 5 nodes, but the degree of nodes in the content graph varied from 1 to 50 in successive simulation trials. Figure 4-9 shows an example two content graphs used in this scenario. In the top panel, the degree of the graphs is 1; in the bottom panel, the degree is 2. The characteristics of this scenario are summarized in Table 4.7.

The second scenario employed content graphs of varying depths. In this scenario, there were 2000 subscribers. The number of nodes in the content graph was set at 1024. The degree of the graph was set at 2, and the depth varied from 1 to 10 in successive simulation trials. Figure 4-10 depicts two content graphs used in this scenario, one with a depth of 2 and one with a depth of 3. The characteristics of this scenario are summarized in Table 4.8.

Performance Measures

For each condition of each scenario, router performance was measured as the router filled subscriptions with 1000 notifications. The outcomes measured were the same as the ones measured in Studies 1 through 3: (a) the size of forwarding tables, (b) the time to process subscriptions; and (c) the time to process notifications. Each scenario ran 20 times and the results of each scenario were averaged together.

4.4.2 Results

Figures 4-11 shows the relationship between forwarding table size and graph characteristics for F3 and content list systems. The top panel illustrates the relationship between table size and degree, the bottom panel shows the relationship between table size and depth. The graphs show that content list table sizes remain constant, independent of the subscription complexity. The size of F3 tables are somewhat larger for F3 than for content list systems, and grow with the complexity of subscriptions.

Figure 4-12 show the relationship between graph characteristics and subscription processing times. Both panels show that processing times increase with complexity for F3 but not for content lists. The increase in processing times is especially clear in the top panel of the figure. With graph degree equal to 10, F3 subscription processing times are 16 times as high as content-list processing times.

Figure 4-13 presents results on notification processing. Both panels show similar results. As subscription overlap increases, notification processing time increases. The

growth in notification processing time is especially striking with increases in depth of graph. With graphs of depth 10, content-list processing times are 10 times as large as F3 processing times. The increase in notification times in content-list systems is due to the greater number of identifiers in content lists with high overlap in topics.

4.4.3 Conclusion

This study showed that performance differences of F3 and content list systems are a function of the complexity of subscription areas. In small subscription areas with little ordering of topics, content-list systems perform almost as well as F3 does. In areas with more topics and more ordering of topics, the picture changes. Content-list systems take much longer to process each notification, and F3 takes longer to process subscriptions.

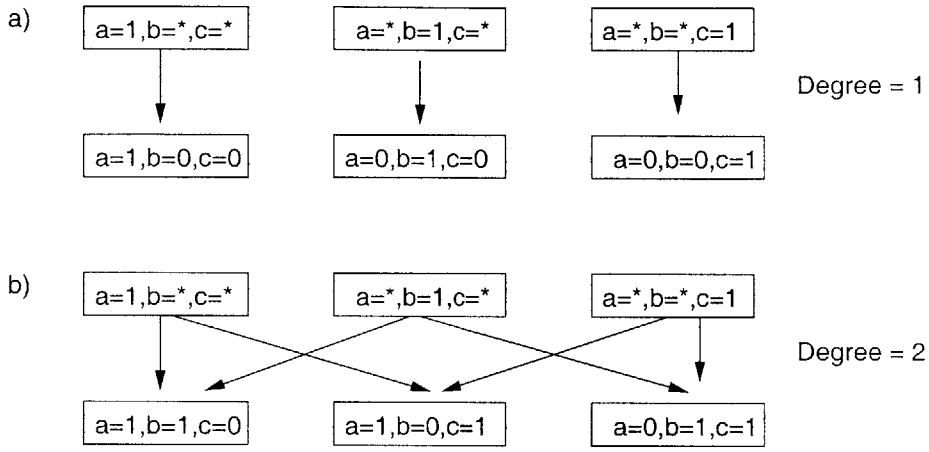


Figure 4-9: Two graphs in which the degree of each node varies, but all other aspects of the content graphs remains the same. The graphs depicted have a depth of 2.

S	The number of subscribers	1000
r	The number of interfaces	10
N	The number of subscription topics	100
d	The depth of the graph	5
e	The degree of a node in the graph	1–10
m	The number of attributes per message	100
a	The number of possible attributes	100
v	The number of possible values	2

Table 4.7: Characteristics of scenario with varying degree.

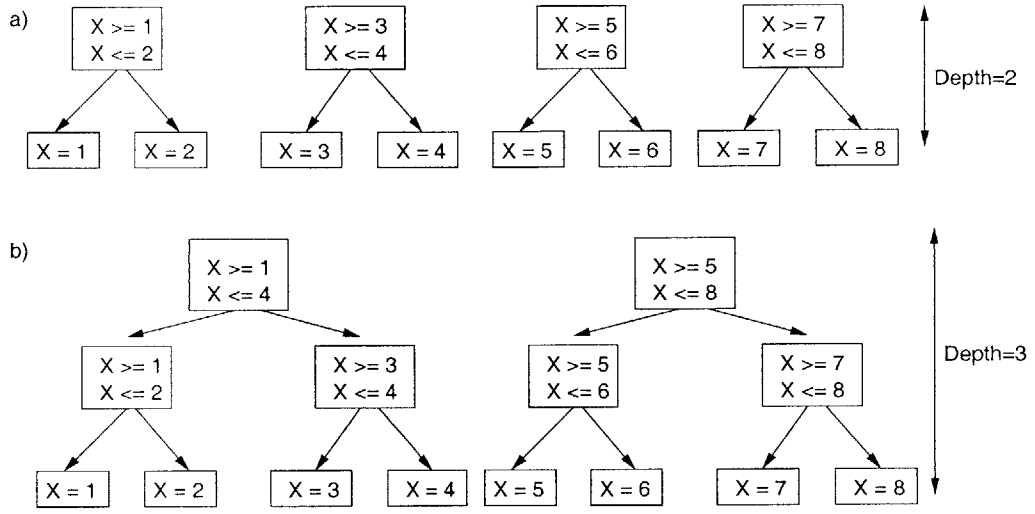


Figure 4-10: Two content graphs where the depth of the graphs vary, but all other aspects of the content graphs remain the same.

S	The number of subscribers	2000
i	The number of interfaces	10
N	The number of possible subscription topics	516
d	The depth of the graph	1-10
e	The degree of a node in the graph.	2
m	The number of attributes per message	1
a	The number of possible attributes	1
v	The number of possible values	516

Table 4.8: Characteristics of scenario with varying depth.

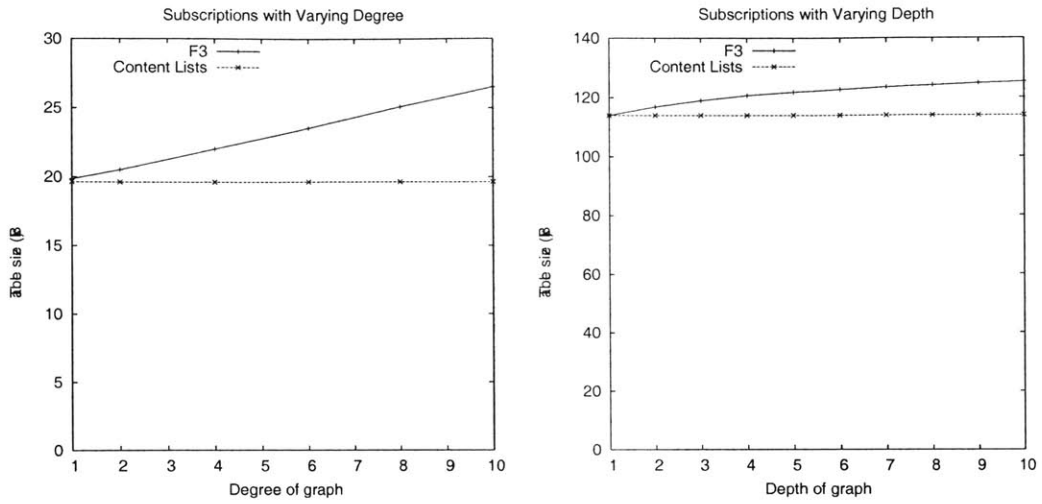


Figure 4-11: Table size of F3 and content list subscription systems when forwarding notices. The subscriptions in this scenario correspond to topics with varying complexity.

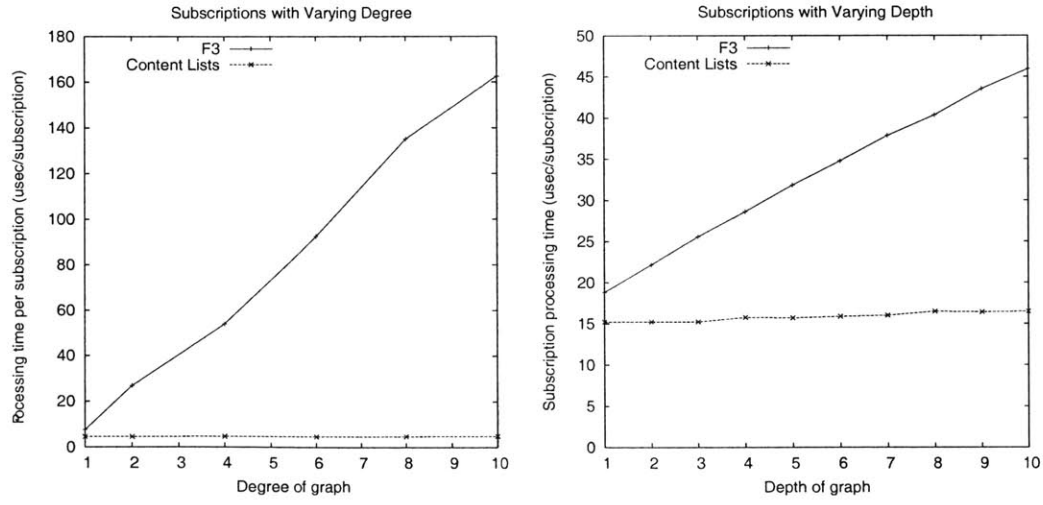


Figure 4-12: Subscription processing times of F3 and content list subscription systems when forwarding notices. The subscriptions in this scenario correspond to topics with varying complexity.

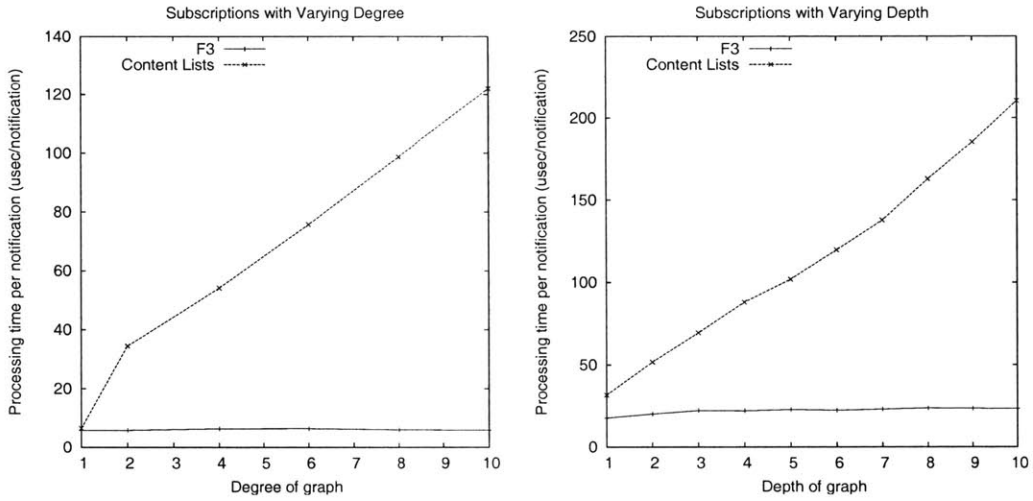


Figure 4-13: Subscription processing times of F3 and content list subscription systems when forwarding notices. The subscriptions in this scenario correspond to topics with varying complexity.

Chapter 5

Related Work

This chapter describes subscription systems developed during the past decade. Most of the systems can be classified into three categories: unicast, single-identifier multicast, and content-based multicast systems. This chapter describes systems in each of these categories and compares the systems to F3. A few lesser-known systems, including type-based, active subscription, and continuous query systems, are harder to categorize. This chapter describes these systems separately and also compares their features to features of F3.

5.1 Unicast Systems

In a unicast subscription system, subscribers send messages directly to a single information provider, and the provider enters the subscriptions into a central subscription server. Publishers also send publications to this server. When a server receives a publication, it looks in its database for addresses of relevant subscribers, and it uses the existing Internet infrastructure to forward the publication directly to these subscribers.

CORBA and TAO CORBA, a distributed object system, was one of the first software systems to support event-driven programming. With CORBA's event-driven interface, CORBA programs can ask to be notified of events produced by CORBA objects [34]. TAO is a commercial system that was specifically developed to distribute notifications to CORBA programs. TAO uses a high-performance, centralized server to distribute these notifications to subscribers in real-time [28].

GEM, Yeast, and MediaAS GEM, Yeast, and MediaAS are three research systems that use unicast message dissemination to deliver notifications to subscribers [36, 31, 27]. With these systems, subscribers can sign up to receive notices of both individual events and sequences of events, also called *compound events* or *composite events*. Yeast provides subscribers with the additional ability to attach *event-actions* to their subscriptions. An event-action is an action that a server takes whenever an event occurs that is relevant for a given subscription.

Web/Email Most Internet information services today use the web and email to handle subscriptions and notifications. An example of a web/email publish-subscribe system is the New York Times News Ticker [58]. Subscribers who are interested in receiving information about certain news topics can go to the New York Times website to sign up for alerts on news topics of interest. When the New York Times publishes news on the topic, it sends an email to relevant subscribers to alert them to appearance of the story. Other information services have set up similar systems on the Internet to dissemination information on weather, sports, traffic, consumer products, and other topics. Bulk emails take a long time to travel through the Internet, however, and like spam, they can clog up mail servers. This limits the utility of these web/email systems. These systems are ill-suited to delivery of time-critical information to large numbers of users.

HTTP Several systems have been proposed that use HTTP rather than email to disseminate notifications. In their proposal for internet-scale event-notification systems, Rifkin and Khare proposed the use of HTTP as a generic transport mechanism for notifications [47]. The Keryx system also uses HTTP to transport event-notifications [8]. Keryx applications use an XML-like language, called Transfer Syntax, to subscribe to and publish notifications.

Le Subscribe This is a unicast publish-subscribe system developed at INRIA that supports a content-based interface [43, 46]. In Le Subscribe, both subscriptions and notifications are expressed as a series of attribute-value pairs. A single Le Subscribe router delivers notifications to the correct subscribers by matching the attribute-value pairs in a given notification with the subscriptions that it has received. Pereira and his colleagues measured the speed of a Le Subscribe router and found that, like other content-based systems, Le Subscribe handled notifications sluggishly [46]. Per-notification processing times grew linearly as the number of subscribers in the system

increased.

Xfilter Like Le Subscribe, Xfilter is a unicast publish/subscribe system that provides users with a content-based interface [3]. In Xfilter, subscribers send subscriptions, expressed as Xpath attribute-value pairs, to a single Xfilter router. Notifications, expressed as XML documents, are sent to a single Xfilter router, which then delivers the notifications to appropriate subscribers. Altinel and Franklin [3] studied the performance of an Xfilter router and found that it too handled notifications slowly. Per-notice processing times for Xfilter, like those for LeSubscribe and content-based subscription systems, grew linearly as the number of subscriptions in the system increased.

TIBCO Rendezvous This is a commercial, unicast system specifically designed for high-speed, robust delivery of notifications to electronic commerce applications [59]. Goals of TIBCO developers included improving fault tolerance and increasing performance of subscription services. To achieve these goals TIBCO developers designed servers that consist of multiple, distributed machines.

JEDI The Java Event-Based Distributed Infrastructure (JEDI) is a publish-subscribe system for Java programmers [16]. Like Xfilter and Le Subscribe, JEDI subscriptions and notifications are expressed as a set of attribute-value pairs. Like TIBCO, JEDI uses multiple servers to distribute the load that would otherwise fall on a single server. JEDI has been successfully used to implement the OPSS work management system.

Elvin4 Elvin4, developed at the University of Queensland, is one of the few unicast systems to actively reduce message traffic [51]. In most unicast systems, publishers send messages to a a centralized server, even when no subscribers have signed up for notices on an event. Elvin4 the server solves this problem by providing *quench messages* to publishers when no subscribers have signed up for event notices.

Unicast systems and F3 Both unicast and F3 provide users with flexible interfaces. Unicast systems can have such interfaces because they employ a simple centralized architecture. Managers of subscription services can add new features to their services simply by changing software on a centralized server. F3 provides flexible interfaces using preprocessors. Like unicast systems, information providers can modify their interfaces simply by changing preprocessor software at the network edges.

F3 differs from unicast systems, however, in its ability to provide prompt notifications to large numbers of subscribers. Unicast systems are not designed to send out immediate notices to millions of subscribers. F3, which makes forwarding decisions at distributed routers, keeps message traffic to a minimum and thus avoids the network clogging that can occur in unicast systems.

5.2 Single-Address Multicast Systems

In a single-identifier multicast system, multiple routers cooperate to distribute notifications from information providers to subscribers. Subscriptions and notifications carry single, unique numeric identifiers, and routers use these identifiers to forward messages. The advantage of using a single-identifier multicast system to disseminate messages is efficiency. Multicast routers forward messages by performing only one hash-table lookup for one identifier on each message.

Herald Herald is a single-identifier multicast system that uses an overlay network of routers to disseminate messages to subscribers [9]. In Herald, as in F3, a single rendezvous point acts as the root of an overlay dissemination tree for each subscription identifier. Herald is unique in that its rendezvous points may consist of multiple, replicated servers. This allows Herald to distribute the load that would normally be placed on a a single rendezvous-point, and it also makes Herald a very robust system. Though F3 does not currently use multiple servers to implement rendezvous points, but F3 could be revised to incorporate this feature.

i3 and SCRIBE Like Herald and F3, the Internet Indirection Infrastructure (i3) [55] and SCRIBE [49] use overlay networks of dissemination trees to disseminate information. What makes these systems unique is their use of peer-to-peer networking to map subscription identifiers to rendezvous-points in the network. With this approach, each subscription identifier is assigned to a random rendezvous point in the overlay network, using a uniform hashing function. Whenever these a notification message arrives with a particular identifier, the system hashes the identifier and forwards the notification to the correct rendezvous-point in the peer-to-peer network.

Early Gryphon None of the single-identifier systems described so far was designed to handle the overlap in subscription topics that is common in real-world applications. Gryphon, developed at IBM, is one of the few single-identifier systems that does this.

Researchers at IBM were the first to define the IP multicast channelization problem and to prove that this problem is NP-hard to solve [1]. The early Gryphon system therefore used heuristics to map subscriptions, expressed as attribute-value pairs, onto subscription identifiers using preprocessors [6, 42]. Gryphon is also noteworthy because it is one of the few publish-subscribe systems that has been widely used. It has been used, for example, to deliver real-time sports scores for the US Tennis Open, the Ryder cup, The Australian Open, and the Sydney Olympics Internet. Gryphon has since evolved and instead uses content-based forwarding to forward notifications. Its current design is described further in Section 5.3.

Single-identifier Multicast Systems and F3 Of the systems described in this chapter, the early Gryphon system resembles F3 the most. Both systems use preprocessors to translate subscriptions from high-level subscription languages into low-level formats for forwarding. The low-level formats used in the two systems are different, however. In Gryphon, the preprocessor produces single identifiers, whereas in F3, the preprocessor produces content graph identifiers. In addition, the early Gryphon system, unlike F3, uses heuristics to assign identifiers to subscriptions and notifications. This means that Gryphon subscribers may receive false positives, or publications that they have not requested. It also means that Gryphon requires information about earlier subscription patterns for its heuristics. F3 does not use heuristics in assigning identifiers. It does not produce false positives, and it does not require information about earlier subscriptions to forward messages efficiently.

The overall architectural design of F3 and single-identifier multicast systems is clearly different. Single-identifier multicast routers use simple hash-tables to store subscriptions. F3 router hash-tables are more complex. Hash-tables in F3 routers store not only identifiers but also information about interrelationship among identifiers. In addition, records of F3 subscriptions and notifications may contain multiple identifiers, and so F3 routers may have to perform multiple hash-table lookups to forward a single notification.

As the empirical results presented in the previous chapter show, these architectural differences have a significant impact on the cost of these systems. Single-identifier multicast systems perform poorly in delivering notices with overlapping topics. In this case, single-identifier multicast systems either use excessive bandwidth or require large routing tables and incur high subscription costs. Though F3 content graphs are more complex to maintain, this extra complexity can account for a significant savings in terms of network bandwidth, table size, and subscription costs.

5.3 Content-based Multicast Systems

Subscribers to content-based systems submit subscription requests that describe the content that interests the subscriber. Content-based routers store subscription information, and whenever a router receives a notification message, the router checks to see whether the message is relevant for subscribers in its list. In most content-based systems, subscriptions and notifications are formatted as attribute-value pairs, which makes it possible for routers to match notifications systematically with subscriptions.

SIENA SIENA Developed at the University of Colorado, SIENA was one of the first subscription systems to use content-based forwarding [13, 12]. In SIENA, subscriptions and notifications are expressed as attribute-value pairs. Unlike F3 and many other multicast systems, SIENA publishers do not first send publications to a single rendezvous-point. As a result, SIENA subscriptions travel through the network to every potential publisher. Carzaniga and his colleagues found that the time it takes a SIENA router to forward a notification grows linearly with the number of subscriptions the router has received [12].

PreCache This is a commercial content-based system that supports subscriptions and notifications expressed as attribute-value pairs [45, 48]. PreCache routers use a proprietary, approximate-matching algorithm that achieves constant time and space complexity with respect to the number of subscriptions received. This means that PreCache subscribers may receive false positives, or publications that they did not actually request. The number of false positives that PreCache produces has not been made publicly available. PreCache resembles F3 in that both systems use preprocessors at the edges of the network to check subscription messages and manage the subscription namespace.

READY This is a content-based multicast system developed at AT&T [24]. Like subscriptions and notifications in content-based systems, READY subscriptions and notifications consist of attribute-value pairs. READY differs from other systems, including

F3, in its support of reliable service and adaptive routing. Like the GEM, Yeast, and MediaAs systems described previously, READY subscribers can sign up to receive sequences of events, also called compound events. Gruber and his colleagues found that the time it takes a READY router to forward a notification grows linearly with

the size of the notification.

Hermes Hermes is a content-based routing system developed at the University of Cambridge [44]. Hermes subscriptions and notifications, like those in other content-based systems, consist of attribute-value pairs. Like F3, Hermes uses dissemination trees to disseminate messages. However, in Hermes, all messages contain a special attribute, called its "type". Hermes assigns a rendezvous-point to each message by performing a hash on the type of the message. This approach is similar to the approach used in peer-to-peer networking systems like i3 and SCRIBE.

Gryphon Developed at IBM, Gryphon originally used single-identifier forwarding to disseminate publications, but Gryphon has over time evolved into a content-based forwarding approach [14, 61]. Gryphon subscriptions and notifications, like those in other content-based systems, consist of attribute-value pairs. Gryphon is able to forward messages faster than other attribute-value systems do. The time it takes a Gryphon router to forward a notification grows sub-linearly, rather than linearly, with the number of subscriptions the router has received [2].

Rebecca This is a content-based multicast system developed at the Darmstadt University of Technology [39]. It was developed as part of a larger project on event-based electronic commerce systems. Like F3, Rebecca is noteworthy because it enables users to define and support new types. Unlike F3, Rebecca achieves this goal within the architecture of a content-based routing system. Rebecca routers use an algorithm that is able to match values of a few base types such as integers and strings. To define a new value type, such as a Calendar date type, applications must supply Rebecca routers with a comparison operator for matching Calendar dates. Rebecca also differs from F3 in that, in F3, these new types would be incorporated to preprocessors, rather than at routers within the F3 system.

Content-Based Multicast Systems and F3. Content-based systems and F3 are similar in two important respects. First, both types of systems are multicast systems, that use distributed routers to forward messages to subscribers efficiently. Second, both types of systems are designed specifically to disseminate information on overlapping subscription topics. Important similarities between F3 and content-based systems stop here, however. Features of F3 that are not available in content-based systems include the following:

- F3 supports a variety of data formats, from uncompressed text data to private, compressed, audio-stream data.

- Using F3, it is possible for applications to detect message losses using sequence numbers.

- Information providers can change subscription formats for their services simply by modifying preprocessors at the network edge without making modifications to all the routers in the network.

- In F3, application-level message processing (such as parsing and type-checking) occurs once at the edge of the network, rather than repeatedly at every router along a message's path.

- In F3, the time it takes to forward a message does not depend on the length of message, the number of attributes in the message, or the number of values in the message. The time it takes a F3 router to forward a notification grows logarithmically with the number of subscriptions the router has received.

- However, F3 processes subscriptions more slowly than content-based systems do. Furthermore, the time it takes F3 to process subscriptions depends on the amount that subscription topics overlap with each other.

5.4 Other Systems

Unicast, single-identifier multicast, and content-based multicast may be the dominant approaches to subscription systems in use today, but they are not the only approaches in use or under development. This section describes type-based, active subscription, and continuous query systems.

5.4.1 Type-based Forwarding

Researchers at the Swiss Federal Institute have developed a subscription system, called DACE, that is specifically designed to support object-oriented applications [19, 20]. Like other multicast subscription systems, DACE uses a distributed system of routers to disseminate information to subscribers. DACE, however, uses a technique called type-based forwarding to disseminate this information. In type-based forwarding, subscribers and notifiers classify each message in an object-oriented, Java class

hierarchy. Routers, in turn, match subscriptions with notifications based upon relationships in the object-oriented hierarchy. DACE compilers process messages before they enter the DACE network, transforming them from complex Java expressions to simple type skeletons.

DACE resembles F3 in several ways. Both systems use preprocessors to extract forwarding information from messages. Both format this information using primitive data-structures rather than application-level formats. In both systems, these data-structures express the relationships between message categories. Routers in both systems use these data-structures to forward messages. As the next chapter discusses, it is, in fact, possible to build an object-oriented interface, like that used in DACE, on top of the F3 system. The primary difference between these systems and F3 is that F3 is designed to support a variety of application-level message formats, such as attribute-value pairs, and not geared toward a specific application language, like Java.

Active Subscription Systems Active subscription systems combine techniques developed for active networks [56] with subscription systems. In all the subscription systems discussed thus far, each subscription consists of a template or pattern, and routers forward notifications by matching these templates with notifications. In an active subscription system, each subscription is not just a passive template, but an executable program. It may contain function definitions, variable bindings, mathematical operations, procedural calls, conditional expressions, etc. Each program takes a notification packet as input and produces a value of yes or no as output. This value indicates whether the packet matches the subscription or not. The XML filtering system developed by Snoeren and his colleagues is an example of such a system [53].

The advantage of these systems is that they are much more flexible than their "passive" counterparts. For example, an active subscription can search a notification for the phrase Steve Jobs and only forward the notification if it occurs more times than the phrase Bill Gates. The disadvantage of active subscription systems is that active subscriptions can require a significant amount of router resources to execute. In fact, it is theoretically impossible to predict the amount of time or storage a given active subscription will take to process. It is also theoretically impossible for routers to sort active subscriptions. As a result, when a router receives a notification, it may have to check all of the subscriptions it has received against the notification.

An important question to ask is whether it is possible to implement an active subscription system on top of F3. The answer is both yes and no. As mentioned

previously, it is impossible for an application to sort active subscriptions statically into a partially ordered set. However, F3 can support programs where subscriptions are not statically sorted. In this approach, every active subscription would receive its own content-graph identifier, and the graph would not contain any edges. When a preprocessor received a notification, it would pass the notification to all of its active subscriptions. If a given active subscription indicated a positive match with the notification, the preprocessor would append the corresponding identifier to the header of the notification. Though this approach would work in theory, it might put such a burden on preprocessors as to make it practically infeasible.

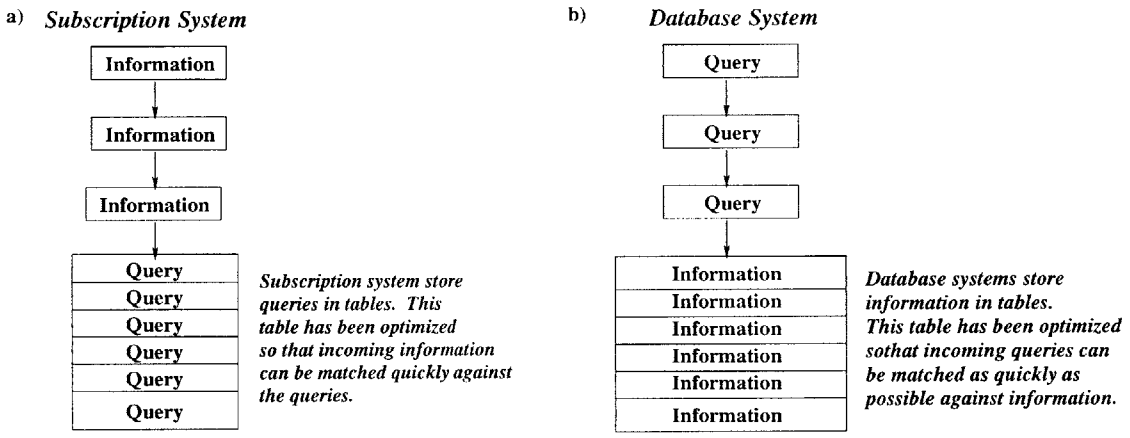


Figure 5-1: The difference between subscription systems (a) and database systems (b).

Continuous Query Database Systems Continuous query systems have recently become a hot topic of research within the database community [15, 33, 57]. Using a continuous query database, users can request to be notified whenever an item in a database has been updated. Users of such databases first submit special queries, called standing queries, to the database. When a database receives a standing query, it stores the standing query away. Then, when items in the database are updated, the database checks its standing queries against the updates and notifies users of relevant changes. Examples of continuous query systems that have been developed recently include NiagaraCQ [15] and OpenCQ [33].

One might think of a continuous query system as a special kind of subscription system, where standing queries play the role of subscriptions and database updates play the role of notifications. There are several differences between subscription sys-

tems and continuous query systems, however. First, subscription systems encompass a broad category of mechanisms for disseminating information updates to subscribers. Subscription systems are relevant to more than just database updates. Second, the design of current continuous query systems differs significantly from that of existing subscription systems, including F3. As several researchers have pointed out [60] [3], subscriptions and notifications typically play opposite roles in subscription systems and database systems. Figure 5-1 illustrates this difference. Subscription systems maintain a table of subscription queries, and this table is specifically designed so that routers can check incoming notifications, against their subscriptions as quickly as possible. Many subscription systems, like F3, also use multiple, distributed servers to reduce message traffic in the wide-area Internet. In contrast, a database system stores information, not queries, in its internal tables. These tables are designed so that the database applications can respond to incoming queries with the requested information as quickly as possible. Current database systems are not optimized to do the reverse, namely to check incoming information against a table of queries. Furthermore, database systems are not designed to reduce network traffic and typically do not use a topology of distributed servers to decrease traffic.

Chapter 6

Directions for Future Work

My goal in this project was to develop and evaluate an Internet subscription system that functions flexibly and efficiently. This dissertation establishes the groundwork for such a system, but there is still more work that can be done. This chapter describes ways in which future systems may be built on F3.

6.1 Access control

Subscription systems that control access to system resources have advantages over systems that do not incorporate access control. First, systems with access control can support applications that disseminate private data. Second, subscription systems that provide access control can prevent misuse of system resources. For example, subscribers can easily attack a system such as SIENA by making repeated requests for non-existent notifications. Such requests can fill up forwarding tables with bogus entries and slow movement of valid notifications to a standstill. Third, systems with access control can easily support commercial applications that charge for use of resources.

F3 currently provides some support for access control. First, F3 notification sources can encrypt data and prevent unauthorized subscribers from accessing it. Second, F3 can weed out requests for bogus subscription identifiers. When an F3 router receives a subscription to an unknown identifier, it sends a message to the rendezvous point for content-graph information on the identifier. The rendezvous point can easily reject requests for invalid identifiers. Finally, each F3 rendezvous points can prevent unauthorized notification sources from sending notifications by maintaining their own access lists and authenticating the sources.

F3 does not yet prevent unauthorized subscribers from submitting subscription requests with valid identifiers. When F3 routers receive subscriptions with valid identifiers, they enter the relevant subscription information into records of their forwarding tables without checking user authorizations. There are ways to remedy this problem. F3 could send an access-request message to each rendezvous point whenever it received a subscription. F3 could also refuse to accept a subscription unless it had been signed by a trusted preprocessor. More research is needed to determine the best way to limit access of F3 applications to authorized subscribers.

6.2 New Topologies

The F3 an overlay, dissemination-tree topology is currently manually generated There are many other ways to generate overlay topologies, however. F3 could use a routing algorithm, such as the distance-vector algorithm used in OSPF [38], to generate routing tables. F3 could also use an algorithm designed specifically for overlay networks. Some algorithms have recently been developed to take into account the properties of the overlay–such as the delay, reliability, and load of the underlying links [7]. F3 also does not have to use a dissemination tree topology to operate. Other proposed topologies use redundant routers, network links, and multi-path routing to decrease load on individual routers and increase system resilience [4, 54]. Although F3 does not incorporate such algorithms at present, such features can be incorporated into future versions of F3.

6.3 Faster Attribute-Value Preprocessors

The attribute-value preprocessor now used with F3 uses a simple algorithm for mapping attribute-value notifications to content-graph identifiers. When a preprocessor receives a notification, it starts at the root of the content-graph and performs a breadth-first search of the graph to find attribute-value pairs that match the notification. When the search reaches a node in the graph that does not represent a superset of the given attribute-value pair, searching stops on that branch of the tree.

This approach works relatively well for balanced graphs whose height equals the log of the number of leaves in the graph. But the approach works less well when content graphs are shallow and sparsely connected. When every node in a tree is a root, for example, this approach amounts to a linear search of all the roots of the

tree. However, there are several ways to improve this basic preprocessor algorithm:

- The preprocessor could sort all the nodes at every level in the graph by attribute and value. The preprocessor could then perform a binary search on the nodes at any given level in order to determine which nodes to search at the next level.
- The preprocessor could build an index, such as those used in a typical database, to quickly search the subscriptions in a tree. For example, the preprocessor could index the subscription table using an R-tree, and then use this R-tree to quickly look up notifications [25].
- The algorithm might incorporate features of fast matching algorithms that have been developed for content-based routing systems such as SIENA [12], Xfilter [3], and Le Subscribe [43, 46].
- The preprocessor could use an MD5 hash to hash every subscription and notification into a unique identifier. In order to look up a notification, the preprocessor would simply hash the notification into an identifier. This approach would work only for applications where notifications exactly matched subscriptions in the content-graph.

Whatever the approach, it is important to note that improvements to these preprocessors only affect the performance of applications at the edge of the network. They do not affect the forwarding performance of routers within the F3 system itself.

6.4 New Interfaces

F3 preprocessors currently support both subject-based and content-based interfaces. But F3 can also support other interfaces. For example, F3 is able to support type-based interfaces, similar to the DACE system [19, 20]. Applications that use this type of interface classify messages in terms of an object-oriented type-hierarchy. Before submitting messages to F3, subscribers using a type-based interface would send the messages to a special preprocessor, which would use the type-hierarchy to generate content graphs. Other possible interfaces for F3 include continuous-query databases. Applications that use continuous-query databases would submit special database requests for notification when particular items in the database change [15, 33, 57]. The database itself would partially-order these requests using content-graphs. Whenever an item in a database changed, the database would likewise use content-graphs to classify the change. One interesting question for future research is how internal database

indeces, such as R-trees, could be used to index into content-graph structures as quickly and efficiently as possible.

6.5 Off-line Subscription Processing

The results presented in Chapter 4 show that F3 subscriptions take much longer to process than F3 notifications. The consequence of this disparity is that F3 routers may be tied up processing subscriptions when there are urgent notifications to be delivered. One way to avoid this problem is to use separate processes to handle F3 routing, one for processing notifications and another for processing subscriptions. These processes could run on the same machine, with the subscription process scheduled to run at a lower priority than the notification process. They could also run on different machines, effectively ending any competition for resources between notification processing and subscription processing.

6.6 Adaptive Forwarding

Adaptive forwarding algorithms are a promising area development in subscription systems. An adaptive forwarding algorithm is a forwarding algorithm that uses a hybrid approach to adapt to changes in network conditions. For example, most multicast subscription systems use multicast forwarding to deliver notifications, even when there is only one subscriber in a network. In such a case, it would be much more efficient to deliver notifications using unicast methods. With adaptive forwarding, F3 might use unicast methods to disseminate notifications when there were only a few subscribers in a network and multicast methods when there were many subscribers. F3 might benefit from adaptive forwarding in its storage of content-graphs. When a content-graph is only a few levels deep, it might be more cost-effective for F3 to distribute notifications using content-list headers rather than content graph headers. With an adaptive approach to forwarding, a subscription system could take into account the efficiencies associated with various forwarding choices.

Chapter 7

Conclusions

Subscription systems help people retrieve new information from a network automatically, without their having to constantly query the network for updates. With subscription systems, users get exactly the information they need, whenever it becomes available, and they have to submit only one subscription request to receive the information. Examples of possible uses of subscription systems come from all parts of the digital world: news services, sensor systems, emergency management, business applications, distributed data repositories, multimedia entertainment, and even online games.

If subscription systems are ever to play a prominent role on the Internet, however, they must meet several requirements. First, subscription systems of the future must be able to support a large population of applications. Subscription services can be useful in almost any area involving electronic communication. Applications are almost endless in number. Second, subscription systems of the future must support extremely diverse applications. Applications involving news services and sensor systems, for example, are likely to be very different. And third, subscription systems of the future will have to handle a wide variety of complex subscription requests. Subscribers need sophisticated mechanisms for requesting and receiving exactly the information that they need.

This dissertation presented an analysis of previous work on unicast, single-identifier multicast, and content-based systems. The review of this work suggested that none of these systems completely meets these requirements. The design of unicast systems does not take into account the large number of subscribers that subscription systems must handle. In scenarios with millions of subscribers, unicast systems clog network links with notification messages. Single-identifier multicast systems are designed specifically to avoid this problem. However, these systems handle complex,

overlapping subscription categories very poorly. To handle overlapping subscription categories, single-identifier multicast systems would have to generate impossibly large routing tables or waste network bandwidth. Finally, content-based multicast systems have to replicate application-level machinery at every router in a system. As a result, content-based systems are costly to augment with new features and can be cripplingly slow in scenarios with large numbers of subscribers.

To overcome these difficulties, this dissertation proposed a novel subscription system, the Fast, Flexible, Forwarding system, or F3. Like other multicast subscription systems, F3 uses an overlay network of routers to distribute messages from information providers to subscribers. F3 differs from other approaches, however, in its use of preprocessors to analyze messages before routing begins. The preprocessors extract information about relationships between subscription and notification messages, and they disseminate this information to routers in the form of content graphs. Routers use the content-graph information to infer relationships between messages and to forward the messages to appropriate subscribers.

The idea of using preprocessors in subscription systems is not new. Researchers, however, have often overlooked some important benefits of preprocessing. First, preprocessors make subscription systems more efficient. This is because preprocessors ensure that application-level information is processed only once, before routing begins, rather than at every single hop along a message's path. Second, preprocessors make subscription systems more flexible. Using preprocessors, a subscription system can forward a variety of application-level messages, using only a single forwarding mechanism within the subscription system itself. In a sense, preprocessors provide insulation between F3 and applications. They allow application-level innovation and subscription-system innovation to occur independently so that neither applications nor subscription have to shift with the winds of the other's design. Finally, preprocessors help applications safeguard private data. They allow applications to communicate in a language that is appropriate for applications, and routers to communicate in a language that is appropriate for routers.

One cannot build a subscription system using preprocessors alone, however. The subscription system must forward messages, and its forwarding algorithm must be flexible enough to mesh with different preprocessor designs and efficient enough for high-speed message delivery. Content-graph forwarding was designed to meet both requirements. Content-graph forwarding differs both from single-identifier forwarding, where routers have too little information about message relationships, and from content-based forwarding, where routers have too much. With content graph for-

warding, routers can detect relationships among messages of various types, including attribute-value pairs, keyword ontologies, and object-oriented type-hierarchies. Because content graphs represent only relationships among messages, not their application-level details, F3 can process subscription and notification messages significantly faster than other systems do.

This dissertation provided experimental results from comparisons of performance of F3, unicast, single-identifier, and content-based forwarding systems. The results showed that F3 does not overproduce messages, as do unicast systems and single-identifier systems with overlapping message topics. Results also showed that F3 generates smaller tables and has lower subscription overhead than single-identifier systems with disjoint categories. Finally, experimental results also showed clear performance differences between F3 and SIENA, a content-based forwarding system. F3 took longer to process subscriptions than SIENA did for large numbers of subscribers in three different experimental simulations. F3 also generated much smaller tables than SIENA did. Most important of all, the time that it took an F3 router to process notifications exhibited logarithmic growth as the number of subscriptions in its table grew. As a result, for large numbers of subscribers, F3 processed notifications between 1,000 and 100,000 times faster than SIENA did in the three scenarios studied.

Although this dissertation proposes a promising, new approach to subscription systems, it describes only the first stage in the development of this approach. Though F3 was designed to support many kinds of preprocessors, the current implementation of F3 supports only a single type of preprocessor, which uses a simple algorithm to process subscriptions formatted as attribute-value pairs. An important task for the future is development of new preprocessors, with novel interfaces and faster preprocessing algorithms. F3 also uses content-graph forwarding as its sole means of forwarding notifications to subscribers. In situations where a variety of factors affect system performance, a multi-pronged approach to forwarding may be more appropriate. Adaptive forwarding algorithms, which employ different forwarding algorithms under different conditions, are therefore another promising area for development of F3.

In conclusion, Internet subscription systems may someday have a revolutionary impact on Internet communications. But the revolution will require new ideas and new technologies if it is to succeed. With preprocessors and content-graph data structures, F3 already provides greater flexibility and speed than other subscription systems do. F3 is only a small, first step toward the greater leaps that will help us accomplish this goal.

Bibliography

[1] M. Adler, Z. Ge, J. Kurose, D. Towsley, and S. Zabele. Channelization problem in large scale data dissemination. In *Proc. of the 9th IEEE International Conference on Network Protocols (ICNP)*, Riverside, CA, USA, November 2001.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, May 1999.

[3] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[4] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. *SIGCOMM Comput. Commun. Rev.*, 32(1):66–66, 2002.

[5] T. Ballardie, P. Francis, and Jon Crowcroft. Core Based Tree (CBT) An Architecture for Scalable InterDomain Multicast Routing. In *Proc. ACM SIGCOMM*, San Francisco, CA, 1993.

[6] Guruduth Banavar, Tushar Deepak Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *International Conference on Distributed Computing Systems*, pages 262–272, 1999.

[7] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217. ACM Press, 2002.

[8] S. Brandt and A. Kristensen. Web push as an internet notification service. In *W3C Workshop on Push Technology*, Boston, Massachusetts, September 1997.

[9] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a global event notification service. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[10] A. Carzaniga, D. Rosenblum, and A. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Symposium on Principles of Distributed Computing*, pages 219–227, 2000.

[12] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

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

[14] IBM TJ Watson Research Center. Gryphon: publish/subscribe over public networks. Technical Report.

[15] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390. ACM Press, 2000.

[16] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[17] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. An architecture for wide-area multicast routing. In *Proc. ACM SIGCOMM*, pages 126–135, London, UK, September 1994.

[18] S. Duarte, J. Legatheaux Martins, H. J. Domingos, and N. Preguica. A case study on event dissemination in an active overlay network environment. In *In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003.

[19] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *ECOOP*, pages 252–276, 2000.

[20] P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. Technical Report EPFL, DSC ID:2000104, January 2001.

[21] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[22] Google News Alerts. `http://www.google.com/newsalerts`.

[23] J. Gough and G. Smith. Efficient recognition of events in distributed systems. In *Proceedings of 18th Australasian Computer Science Conference (ACSC)*, February 1995.

[24] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

[25] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. In *Proc. of ACM SIGMOD Int. Symp. on the Management of Data*, pages 45–57, 1984.

[26] T. Hardjono and B. Weis. The multicast security architecture. Internet Draft, 2003.

[27] A. Hinze and D. Faensen. A unified model of internet scale alerting services. In *Proc. of the International Computer Science Conference*, Hong Kong, December 1999.

[28] Object Computing Inc. The ace orb (tao). `http://www.theaceorb.com`.

[29] Sneha Kumar Kasera, Gísli Hjálmtýsson, Donald F. Towsley, and James F. Kurose. Scalable reliable multicast using multiple multicast channels. *IEEE/ACM Transactions on Networking*, 8(3):294–310, 2000.

[30] Eddie Kohler. *The Click modular router*. PhD thesis, Massachusetts Institute of Technology, November 2000.

[31] B. Krishnamurthy and D. S. Rosenblaum. Yeast: a general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[32] Joanna Kulik. Fast and flexbile forwarding for internet subscription systems. In *In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003.

[33] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[34] C. Ma and J. Bacon. Cobea: A corba-based event architecture. In *USENIX COOTS'98*, pages 27–30, Santa Fe, NM, April 1998.

[35] Major League Baseball. `http://www.mlb.com`.

[36] M. Mansouri-Samani and M. Sloman. Gem a generalized event monitoring language for distributed systems. *IEE/BCS/IOP Distributed Systems Engineering*, 4(2):96–108, June 1997.

[37] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, pages 99–114, August 1999.

[38] J. Moy. OSPF version 2. RFC 1583, IETF, March 1994.

[39] Gero Mühl. Generic constraints for content-based publish/subscribe systems. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 211–225, Trento, Italy, 2001. Springer-Verlag.

[40] University of Colorado Software Engineering Research Laboratory. Siena fast forwarding. `http://www.cs.colorado.edu/ carzanig/siena/forwarding/`.

[41] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Middleware*, 2000.

[42] L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[43] João Pereira, Françoise Fabret, François Llirbat, and Dennis Shasha. Efficient matching for web-based publish/subscribe systems. In Opher Etzion and Peter Scheuermann, editors, *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, volume 1901 of *LNCS*, Eilat, Israel, 2000. Springer-Verlag.

[44] Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, July 2002.

[45] PreCache. `http://www.precache.com`.

[46] R. Preotiuc-Pietro, J. Pereira, F. Llirbat, F. Fabret, K. Ross, and D. Shasha. Publish/subscribe on the web at extreme speed. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Cairo, Egypt, 2000.

[47] A. Rifkin and R. Khare. The evolution of internet-scale event notification services. `http://www.cs.caltech.edu/ adam/isen/wacc/`, August 1998.

[48] D. Rosenblum. Some open problems in publish/subscribe networking. In *In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003.

[49] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[50] Scottrade. `http://www.scottrader.com`.

[51] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proceedings AUUG2K*, Canberra, Australia, June 2000.

[52] The Network Simulator. `http://www.isi.edu/nsnam/ns/`.

[53] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using xml. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[54] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th ACM International Conference on Mobile Computing and Networking (Mobicom 2000)*, August 2000.

[55] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM 2002*, August 2002.

[56] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 26(2):5–17, 1996.

[57] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330. ACM Press, 1992.

[58] The New York Times News Tracker. `http://www.nytimes.com`.

[59] TIBCO Software Inc. `http://www.tibco.com`.

[60] T. Yan and H. Garcia-Molina. SIFT—A tool for wide-area information dissemination. In *Proc. 1995 USENIX Technical Conference*, pages 177–186, New Orleans, 1995.

[61] Yuanyuan Zhao and Rob Strom. Exploitng event stream interpretation in publish-subscribe systems. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 219–228. ACM Press, 2001.

Appendix A

Theoretical Evaluation

This chapter describes various aspects of F3's performance in terms of their theoretical bounds. It also compares F3's performance to that of content-list, disjunctive-Id, and SIENA.

A.1 Forwarding Table Size

A variety of factors contribute to the performance characteristics of F3. A list of all possible factors is given in Table A.1. The size of an F3 forwarding table depends upon n, the number of unique subscriptions received by the router:

$$\text{Forwarding Table Size} = O(n * \text{Descriptor size})$$

Each node's descriptor consists of several components, including a list of edges, a list of direct subscribers to the node, and a list of output-labels for the node:

$$\text{Descriptor Size} = \text{Edges} + \text{Subscribers} + \text{Output-labels}$$

The number of edges depends directly on e, the degree of the graph and the number subscribers depends directly on s, the number of subscribed interfaces:

$$\begin{aligned} \text{Edges} &= O(e) \\ \text{Subscribers} &= O\left(\frac{i * s}{n}\right) \end{aligned}$$

An interface can only appear as an output-label if it is not directly subscribed to a particular node. Each of these output-labels may include one or more node-Ids.

The maximum number of node-Ids per output-label is, at most, the width of the graph. The probability that a node has an output label is proportional to how many ancestors it has. This yields:

$$\begin{aligned}
\text{Output-labels} &= O(\text{Number of labels} * \text{Size of label}) \\
\text{Number of labels} &= O(\text{Subscribed ancestors} * \text{Unsubscribed Interfaces}) \\
\text{Subscribed ancestors} &= O\left(\frac{s}{n} * de\right) \\
\text{Unsubscribed interfaces} &= O\left(i - \frac{is}{n}\right) \\
\text{Size of label} &= O(e)
\end{aligned}$$

Together, these equations give:

$$\begin{aligned}
\text{Forwarding Table Size} &= O\left(n * \left(e + \frac{is}{n} + i\frac{sde}{n}\left(1 - \frac{s}{n}\right)\right)\right) \\
&= O(ne + is + isde\frac{n-s}{n})
\end{aligned}$$

When the number of subscriptions per interface is small, or when $s \ll n$, then we expect the size of the forwarding table to grow with:

$$\text{Forwarding Table Size} = O(ne + isde)$$

As the number of subscriptions per interface grows large, or as s approaches n, then the size of each output label is governed by:

$$\text{Forwarding Table Size} = O(ne + is)$$

Table A.2 compares the theoretical characteristics of F3 to other systems. Content-list systems have the smallest forwarding tables of all the systems studied. Content-graph systems have the next largest tables, differing from content-lists by a factor of e, the number of edges in the table. Disjunctive-Id systems and content-based systems have the largest tables. The size of disjunctive-Id tables depends upon the total number of possible subscription categories, N. The size of a SIENA table depends upon the total number of subscriptions received, S. If there are few subscription categories, but many subscribers, one would expect disjunctive-Id systems to have smaller tables. If there are many subscription categories but few subscribers, one would expect content-based systems to have larger tables.

Variable	Factor
S	The number of subscriptions received.
N	The total number of possible unique subscriptions. Subscriptions may be overlapping.
n	The number of unique subscriptions received. Subscriptions may be overlapping.
i	The number of interfaces
s	The number of unique subscriptions per interface.
d	The depth of the graph
e	The degree of a node in the graph

Table A.1: Factors affecting subscription system performance

Characteristic	Content-graph	Content-list	Disjunctive-Id	SIENA
Table size	$O(ne+is)$	$O(n+is)$	$O(2^N i)$	$O(S)$

Table A.2: Theoretical forwarding table sizes of 4 subscription systems.

A.2 Subscription Processing Time

When a router receives a new subscription, it must look up the subscription in the table. If the subscription is not found, it must add a new subscription to the table. If the subscription is found, then the router must also label all of the subscribers descendants. The time it takes to process a subscription is then:

$$\begin{aligned}
\text{Subscription Time} &= O(\max(\text{Table Insertion Time}, \text{Label Descendants Time})) \\
\text{Table Insertion Time} &= O(\log n) \\
\text{Label Descendants Time} &= O(\log n) + \text{Number of Descendants} * O(\log i) \\
\text{Number of Descendants} &= O(d * e) \\
\Rightarrow \text{Subscription Time} &= O(\log n + (d * e)\log i)
\end{aligned}$$

These equations indicate that the shape of a content-graph can play a significant role in the amount of time that it takes a router to set up subscriptions. If the router maintains a graph that is wide and deep, than the number of descendants per node in the graph will be large. In this case, one would expect the time it takes to process each subscription to depend mostly on $d * e$, the number of descendants per node in the graph.

As characterized by Table A.3, content-based systems such as SIENA process

subscriptions the fastest, in constant time. Content-list systems also process subscriptions. The time it takes content-list systems to process notifications grows logarithmically with the size of the forwarding table. Content-graph systems process subscriptions more slowly. They take a factor of $(d * e)$ more time to process subscriptions than content-list systems. Finally, disjunctive-Id systems take the most time to process subscriptions. In a disjunctive-Id system, each subscription can contain many identifiers, which grows exponentially with the total number of subscription topics, N.

Characteristic	Content-graph	Content-list	Disjunctive-Id	SIENA
Subscription time	$O(\log n + de \log i)$	$O(\log n + \log i)$	$O(2^n)$	O(1)

Table A.3: Theoretical subscription processing times of 4 subscription systems.

A.3 Graph Setup Time

When a router receives a graph-setup message, it must add the edges indicated by the setup message to its graph and fix up the output-labels in the graph. The time it takes a router to process a graph-setup message is then:

$$\begin{aligned}
\text{Graph Setup Time} &= \text{Add Edges Time} + \text{Label Descendants Time} \\
\text{Add Edges Time} &= O(\log n) + O(e) \\
\text{Label Descendants Time} &= O(\log n) + \text{Number of Descendants} * O(\log i) \\
\text{Number of Descendants} &= O(d * e) \\
\Rightarrow \text{Graph Setup Time} &= O(\log n + (d * e) \log i)
\end{aligned}$$

Like subscription messages, the amount of time it takes a router to set up a graph message depends largely on the shape of the router's content graph.

A.4 Notification Processing Time

When a router receives a notification, it must look up the node-Ids in the notification's header, find their output-labels, and output a copy of the notification to the appropriate interfaces. The time it takes a router to process a notification is then:

$$\text{Notification Processing Time} = \text{Lookup time} + \text{Output time}$$

$$\begin{aligned} \text{Lookup time} &= O(\log n) \\ Outputtime &= O(i) \\ \Rightarrow \text{Notification Processing Time} &= O(\log n + i) \end{aligned}$$

As summarized in Table A.4, F3 takes less time to process notifications than all other systems. The cost of processing a notification in F3 is essentially the cost of performing a single hash-table lookup. Disjunctive-Id systems, content-list systems, and content-based systems take more time to process notifications than F3. The time it takes a disjunctive-Id system to process a notification depends on N, the total number of possible subscriptions. Content-lists, on the other hand, take a factor of of $d * e$ longer to process notifications than content graphs. Content-based systems generally take the most time, where the amount of time it takes to process the notification grows linearly with the total number of subscriptions received.

Characteristic	Content-graph	Content-list	Disjunctive-Id	SIENA
Notification time	$O(\log n + i)$	$O(de(\log n + i))$	$O(N + i)$	$O(S)$

Table A.4: Theoretical notification processing times of 4 subscription systems.

A.5 Resubscription Processing Time

Every F3 router periodically checks the entries in its table to check for expired subscriptions and to generate resubscription messages. If the number of nodes in the forwarding table is, n, then the cost of processing a subscription is therefore $O(n)$.

A.6 Unsubscription Processing Time

A router performs three tasks whenever it receives an unsubscription. It removes the subscription from the node indicated in the unsubscription, it fixes the labels for all of the node's descendants, and removes the node from the table, if necessary. The time it takes a router to process an unsubscription is then:

$$\begin{aligned} \text{Unsubscription Time} &= \text{Lookup Time} + \text{Label Removal Time} + \\ &\quad \text{Graph Deletion Time} \\ \text{Lookup Time} &= O(\log n) \end{aligned}$$

$$
\begin{aligned}
\text{Label Removal Time} &= \text{Number of Descendants} * O(\log i) \\
\text{Number of Descendants} &= O(d * e) \\
\text{Graph Deletion Time} &= O(\log n) + O(d) \\
\Rightarrow \text{Unsubscription Time} &= O(\log n + (d * e) \log i)
\end{aligned}
$$

Like subscription and graph-setup messages, the amount of time it takes a router to set up a graph message depends largely on the shape of the router's content graph.