UNIVERSITE DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
Ecole Doctorale STIC - Sciences et Technologies de l'Information et de la Communication

THESE

pour obtenir le titre de

Docteur en Sciences

de l'UNIVERSITE de Nice-Sophia Antipolis

Discipline: Informatique

présentée et soutenue par
Raphael CHAND

Large scale diffusion of information in Publish/Subscribe systems

These dirigée par **Ernst BIERSACK**
co-dirigée par **Pascal FELBER**
soutenue le 22 Septembre 2005

Jury:

Prof. Roberto BALDONI, rapporteur

Prof. Pierre SENS, rapporteur

Prof. Michel RIVEILL, examinateur

Prof. Pascal FELBER, examinateur

Prof. Ernst BIERSACK, examinateur

Acknowledgements

I would like to thank my advisor Prof. Pascal Felber for his invaluable help and advice. He trusted me, supported me, and encouraged me throughout my PhD studies. He taught me a great deal of knowledge and skills in various areas of research, from programming to the art of writing scientific articles. Thanks to him, I developped a real taste for research.

I am also thankful to Prof. Refik Molva for his constant support throughout my studies at Institut Eurecom, Prof. Ernst Biersack who enabled me to do this PhD, and Institut Eurecom for providing me with excellent working conditions.

Furthermore, I am grateful to Prof. Pierre Sens, Prof. Roberto Baldoni and Prof. Michel Riveill, who kindly accepted to be part of the jury.

Last but not least, I would like to thank Carlinha for bearing me and for her infinite patience, all my friends and, of course, my family for her constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"The Internet has transformed and revolutionized the communications world like nothing before it. Although previous technologies allowed the transmission of information between locations, the Internet took the best features of the diversity of communications technologies out there and combined them into a unique and synergistic whole. It can not only broadcast and disseminate information world-wide, but it can allow people working together at a distance to collaborate and interact in real time" [1]. Possibilities are increasing even more with the widespread diffusion of high-bandwidth links and the development of new mobile communication systems such as next generation mobile phones, wireless laptops or personal digital assistants.

Most current large-scale computer systems are based on synchronous client/server communications (,e.g., CORBA [63], [77]). In the client/server paradigm, clients send requests to servers. Servers reply to clients' requests. Most importantly, the communication between a client and a server is synchronous in that a client is blocked from the time it has issued a request until it has received the corresponding reply.

RPC (remote procedure call) [65, 79] and more recently object-oriented successors like RMI (remote method invocation) [78] or Microsoft DCOM [76, 90] are the most common techniques for constructing distributed applications, due to their simplicity and ease of use. Both are based on the synchronous client-server model and extend the notion of local procedure calling so that a remote procedure call is almost identical.

The major drawbacks of RPC-like platforms, and more generally of the client/server model, are twofold. First, the client must address its request to a particular server. Second, the server must process the request and send the reply to the client, which is blocked until it receives it. Hence, the components of the system are tightly coupled, in the sense that a piece of information must be addressed to a specific destination. Also, multiple demands from a client must be executed either sequentially, or in error-prone multi-threaded requests. In addition to that, the Internet has considerably changed the scale of distributed systems. Distributed systems now involve a great number of participants (like hundreds of thousands), which may be widely dispersed geographically and with different resources (wired, wireless) and behaviors.

For those reasons, many applications cannot be built with the classical abstractions on which distributed systems have been built until now, since they lead to rigid and static applications and make the development of dynamic large scale applications cumbersome.

Several solutions are proposed, that are more appropriate for the diffusion of information in large-scale distributed systems. Some are based on network-level technologies such as IP multicast. Multicast is an important service for improving the efficiency and robustness of distributed systems and applications. However, there are several issues that have limited the commercial deployment of IP multicast since its introduction [52]. Some of these issues include: group management, distributed multicast address allocation, flow control and security, and support for network management. For those reasons, solutions based on IP multicast often yield to static and cumbersome applications. More recently, other solutions were proposed, that are based on peer-to-peer overlay network infrastructures [113, 126, 104] or epidemic multicast algorithms [56].

A new communication paradigm called publish/subscribe (pub/sub) has been proposed recently and has received increasing attention. It addresses many issues raised by current distributed applications. Most importantly, it is claimed to provide the loosely coupled form of interaction required in large scale dynamic distributed systems.

1.1 Publish/Subscribe systems

In pub/sub systems, there are no clients or servers, but rather consumers and producers. Producers produce pieces of information, called events, which are consumed by consumers. Consumers have the ability to express their interest in an event, or a pattern of events, and are subsequently notified of any event, generated by a producer, which matches their registered interest. Hence, in a publ/sub system, the receivers of information, the consumers, are not directly targeted by a sender, i.e. a producer, but are rather indirectly addressed according to the interests that they registered and the content of events. The notification service realizes the interaction between producers and consumers and handles the dissemination of information from producers to consumers. A Producer publishes an event to the notification service without addressing it to any particular consumer. The event is then propagated to all consumers that registered interest in that given event, in an asynchronous manner, that is, allowing the consumers not to be blocked waiting for events to arrive but instead to perform concurrent operations.

The publ/sub paradigm offers numerous advantages over RPC-based distributed middleware: due to its asynchronous nature, the system is potentially more scalable and can work in "disconnected" mode (e.g., for mobile users and wireless devices). Independence of application-level interfaces permit integration of heterogeneous components from different sources. Indirect addressing makes it possible for the infrastructure to implement reliability, load balancing, fault-tolerance, persistence, or transactional semantics. Besides, by simplifying business integration and making the system potentially cheaper to operate, faster, and less prone to errors, the pub/sub paradigm is also a cost-effective solution for the industry. Most importantly, the strength of the pub/sub paradigm lies in the full decoupling in time, space and synchronization that it realizes between consumers and consumers.

1.2 Focus and contribution of the thesis

In this thesis, we focus on the achievement of large scale diffusion of information in pub/sub systems. The sub/sub paradigm has become a hot research topic in the last few years, because the strong decoupling that it offers between the different participants makes it well adapted to large scale distributed information systems.

This thesis focuses on the study, the design and the implementation of pub/sub systems that achieve scalable and efficient diffusion of information. The contributions of this thesis can be divided in three major parts.

The XNet XML Content Network. We present the architecture of the XNet XML content network that we designed to implement efficient and reliable distribution of structured XML content to very large populations of consumers. For that purpose, our system integrates several novel technologies: the routing protocol, XRoute, makes extensive use of subscription aggregation to limit the size of routing tables while ensuring perfect routing (i.e., minimizing inter-router traffic). The filtering engine, XTrie, uses a sophisticated algorithm to match incoming XML documents against large populations of tree-structured subscriptions, while the XSearch subscription management algorithm enables the system to efficiently manage large and highly dynamic consumer populations. Finally, our XNet system integrates *reliability* mechanisms to guarantee that its state is consistent with the consumer population and implements several approaches to fault-tolerance to recover from various types of router and link failures. We have performed an extensive performance evaluation

of our system. We have analyzed its efficiency by means of simulations in various settings and, to experiment with the conditions of the real Internet, we have performed a large scale deployment in the PlanetLab testbed. Experimental results demonstrate that XNet does not only offer very good performance and scalability under normal operation, but can also quickly recover from system failures.

Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks. The second contribution of this thesis was motivated by some of the limitations of traditional pub/sub systems based on server overlays. First, they are usually based on a fixed infrastructure of reliable brokers which cannot easily be modified or extended as the population of the producers and consumers evolves. Furthermore, the challenging task of routing the messages based on their content remains a complex and time-consuming operation and often provides results that are just barely better than a simple broadcast. Consequently, we present a novel approach to content routing that was designed to specifically address these issues. The producers and consumers are organized in a peer-to-peer network that self-adapts upon peer arrival, departure, or failure. Most importantly, our pub/sub system features an extremely simple routing protocol that requires almost no resources and no routing state to be maintained at the peers. The price to pay for this simplicity is that routing may not be perfectly accurate in the sense that some peers may receive some messages that do not match their interests (false positives), or fail to receive relevant messages (false negatives). However, by organizing the peers in "semantic communities", i.e., by organizing them according to their interests with adequate proximity metrics, we can minimize the routing inaccuracy. We propose a containment-based proximity metric that allows to build a bandwidth-efficient network topology that produces no false negatives and very few false positives. We have also developed a proximity metric based on subscription similarities that yields a more solid graph structure with negligible false negatives and very few false positives. Experimental results demonstrate that the routing process is indeed very accurate and highly efficient, and that our system features excellent scalability to large consumer populations, both, in terms of routing and peer management overhead.

Similarity-based Proximity Metric for XML Documents and XPath Subscriptions. Finally, we present the proximity metric based on subscription similarities that we developed to organize the peers in the aforementioned system in an efficient graph structure. The metric evaluates the proximity between two given subscriptions in terms of filtering, i.e., it evaluates the error that would be induced by filtering XML documents against the other subscription instead of the original. Our proximity metric can operate in various conditions, that is, it adapts to the amount of knowledge that we have about the data. More precisely, if the *type* of the data or a *history* of previous documents is known, the proximity metric exploits this knowledge to improve accuracy. In particular, a major innovation is that if the type of the data is known, then the proximity metric makes use of the *correlations* between elements that are defined in it, so as to improve correctness. Performance evaluation shows that the proximity obtained by our metric between subscriptions is highly accurate, in that it helps minimizing the filtering error, and consistent, in that it corresponds indeed to the observed filtering error. Also, note that although used in the context of pub/sub systems, and designed for XML documents and XPath subscriptions, our proximity metric can be easily extended to other tree-structured languages, and can be used to address different data management problems.

Chapter 2

The Publish/Subscribe paradigm

2.1 Introduction

In this chapter, we describe the Publish/Subscribe (pub/sub) communication paradigm. A formal definition of a pub/sub system is given. We describe and explain the role of each of its components, and various aspects of a pub/sub system. In particular, the subscription model that is used to specify the subscribers' interest with regards to the information that transits in the system is of great importance, since it offers various degrees of expressiveness and highly influences the system overall. Also, the architectural model on which the system is implemented has great consequences on the scalability and the dynamicity that are allowed in a wide scale distributed environment. We then explain in more details the main functionalities of a pub/sub system, which enable it to achieve wide-scale dissemination of information. This allows us to isolate the issues that are raised and the challenges that remain in pub/sub systems. In the final part of the chapter, we survey the most representative existing pub/sub systems.

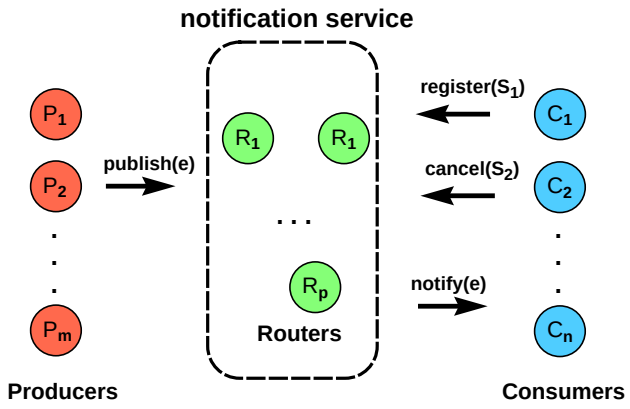


Figure 2.1: High level view of a pub/sub system.

2.2 Definition

2.2.1 Elements of a Publish/Subscribe system

A pub/sub system consists of three major components:

- The *producers*, also called *publishers*, are the producers of information in the system. They produce pieces of information that are called *events* and which are *published* to the notification service.

- The *consumers*, also called *subscribers*, are consumers of information. They express their interest in an event or a pattern of events by issuing one or more *subscriptions* that they *register* to the notification service. Each subscription represents a set of events in which the consumer is interested. When a consumer is not interested anymore in that set of events, it *cancels* the corresponding subscription by the notification service.

- The *notification service* is the main dedicated middleware infrastructure. It realizes the *interface* between consumers and producers. It processes events published by the producers and subscription registrations and cancellations issued by the consumers. Most importantly, it achieves the dissemination of information by *routing* events published from the publishers towards the consumers interested in those events, by issuing them *notifications*.

Those concepts are illustrated in figure 2.1.

2.2.2 Dissemination of information in Publish/Subscribe systems

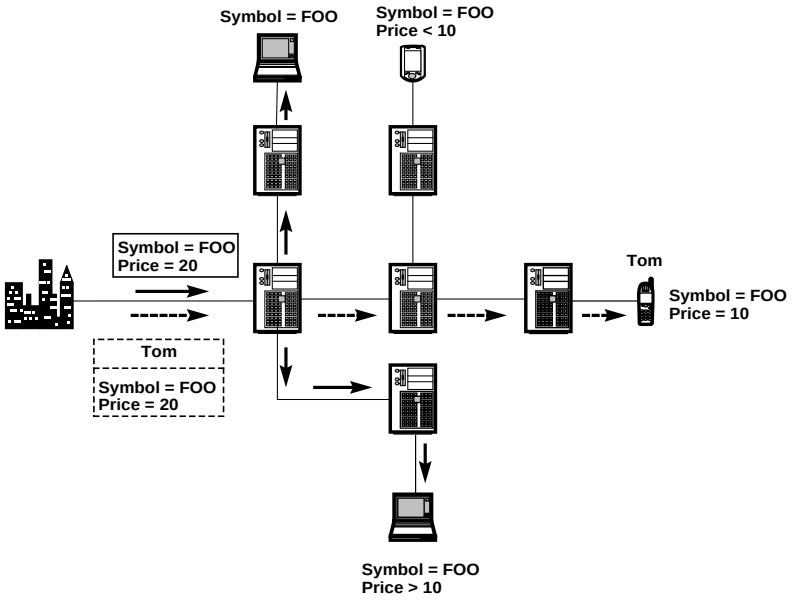


Figure 2.2: Example of content routing vs. IP routing. Consumers' interests are shown next to them. In IP routing, the message addressed to "Tom" is directly routed to him, independently of his interests and the message's content (shown in dashed lines). In content routing, a message is routed according to its content and the interests specified by the consumers (shown in plain lines).

The main specificity of the pub/sub paradigm resides in the way the information is disseminated from the source to the destination(s). Unlike most other existing systems(e.g., those based on IP routing), receivers in a pub/sub system are not directly targeted by a sender but are rather indirectly addressed according to the content of events and the interests they registered. A producer publishes an event to the notification service without addressing it to any particular receiver. The notification service then examines the *content* [1] of the event to send a notification to all (in the ideal case) the consumers that are interested in receiving it. Those are the consumers that specified an interest for that event by registering an appropriate subscription. This concept is called *content routing* and is illustrated in Figure 2.2.

This form of communication allows the different participants in the system to be totally decoupled from each others.

[1] the term content is used here by opposition to a destination address. It can refer to the simple specification of a topic in the event or to its whole content.

2.2.3 Decoupling between producers and consumers

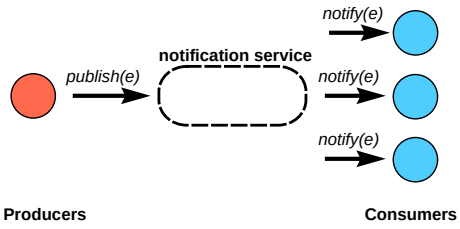


Figure 2.3: Space decoupling: producers and consumers do not know each others.

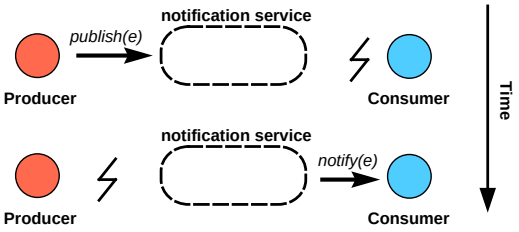


Figure 2.4: Time decoupling: producers and consumers do not participate in the interaction at the same time.

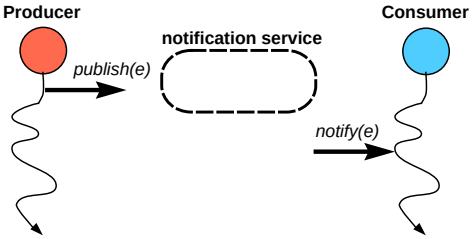


Figure 2.5: Synchronization decoupling: producers publish and do something else, consumers may be doing something else when being notified.

The notification service provides the interaction between producers and consumers. It enables producers and consumers to act independently of each others, or in other words, it *decouples* them. In [55], the authors classified the various forms of decoupling achieved by a pub/sub system along three dimensions as follows:

- *Space decoupling:* The participants in the system are unaware of each others. Producers publish events directly to the notification service, without addressing them to particular consumers. Producers generally do not even detain any kind of information on the population of consumers (their identity or numbers). Similarly, consumers are unaware of the population of producers in the system (although producers may have the ability to advertise the nature of their events). This form of decoupling is illustrated in figure 2.3.

- *Time decoupling:* The participants do not need to be connected to the notification service at the same time during an interaction: the producer may publish an event while the consumer is disconnected, and conversely, the consumer may receive a notification for an event while the producer that published it is disconnected. This form of decoupling is illustrated in figure 2.4.

- *Synchronization decoupling:* Communication between producers and consumers is performed in an asynchronous manner. Unlike in the client/server model (where the client is usually the consumer of information), consumers in a pub/sub system are not blocked while waiting for information. This is due to the fact that in the client/server model, a client is waiting for information at a specific time. In a pub/sub system, consumers may be notified for information at any time, while performing other concurrent activities. Similarly, producers are not blocked while producing and publishing events. This final form of decoupling is illustrated in figure 2.5.

The strong decoupling that a pub/sub system offers between producers and consumers of information makes it well adapted to distributed systems that are intrinsically asynchronous (such as mobile environments [70]), and offers higher chances to scale to large populations of participants.

2.3 Subscription models

2.3.1 Introduction

In a pub/sub system, consumers express their interests in particular events by issuing *subscriptions* and registering them by the notification service. A subscription S is a filter on the set of all possible events. It represents the set of events in which the consumer – that registered the subscription – is interested. We say that such an event *matches* subscription S.

As a consequence, the subscription language that is used to express the consumers' interests is extremely important. Different subscription languages allow the consumers different degrees of expressiveness to specify their interests. Of course, the subscription language depends greatly on the data format that is used to produce events. For example, if events are simple flat structures, there is no point in using a structured subscription language.

Also, when an event e is published, the notification service is to deliver a notification to all (in the ideal case) the consumers that have a matching subscription. Hence, the notification service has to determine, in some way, the set of subscriptions that event e matches. Consequently, the subscription language also greatly influences the behavior of the system in terms of performance and scalability.

There are many different subscription languages available to express a consumer's interests. They are commonly classified as belonging to one of the three following major *subscription models*:

- The topic-based subscription model
- The content-based subscription model
- The type-based subscription model

2.3.2 Topic-based subscription models

In a topic-based subscription model, events are grouped in topics (or subjects). That is, each event belongs to one or more topics. A subscription reduces to the specification of a topic. The consumer that registered the subscription wishes to receive all the events that belong to that topic. Most earliest pub/sub systems are based on the topic-based subscription model. Examples of such systems are TIB/RV [86], Vitria's Publish-Subscribe architecture [110], iBus [6], SCRIBE [33], Bayeux [127] and the CORBA Notification Service [64].

Topics are very similar to the notion of groups as defined in group communication [53]. This is not surprising since the first systems to offer pub/sub interaction were actually extensions of group communication toolkits [20, 47], and the subscription scheme was thus inherently based on groups [19]. Consequently, registering a topic T can be viewed as becoming member of a group T, and publishing an event e that belongs to topic T is similar to broadcasting the event to all the members of group T.

A topic corresponds to a logical *event channel* between a possible producer and all interested consumers. In other words, there exists a static association between a channel and all the corresponding consumers. Hence, the notification service does not have to compute the set of interested consumers when an event is published. Also, because of that equivalence between topics and groups in group communications, implementations of topic-based pub/sub systems can be done quite easily and efficiently, most notably by exploiting the large amount of research done in the multicast area and the existing network-level multicast implementations (though IP-multicast has seen a rather limited commercial deployment since its introduction [52]).

On the other hand, the topic-based subscription model suffers from one major drawback, namely the lack of expressiveness that it offers to the system's consumers. This is quite obvious, since a topic is a "rough" filter, in the sense that events that belong to that topic are in most cases different from each others. When registering that topic, a consumer has no choice but to receive all the events that belong to the topic. To address this problem, several solutions have been proposed, most notably the

organization of topics into *hierarchies*, according to containment relationships (such a hierarchical organization is used by Usenet News). A registration made to a topic in the hierarchy implies the registration to all subtopics. Such a hierarchical organization can be found in Usenet News [105, 2], for example, where articles are organized into newsgroups, which are themselves logically organized into hierarchies of subjects. Also, most systems allow the use of convenience operators, such as wildcards, first introduced in TIB/RV [86] and which enable the registration to several topics with one single subscription. However, those improvements do not solve fundamentally the issue of lack of expressiveness. Worst, they may lead to an inefficient use of bandwidth, and to an increasingly high number of topics, which may turn out to be unpractical and unscalable, since current network-level multicast implementations like IP-multicast do *not* handle well dynamic group management.

2.3.3 Content-based subscription models

The content-based model is the most general subscription model and has gained a lot of attention from the research community. The main reason for this is that it offers much higher expressiveness to the consumers in the system than the topic-based model. Most recent pub/sub systems employ the content-based model. Examples of such systems are: Gryphon [14], Siena [28], Elvin [107], Jedi [44], LeSubscribe [98], Ready [66], Hermes [96].

In the content-based model, events are not classified according to pre-defined external criterion (such as topics), but according to the properties of the events themselves. Consumers express their interest by specifying conditions over the *content* of the events. Such conditions may be constraints on the *values* of some of the attributes contained in the events, or constraints on the *structure* of the events (provided that they have one). Value constraints are generally expressed in the form of name-value pairs and with common comparison operators ($=$,$\neq$,$<$, $\leq$, $>$, $\geq$). Constraints on the structure come in different forms, often depending on the format used to produce events. Also, constraints can be logically combined with some operators (and, or). In most existing pub/sub systems, such as Siena [28], events come in the form of a collection of attribute-value pairs. A subscription is composed of conjunctions over the values of attributes in the events. Only recently have some systems used some subscription languages that include both constraints on values and on structures.

Although obviously more expressive than the topic-based model, the content-based subscription model adds significant complexity to the routing operation performed by the notification service. Indeed, unlike in the topic-based model, there is no static association between the topic of an event and the set of interested consumers, in the content-base model. Rather, each time an event is published, the notification service has to examine the content of the event to determine which consumers are interested in it. Nevertheless, the content-based model still represents the most viable option in large-scale distributed systems, where information frequently comes in complex forms and most often cannot be classified in topics. Also, the high expressive power that it offers makes up in most cases for the added complexity.

2.3.4 Type-based subscription models

Type-based pub/sub was first proposed in [99] and fully developed in [89]. The observation that topics usually regroup events that present commonalities not only in content, but also in structure, led to the idea of replacing the topic-based subscription model by a scheme that filters events according to their type. In other terms, the notion of event kind is directly matched with that of event type. This enables a closer integration of the language and the middleware. Moreover, type safety can be ensured at compile-time by parameterizing the resulting abstraction interface by the type of the corresponding events.

The type-based subscription model uses concepts from object-oriented programming: events are declared as objects belonging to a particular type, which can thus encapsulate member attribute as well as methods. The type-based model can then lead to a natural description of the content-based

model through public members of the considered event type, while ensuring the encapsulation of these events.

2.4 Architectural models

The notification service realizes the interaction between consumers and producers, enabling them to be asynchronously decoupled. In this section, we focus on the various architectural models around which a notification service can be built, how its various components are structured and how they communicate with each other.

2.4.1 Centralized infrastructures

The centralized architecture is the simplest one. As its name suggests, this approach implements the notification service as a single component, called a router (or broker, or server). This component is responsible for performing all the tasks of the notification service. This approach has the merit to be quite simple to realize, since there is no need to implement any form of communication protocol. Only the interface between the consumers and the producers has to be implemented. However, for scalability reasons, it is not suitable for large scale distributed systems and may only be viable in local networks or in environments with a very limited number of consumers and producers. The most notable pub/sub system implemented around the centralized infrastructure is the Elvin content based routing system [107]. The authors mention a distributed extension of Elvin, but do not discuss how they plan to achieve distributed content routing.

2.4.2 Multicast based approaches

Because of the equivalence between topics and groups in group communications, applying multicasting to a topic based pub/sub system is straightforward. For that reason, most early –topic-based– pub/sub systems relied on a network-level multicast protocol such as IP-multicast, as the communication layer for the notification service (brokers are not even needed). In addition to that, a network level multicast protocol has the advantage of providing efficient dissemination of information in terms of latency and throughput. On the other hand, the implementation of a large scale pub/sub system using existing multicast is strongly limited by the rather limited commercial deployment of network-level multicast protocols. Besides, if there is not much commonality in the interests of the consumers, using multicast requires to setup large numbers of groups, which can lead to scalability problems. Otherwise, using too few groups yields to inefficient use of network bandwidth and requires consumers to filter out potentially large amounts of unwanted data, which is strongly undesirable for consumers with limited resources (such as mobile devices).

Finally, although straightforward in the case of topic-based pub/sub systems, applying multicasting to content-based pub/sub systems is not trivial at all. This is due to the fact that multicast services do not consider the content or structure of the information in the multicast process. Thus, consumers' interests cannot be directly mapped to multicast groups. One way to address that issue consists in using the concept of multiple layers in the same multicast group (as commonly proposed for video streaming, for example), but this solution requires the data to be highly structured and typically the structuring is such that the recipient cannot receive any arbitrary layers it chooses to. In [108], the authors propose another approach, termed Content-Based Multicast (CBM), which basically performs content filtering at the interior nodes of an IP-multicast tree. This solution enables to take into account the structure and semantics of the information while reducing network bandwidth usage and delays experienced by consumers. The authors propose algorithms for determining the optimal placement of a given number of content routers in the IP-multicast tree, with respect to two criteria: minimizing total network bandwidth utilization and minimizing mean information delivery delay.

2.4.3 Overlay Network of Content Routers

In this approach, the notification service is composed of a collection of routers organized in an overlay network. An overlay network is a virtual network of nodes and logical links that is built on top of an existing network with the purpose to implement a network service that is not available in the existing network, that is, in this case, the pub/sub functionalities of the notification service. Most often, wide scale systems are implemented on top of the Internet and routers communicate using the underlying transport protocol TCP/IP.

The tasks performed by the notification service are distributed amongst the set of routers, each of which only knows and communicates with a limited set of neighbor routers. Also, a given router may have some consumers and producers connected to it as well. Consequently, this approach achieves high degrees of scalability, because even if the size of the system grows, the number of neighbors for each router remains bounded which ensures that each router manages a bounded number of concurrent connections and data structures, and that the workload is evenly distributed amongst the routers in the system.

For those reasons, most actual pub/sub systems structure their notification service as an overlay network of content routers. Examples of such systems are TIB/RV [86], Jedi [44], Gryphon [14], or Siena [28].

The topology around which the routers are organized is a graph in the most general case, but in most current systems(Jedi [44], Gryphon [14], or Siena [28]) the routers are organized in tree structures, where consumers access the system at the leaves and producers at the root. The main advantage of that topology is that it offers major simplifications by enabling a one way propagation of events (from roots to leaves) and consumers' subscriptions (from leaves to roots).

Systems architectured around an overlay network of routers are perfectly suitable for large-scale distributed environments. However, implementing and deploying such systems is significantly more difficult than with the two previous approaches. Indeed, complex routing and subscription management protocols must be implemented on each router to ensure proper dissemination of information from producers to consumers. Also, those protocols, often implemented at the application level, are inherently slower than one implemented at the network level, even with the high power of modern devices and networks. This may adversely impact performance in terms of delays and throughput.

Finally, one other major limitation of systems built as an overlay network of routers resides in the infrastructure itself. Indeed, building and maintaining the topology is often a complex operation which consumes a lot of resources. Also, the system usually relies on a fixed infrastructure of reliable brokers, or assume that a spanning tree of reliable brokers is known beforehand. This approach clearly limits the scalability of the system in the presence of large and dynamic consumer populations.

2.4.4 Peer-to-peer Overlay Networks

A peer-to-peer(P2P) network is a network that relies on computing power at the edges of a connection rather than in the network itself. A pure P2P network does not have the notion of clients or servers, but only equal peer nodes that simultaneously function as both clients and servers. Most of the times, a peer can only communicate with its neighbors, and with further peers through multiple hops.

This model has been used extensively for implementing file sharing applications. Examples of popular applications are Napster [84], Gnutella [61], or Freenet [41], I2P [71], GNUnet [60, 16, 17], which have anonymity features built in, or very recently Pastis [73], a decentralized multi-writer P2P file system.

Besides popular file-sharing applications, the P2P paradigm is appropriate for building any kind of large-scale distributed systems/applications. Indeed, P2P networks are completely decentralized and self-organizing, allowing the system to re-structure the network when a node joins or leaves. Most importantly, by using the resources of all participants, the system offers high scalability to large populations.

A popular class of p2p systems are "structured" P2P networks. These systems implement the services of a Distributed Hash Table and offer fast and scalable resource look-up/routing. Examples of such systems are Pastry [104], Chord [113], Tapestry [126],I3 [112], CAN [101] or TOPLUS [59].

Because of their self-organization and scalability properties, implementing pub/sub over P2P networks has become a hot research topic in the recent years. Several proposals have been made to implement the pub/sub interface over P2P overlays. We cite Bayeux [127] and Scribe [33] for topic-based systems and Hermes [96], Reach [91], Homed [40] and daMulticast [9] for content-based systems. Finally, in [114] and [116], the authors use the Chord [113] structured P2P network to implement a content-based pub/sub system.

2.5 Content routing in a Publish/Subscribe system

In section 2.2.2, we defined the concept of *content routing*, namely: the process of disseminating a given event, according to its content, to the consumers that are interested in it. In other words, the process of routing an event according to its content and the subscriptions registered by the consumers. In this section, we explain the basic mechanisms that must be implemented in a notification service to achieve content routing.

2.5.1 Event filtering

A consumer C_i that is interested in receiving a particular event e must first register a subscription S_i such that e matches S_i. Hence, in order to deliver an event e to interested consumers, the notification service must identify the subscriptions registered by the consumers, that event e matches. This process is called *event filtering*. It is basically an extension to the case of multiple subscriptions of the notion of event matching that we defined in section 2.3.1. The notion of *event filtering* is formally defined as follows: given a set of subscriptions $\{S_1 \cdots S_n\}$ and an event e, identify the set of subscriptions that event e matches.

Event filtering is a central and challenging problem. Indeed, a system that uses an inefficient filtering algorithm has little chances to scale to large-scale systems that comprise large consumer populations (and hence subscriptions). Also, in a typical pub/sub system, a high rate of event publication is expected. Then, the system must perform event filtering at a high rate. Obviously, the trivial solution that consists in applying sequential operations of matching tests is not viable.

Besides, the event filtering operation greatly depends on the subscription language that is used to express the consumers' interests. In the case of the topic-based pub/sub systems, event filtering boils down to identifying the subscriptions that belong to the topic specified in the event. As for content-based subscription models, the majority of existing pub/sub systems(Gryphon [14], Siena [28] and Elvin [107]), use "flat patterns" subscriptions, in the form of a set of attributes and simple arithmetic or boolean comparisons on the values of these attributes. There has been various works on the filtering of data using such "flat patterns", including research on rule/trigger processing systems [69, 68] and pub/sub systems [4, 57, 85].

However, since XML(eXtensible Markup Language) [119] emerged as a standard for information exchange on the Internet, there has been also an increased interest in using more expressive subscription languages that exploit both the structure and the content of published XML documents, such as the XPath language [118]. In this context, a large number of XML filtering approaches have been recently developed [7, 83, 35, 48, 62, 67, 75].

2.5.2 Content routing

In centralized systems, event filtering is sufficient to achieve content routing. Indeed, all consumers and producers are connected to one single entity, which knows all the subscriptions registered by the consumers. Hence, once event filtering has been performed, routing the event reduces to forwarding it to the consumers that registered the identified subscriptions.

However, in distributed systems, event filtering is not enough and routing strategies must be developed to achieve content routing. In a distributed system, the notification service consists of a set of nodes called routers, or brokers. A given router has a certain number of neighbor routers and may have some consumers and producers connected to it as well. To achieve content routing, each router knows a certain number (maybe zero) of the subscriptions registered by the consumers. This data is stored in some form in a *routing table*. When the router receives an event, it uses its routing table to determine to whom it must forward the event, according to its content.

A first routing strategy is the "data flooding approach". It consists in having a router only know about the subscriptions of the consumers connected to it, if any. When an event e is published, it is broadcast to all brokers. Then, the routers that have local consumers filter the event and deliver it to the consumers that are interested in it. This strategy has the advantages to be simple and optimal in terms of space requirements. On the other hand, it is not efficient in terms of usage of network bandwidth and routers' resources, as each router in the system processes every event, even if no or few consumers are interested in it. This strategy is illustrated in figure 2.6.

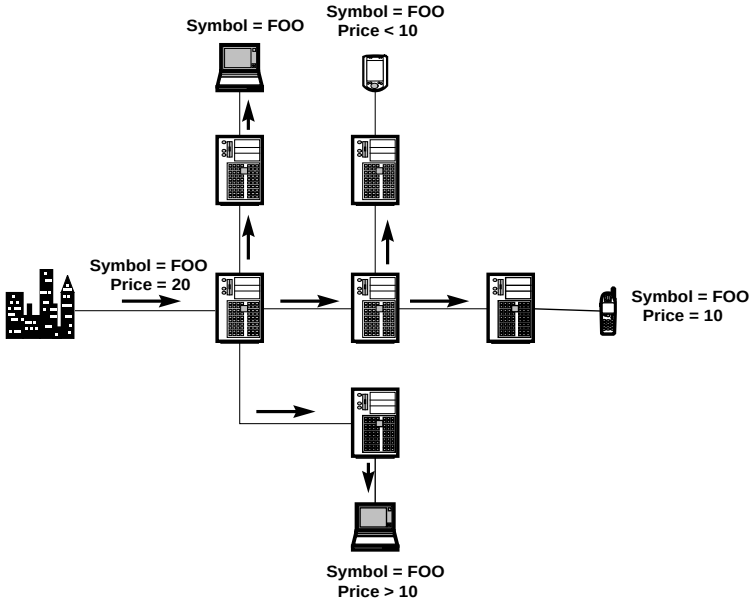


Figure 2.6: In the data flooding approach, all routers process an event.

Another strategy is the "subscription flooding approach". Each router that manages some local consumers advertises their subscriptions to all other routers. Hence, each router knows all the subscriptions registered by all the consumers. When an event is received at a given router, it is filtered and forwarded to the neighbor routers that have at least one matching subscription. The "subscription flooding approach" is optimal in terms of network bandwidth usage, since an event is only forwarded to the neighbor routers that have matching subscriptions. On the other hand, since each router has to store, and filter events against a large number of subscriptions, this approach is clearly inefficient in terms of space and processor usage. The "subscription flooding approach is illustrated in figure 2.7.

Both the data flooding and subscription flooding approaches perform poorly, either in terms of bandwidth, space or processor usage, and have little chances to scale to large consumer populations. Hence, other approaches have been developed to realize a compromise between data and subscription flooding. They are based on more elaborate routing strategies and subscription advertisement schemes. Most of these approaches consist in having a given router only know about a subset of the subscriptions registered by the consumers, and only advertise that restricted set of subscriptions to its neighbor routers. The challenge then consists in optimizing *routing accuracy*. *Routing accuracy*

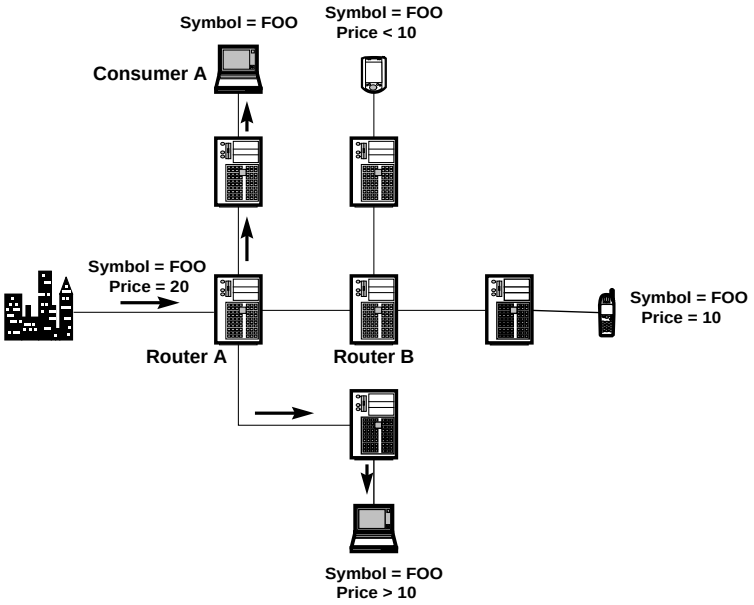


Figure 2.7: The subscription flooding approach. Router A knows all the subscriptions registered by the consumers. It prevented necessary propagation of the event to router B.

can be defined in terms of the number of irrelevant events received by a router or a consumer, those events are called *false positives*, and the number of events that failed to be delivered to a router or a consumer that was interested in it. Those are termed *false negatives*. Optimizing routing accuracy consists in minimizing the number of false positives and/or false negatives.

2.5.3 Subscription management

Systems based on more complex routing strategies than the "data flooding" or the "subscription flooding" approaches implement elaborate subscription management techniques. The first aspect of subscription management is the subscription advertisement scheme. Subscription advertisement can be defined as the way the routers in the system learn about the subscriptions registered by the consumers. More precisely, considering a given router R, the subscription advertisement scheme defines which subscriptions should be advertised and to which neighbor routers they should be advertised. For example, the subscription flooding approach is based on a subscription advertisement scheme where all subscriptions stored at a router are advertised to all neighbor routers.

The other aspect of subscription management is the way subscriptions are managed locally at a given router. Especially, reducing the number of subscriptions stored at a given router and the number of subscriptions advertised to other routers is desirable since it reduces space requirements, usage of network bandwidth, and speeds up event filtering. To achieve this, some key techniques have been developed, such as subscription containment and subscription aggregation.

Subscription containment. Subscription containment [2] is defined as follows: we say that subscription S_1 contains another subscription S_2 (written $S_1 \supseteq S_2$) if and only if any event e that matches S_2 also matches S_1. Conversely, we say that S_2 is contained by S_1 and we write $S_2 \subseteq S_1$.

This concept is illustrated in Figures 2.8 and 2.9. Note that the containment relationship is transitive and defines a partial order. Also, note the analogy between subscription containment and the covering of IP subnets.

Subscription containment has been well studied in the context of Siena [28] for attribute/value-based subscriptions. It is a key technique to reduce the number of subscriptions stored at the

[2]The term covering is also commonly used in the literature

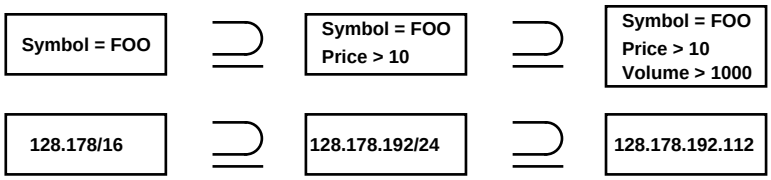


Figure 2.8: Analogy between subscription containment and IP subnets covering

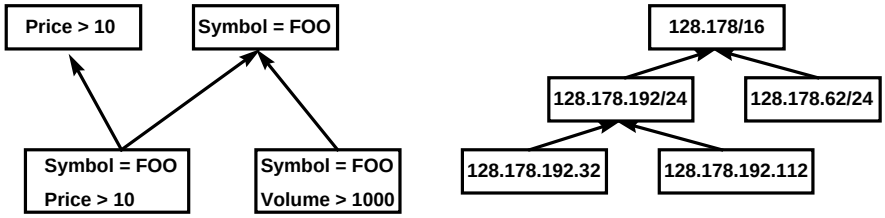


Figure 2.9: There are no relationships between $Price > 10$ and $Symbol = LU$: the containment relationship defines a partial order.

routers and in certain cases the number of subscriptions advertised to other routers (depending on the advertisement scheme). This technique is based on the following observation, illustrated in figure 2.10: if S_1 contains S_2, then at router A, it is sufficient to filter events against S_1. Indeed, if an event e matches S_1, then it is useless to test it against S_2, since e has to be forwarded to router B anyways. Now if e does not match S_1, then because S_1 contains S_2, e does not match S_2 either $((A \Rightarrow B) \Leftrightarrow (\overline{B} \Rightarrow \overline{A}))$. Hence, it is again useless to test e against S_2. As a consequence, router A needs only to know subscription S_1.

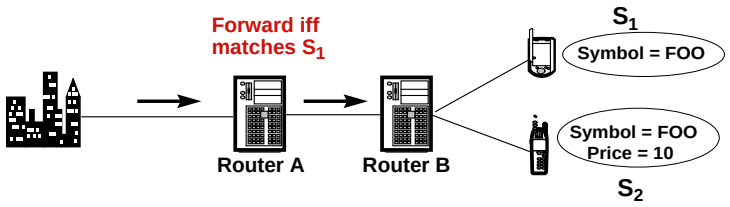


Figure 2.10: Thanks to subscription containment, router A needs only to know about S_1

In addition to that, subscription containment enables to reduce the number of subscriptions advertised in the system. Indeed, when a router advertises the set of its subscriptions to other routers, it only needs to send containing subscriptions, that is, the subscriptions that are not contained by another subscription.

This technique enables to reduce significantly the number of subscriptions advertised in the system and stored in each router, as in practice, many consumers have similar interests. However, it may introduce *routing inaccuracy* in the presence of *subscription cancellation*. This issue will be developed in greater details in the next chapter.

Subscription aggregation. Subscription aggregation is another key technique that enables to reduce even more the number of subscriptions stored and advertised in the system. Also, this technique enables to cope with the case where there are no containment relationships between subscriptions. Subscription aggregation can be defined as follows: consider two subscriptions S_1 and S_2 such that there are no containment relationships between each others. It is possible to build another subscrip-

tion S_a, called the *aggregated subscription*, that contains both S_1 and S_2. [3] The definition can be trivially extended to the case of multiple subscriptions. We say that aggregation is perfect if any event that matches the aggregated subscription matches at least one of the original subscriptions. Aggregation is imperfect if there exists an event that matches the aggregated subscription but none of the original subscriptions. For example, if S_1 = "$price < 11$" and S_2 = "$price > 11$", a possible aggregated subscription is S_a = "$price \neq 11$", and aggregation is perfect. Now if S_1 = "$price < 10$" and S_2 = "$price = 11$", then a possible aggregated subscription is S_a = "$price \leq 11$" and aggregation is imperfect, since an event with a price between 10 and 11 (boundaries excluded) does not match either S_1 or S_2.

Similarly to subscription containment, subscription aggregation enables to reduce the number of subscriptions stored and advertised in the system, by advertising the aggregated subscription instead of the original ones. However, imperfect aggregation introduces routing inaccuracy in the system in terms of false positives (an out-of-interest event that is received). Then, a key challenge is to build the aggregated subscription so as to minimize the number of false positives forwarded to each router. Intuitively, the "tightest" the aggregated subscription S_a contains the original subscriptions, the less the loss in routing accuracy. In [34], the authors address that issue in the case of XML events and subscriptions built using the XPath language. Given a set S of subscriptions, they propose an algorithm that enables to compute an aggregated subscription S_a so as to minimize the induced routing inaccuracy. They also extend that notion to the case of a set of aggregated subscriptions that satisfies a given space constraint.

2.6 Formalization of a Publish/Subscribe system

In this section, we give a simple modelization of a pub/sub system. We provide some formal definitions and notations that will be used throughout the rest of the thesis. Also, we give several definitions and criteria to analyze and evaluate the behavior of a pub/sub system in terms of correctness and reliability.

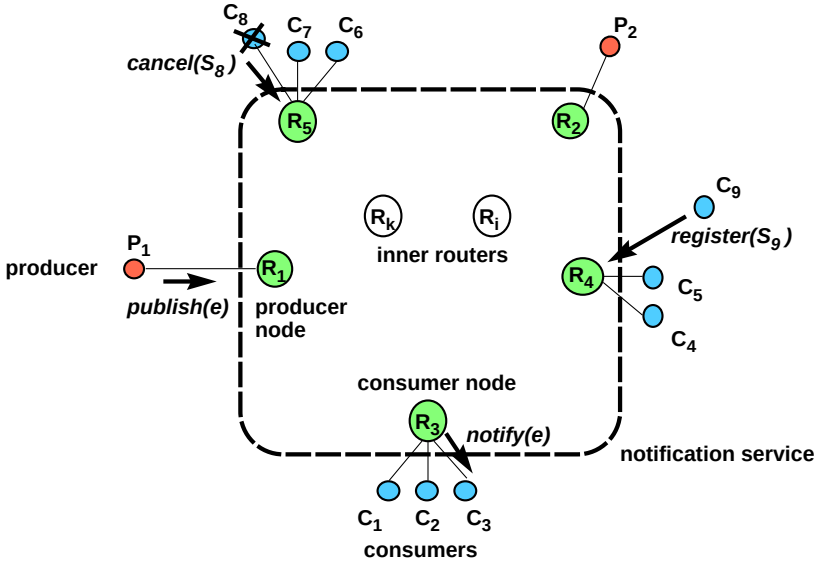


Figure 2.11: Formalization of a pub/sub system

[3] In the case where a subscription contains the other, then the aggregated subscription is the one that contains the other

2.6.1 Definitions and notations

A Pub/Sub system can be formalized as in Figure 2.11. The notification service is composed of a collection of interconnected *routers*. The routers where some producers are connected are called *producer nodes*, while the routers where some consumers are connected are referred to as *consumer nodes*. The other routers are termed *inner routers* or *routing nodes*. A producer *publishes events* at its producer nodes. A consumer node issues *notifications* to consumers to notify them of events of interest. Consumers issue *registrations* to register *subscriptions* or *cancellations* to cancel previously registered subscriptions. Routers are connected with their *neighbor routers* via *interfaces* or *links*. A router R_1 that sends an event to a neighbor router R_2 is said to be the *upstream router* for router R_2. Conversely, R_2 is said to be a *downstream router* for router R_1. We similarly define the notions of *upstream* and *downstream* interfaces. Note that these notions are local to a particular spanning tree of an event.

In the case of a peer-to-peer overlay network, all the *peers* in the system are routers. Most often, they are also producers and consumers at the same time. In the case where events are XML documents, we will interchangeably use the term *documents*. Also, when subscriptions are expressed with a tree structured language such as XPath, we will interchangeably use the term *tree patterns*. Finally, we defined a *notification* as the result of a consumer node notifying a consumer of an event of interest. Although formally different than the event itself, the two terms are often used interchangeably in the literature.

The notations that we defined are summarized in Table 2.1.

Name	**Generic notation**	**Alternate names**
Router	R_i	Node, Broker
Consumer	C_i	Subscriber
Producer	P_i	Publisher
Peer	P_i	
Event	e	Publication, Notification, Document
Subscription	S_i	Interest, Tree Patterns
Consumer node	C_i	
Interface	I	Link
Upstream interface	I_{up}	
Downstream interface	I_{down}	
Neighbor		
Contains	$\supseteq$	Covers
Is Contained	$\subseteq$	Is Covered

Table 2.1: Formal notations in a Publish/Subscribe system.

2.6.2 Correctness criteria and reliability in a Publish/Subscribe system

In this section, we provide some definitions and criteria to model the behavior of a pub/sub system.

Perfect Routing. We say that routing is *perfect* if and only if *all* the consumers interested in an event, and only those, receive it. In our model of a Pub/Sub system, it follows that consumer nodes filter out irrelevant events, and hence consumers never receive out-of-interest events. Thus we need a definition of perfect routing adapted to our model, as follows: routing is *perfect* if and only if all the *consumer nodes* that have consumers interested in an event, and only those, receive it. We then define the notion of "bandwidth-efficient" perfect routing as follows: an event is propagated to a link if and only if it leads to at least one consumer interested in the event. An equivalent definition is: an event is received by a router if and only if it is interested in it, in the sense that there is a matching subscription in its routing table. This latter definition of "bandwidth-efficient" perfect routing suits to all Pub/Sub models. In the rest of the thesis, and unless otherwise specified, we will only consider the definition of "bandwidth-efficient" perfect routing. For example, routing in Figure 2.6 is perfect but not bandwidth-efficient. In Figure 2.7, it is "bandwidth-efficient" perfect.

Routing Accuracy, false positives, false negatives. If routing is not perfect, then there may be some routers not interested in the event that received it, termed the *false positives*, or some routers interested in the event that did not receive it, termed the *false negatives*. Routing accuracy is defined in terms of false positives and false negatives. The less the number of false positives and false negatives, the more accurate routing is. *Perfect routing* and routing accuracy directly depends on the *correctness* of subscription advertisement.

Correctness of subscription advertisement. As previously mentioned, subscription advertisement is the way subscriptions are advertised by a router to some of its neighbor routers. We say that subscription advertisement is *correct* if it yields to *perfect routing*. More formally, consider a router R. We say that subscription advertisement is correct at router R if and only if its routing table is *sufficient* for the neighbors for which it routes events. That is, if one such neighbor holds a matching subscription for a given event, then R also holds a matching subscription (maybe not the same). Conversely, if no neighbors hold a subscription that matches a given event, then neither does R. It is then trivial to see that if subscription advertisement is correct, then routing is perfect. For example, the *subscription flooding* subscription advertisement scheme is *correct*. However, it is obviously not optimal in terms of storage and hence routing speed, and in terms of bandwidth usage.

Consistency. We say that the system's state is *consistent* (with the consumer population) if subscription advertisement is correct at all the routers and corresponds to the *actual* consumer population at the considered time , except the advertisements that are being propagated and processed by the system at that time. In other terms, the system's state is consistent if the producers-to-consumers routing paths are correct and reflect all the subscriptions registered by the consumers.

Fault-tolerance and recovery delay. Fault-tolerance is the ability of the system to recover from system failures. System recovery and component recovery must not be confused. A component, such as a link or a router, recovers when it becomes operational once again after a crash. In contrast, the system has recovered when it operates "normally" once again. This usually requires a delay after the last down component recovered, during which the system is updated and takes necessary actions to cope with the failures that have just occurred. This delay is referred to as the *recovery delay*. In contrast, the *downtime* or *crash duration* is the delay between the time the first component crashed and the last one recovered.

Reliable delivery. Delivery is *reliable* if consumers receive all the events they are interested in in spite of routers failures, including during the downtime duration, but false positives may be delivered.

Reliable subscription advertisement. Subscription advertisement is *reliable* if the system's state is *consistent* in spite of failures. In other words, subscription advertisement is *reliable* if it preserves correct producers-to-consumers routing paths that reflect all the subscriptions registered by the consumers despite failures. The downtime duration is not taken into account in our definition of reliable subscription advertisement. Indeed, it is often hard and burdensome to maintain a consistent state during the period of the outage. Rather, after the downtime period, the system necessitates a recovery delay to update its state. After that recovery delay, the system's state must be consistent with the consumer population. Also, the registration of new consumer subscriptions *must* be handled during the outage. When the system has recovered, its state must be consistent with the *actual* consumer population, and not the one before failures occurred.

2.7 Challenges of Publish/Subscribe systems

In this section, we identify the issues that are raised in large-scale distributed pub/sub systems, and the challenges that need to be addressed.

Scalability. The main challenge of actual pub/sub systems is their scalability, that is, their ability to achieve gracefully and efficiently the dissemination of information in large scale environments. In such conditions, one can expect a large and dynamic population of consumers, a high number of registered subscriptions and a high rate of events publications as well. Elaborate routing strategies and subscription management algorithms must be used to ensure that the system scales to large and dynamic consumer populations. Moreover, efficient filtering algorithms must be implemented to maintain good performance in the presence of high rates of events publication and to ensure the scalability of the system in terms of registered subscriptions. In addition, the minimization of routing tables sizes is crucial for the ability of the system to scale to large number of registered subscriptions. Indeed, not only does it save storage space, and hence enable to handle more subscriptions, but most importantly, it enhances content routing by speeding up event filtering. Finally, the architecture of the overlay must be designed carefully, since it has a great impact on the overall efficiency of the system, in terms of consumers' experienced latency, throughput (in terms of processing events' publication), and its scalability.

Reliability. Another major challenge of pub/sub systems is their reliability and their ability to cope with failures in the system. In particular, it is highly desirable to ensure the correctness of subscription advertisement, so that the system state is *consistent* with the consumer population at all times. In other words, the routers' routing tables must be accurate so that they reflect the actual consumer population. Otherwise, as previously mentioned, routing will not be accurate and the system will probably deliver out-of-interest events (false positives) or worse, fail to deliver events of interests (false negatives). Besides, in most large scale pub/sub systems (most of them are implemented on top of the internet), the overlay is not reliable. A router may fail at any time, and may or may not recover. It is highly desirable that the system handles those failures so that when it has recovered, the system's state is *consistent* once again with the actual consumer population. Also, it should recover gracefully, with minimum impact and maximum transparency for the consumers and producers for the duration of the outage. Finally, a desirable but often secondary goal is to maintain reliable delivery of events for the duration of the outage.

Security issues. A large scale distributed pub/sub system raises several security issues, such as access control (who is authorized to publish events or register subscriptions) and data encryption (only selected consumers can decode data). For that purpose, one can use the large amount of research work that has been done in the context of secure systems. However, there is a major issue in pub/sub systems, which is that the infrastructure that composes the notification service can generally not be trusted (especially for systems that are implemented on top of the Internet). In that case, events must be encrypted so that they can not be decoded by the infrastructure. Similarly, subscriptions registered by consumers must be encrypted so that the other parties (the infrastructure and the other consumers) can not decode them. As a consequence, several new challenges have been raised. One is to implement an event filtering algorithm that is capable of operating with both encrypted events and encrypted subscriptions. Another is to implement subscription management algorithms that can operate with encrypted subscriptions. This is especially challenging when subscription aggregation techniques are used. Security issues are *not* explicitly addressed in this thesis.

Expressiveness. The subscription language enables the consumers to express their interests in particular events. Therefore, to allow consumers to register diverse and flexible interests, it is highly desirable that the subscription language be as expressive as possible. On the other hand, more complex subscription languages means more complex filtering and subscription management algorithms. Therefore, the subscription language greatly influences the behavior of the system in terms of performance and scalability. The challenge is then to allow the consumers as much expressiveness as possible while maintaining good performance and scalability.

Most early and actual pub/sub systems such as Gryphon [14], Siena [28] and Elvin [107], use "flat-patterns" subscriptions in the form of attribute/value pairs. While better than topic-based models, those subscription language lack expressiveness especially by not taking into account the structure of information. Besides, the recent emergence of XML as a standard for information exchange on the Internet has led to an increased interest in using more expressive subscription languages. While XML filtering has aroused significant interest in the database community, XML-based pub/sub systems are yet to be deployed on an Internet-scale. In this thesis, we specifically deal with the XML language and the XPath [118] subscription language.

2.8 Survey of existing Publish/Subscribe systems

In this section, we survey the most popular existing pub/sub systems. We only focus on their most basic description and relevant features. Other aspects, such as routing, subscription management or reliability issues will be addressed in the related work section of the next chapters.

2.8.1 Topic-based systems

TIB/RV. TIB/RV [86] is one of the earliest commercial pub/sub systems and is widely used as a messaging middleware in enterprises. TIB/RV is a topic-based pub/sub system based on a distributed architecture. An installation of TIB/RV (a RV program) resides on each host on the network. It allows programs to send messages in a reliable, certified and transactional manner, depending on the requirements. Messaging can be delivered in point-to-point or pub/sub, synchronously or asynchronously, locally delivered or sent via WAN or the Internet. There are two main components in TIB/RV that enable to achieve dissemination of information: RV Daemons and RV Routing Daemons. A RV Daemon is a background process that sits in between the RV program and the network. It is responsible for the delivery and the acquisition of messages in a local level (a LAN) , either in point-to-point or according to the pub/sub paradigm. A RV Routing Daemon is an entity that represents a LAN. RV Routing Daemons are responsible for disseminating information at wide-area level.

Scribe. Scribe [33] is a research system designed at Microsoft Research that implements a topic-based system following a different approach than TIB/RV. Rather than implementing its own infrastructure, Scribe is built on top of Pastry [104]. Pastry is a generic peer-to-peer object location and routing substrate overlayed on the Internet. It allows to perform efficient large-scale routing of messages. Each node in Pastry is assigned a unique identifier in the network, called a nodeID. Messages can be routed to a specific node simply by specifying its identifier. Scribe is actually a pub/sub interface over Pastry. Dissemination of information is achieved by levering Pastry's direct routing capabilities: events are first routed to a rendezvous node before being multicasted from that node. It supports large numbers of topics, with a potentially large number of subscribers per topic.

Bayeux. Bayeux [127] is another topic based system built on top of an overlay network infrastructure, namely: Tapestry [126]. Tapestry is a wide-area location and routing architecture used in the OceanStore [74] globally distributed storage system. Bayeux leverages Tapestry direct routing capabilities, by routing events to rendezvous nodes and multicasting them from those nodes. Bayeux supports an arbitrary large number of topics while tolerating routers or links failures. Also, Bayeux includes some mechanisms for efficient load-balancing and bandwidth consumption.

2.8.2 Content-based systems

Elvin. Elvin [107] is a content-based pub/sub system developed at the Distributed Systems Technology Center (DSTC). Elvin is architectured around a single server that filters and forwards producer messages directly to consumers. Hence, it is more suitable to local networks than large-scale

distributed systems. The authors mention a distributed extension of Elvin, but the routing algorithms are not described. The subscription language in Elvin is quite expressive. Events are sets of named and typed elements. A subscription is a declarative boolean expression over the components of events. By issuing a subscription, a consumer can declare its interest in a number of events characterized by some common property. There exists an interesting concept in Elvin called *quenching* [106]. It allows producers to detect if there are no consumers that are interested in the events that they publish, in which case they stop publishing them. However, the authors do not describe how this concept is achieved in their system.

Jedi. Jedi [44, 22, 21, 45, 43] is content-based pub/sub system that uses a network of event servers organized in an overlay. The servers are organized in an arbitrary tree. Subscriptions are propagated upward the tree, and messages are propagated both upward and downward to the children that have matching subscriptions. The expressiveness of their data/filter model is rather limited because events come in the form of ordered sets of strings, and subscriptions are filters based only on equality and prefix tests on a message's strings.

Gryphon. IBM Gryphon [18, 87, 88, 72, 15, 5, 14] is a content-based pub/sub system that was developed at the IBM Watson research center. Gryphon uses a set of networked brokers to distribute events from producers to consumers. It uses a distributed filtering algorithm based on parallel search trees to efficiently determine where to route the messages. Gryphon was designed for widely distributed and high-volume environments, that is, distributed across countries or continents, and with large numbers of events and consumers. Gryphon is a real implemented system, and represents the reference framework for all the research in content-based dissemination of information carried out at IBM Watson.

Siena. Siena [23, 24, 25, 26, 29, 31, 103, 32, 27, 109] is another important contribution to the research in content-based pub/sub system. Siena also uses a network of event servers for content-based event distribution. The system was designed to provide efficient and scalable event routing over a wide-area network. Servers are organized in tree structures which leaves are the consumers' access points and roots are producers'. Siena implements a subscription advertisement scheme so that each server only holds a subset of the subscriptions that correspond to the subscriptions advertised by its neighbor servers or consumers. Also, Siena makes use of the containment relationships to reduce routing table sizes and subscription advertisements in the system. However, it is not clear how routing accuracy is preserved in the presence of subscription cancellation. Also, the expressiveness of its subscription language is rather limited, since events are attribute-value pairs and subscriptions are conjunctions over the values of attributes.

Rebecca. Rebeca [81, 58, 102] is a prototype notification service that incorporates several routing strategies. Also, advertisements are supported and other routing algorithms can be included in the system. The topology is that of a tree of brokers with a single root called the "root router". The Rebeca project was designed to provide a new and powerful event-based architecture for electronic business applications. Two example applications are currently implemented, a stock trading platform and an infrastructure for self-actualizing web-pages.

Onyx. Onyx [123] is an ongoing research project that aims at implementing a large-scale dissemination system that delivers XML messages based on user specifications for filtering and transformation. The system offers good expressiveness of their data/filter model since messages are XML documents and subscriptions are specified using a subset of the XQuery [121] language. Onyx leverages the YFilter [49, 50] technology for content-driven routing. Also, the authors address the issue of how to perform incremental message transformation in the course of routing. Several other challenges, such as routing table construction and query population partitioning are addressed.

2.8.3 Type-based systems

Hermes. Hermes [95, 93, 94] is a distributed event based middleware platform proposed recently by Bacon and Pietzuch. Hermes follows a type- and attribute-based pub/sub model that places particular emphasis on programming language integration by supporting type-checking of event data and event type inheritance. It is implemented on top of a peer-to-peer overlay network to implement content-based event delivery and handle dynamic, large-scale environments. Hermes also encompasses other functions such as security and fault-tolerance.

2.9 Concluding remarks

Pub/Sub has become a hot research topic in the recent years. It has inspired several research areas such as databases, distributed systems, data engineering or security. In this chapter, we tried to give a broad overview of pub/sub systems, spanning different aspects such as the subscription languages or the achitectural models. We also explained in a general manner how dissemination of information can be achieved in a pub/sub system. Finally, we surveyed the most popular and relevant systems that implement pub/sub. Note that we did not aim at giving a complete, formal specification of pub/sub systems, but rather to offer a general and clear understanding of their major features and properties. Formal specifications and models of pub/sub systems have been presented in [81], [117], or [13]. In the next chapter, we proceed to present the *XNet* content-based routing system that we designed for efficient dissemination of XML content in a large-scale environment.

Part I

The XNet XML Content-based Routing System

Chapter 3

XNet: Overview

3.1 Motivations

In this chapter, we present the architecture of the XNet XML content network, that we designed to implement *efficient* and *reliable* distribution of structured XML content to very large populations of consumers.

For that purpose, our system integrates several novel technologies. The routing protocol, XRoute [37], makes extensive use of subscription aggregation to limit the sizes of routing tables while ensuring perfect routing (i.e., minimizing inter-router traffic). The filtering engine, XTrie [35], uses a sophisticated algorithm to match incoming XML documents against large populations of tree-structured subscriptions, while the XSearch subscription management algorithm [100] enables the system to efficiently manage large and highly dynamic consumer populations. Finally, our XNet system is reliable in the sense that its state is consistent with the consumer population, and integrates several approaches to fault-tolerance to recover from various types of router and link failures [39].

We have analyzed the efficiency of our techniques with various simulations. In addition to that, to assess the performance of our system in realistic settings and to show that it is perfectly suitable for large-scale distributed environments, we have performed a large scale experimental deployment of our system in the PlanetLab [97] testbed.

The rest of this chapter is organized around the different technologies integrated in our system. We first discuss the model around which our system is organized in section 3.2. We then present the filtering, the routing and the subscription management algorithms, in the rest of this chapter and the next two chapters, respectively. We then focus on the reliability aspects in our system in chapter 6 before presenting results from experimental evaluation and a short survey of the related work.

3.2 System model

3.2.1 Architecture

XNet is implemented as an overlay network of routing brokers. Events are propagated through the nodes of the network, according to the messages' content and the subscriptions registered by the consumers. Each node of the overlay network acts as a content-based router. Each data consumer and producer is connected to some node at the edge of the network; we call such nodes *consumer* and *producer* nodes. To simplify the presentation, we assume that consumer and producer nodes are distinct, i.e., one cannot directly connect both a producer and a consumer to the same router node. The other nodes, that have no consumer or producer, are called *routing* nodes or *inner* nodes.

We assume that all routers know their neighbors, as well as the best paths that lead to each producer. This latter point can be easily achieved by having each producer create a spanning tree by broadcasting some kind of "announcement" message. We also assume that the number and location

of the producer nodes is known. In contrast, the consumer population can be highly dynamic and does not need to be known a priori.

Each routing node has a set of *links*, or *interfaces*, that connect the node to its direct neighbors. We assume that there exists exactly one interface per neighbor (we ignore redundant links connecting two neighbors). Nodes communicate using reliable point-to-point communication and are equipped with failure detectors that eventually detect the failure of their communication links and neighbors but may make mistakes. As will become clear later, if a node incorrectly suspects its upstream neighbor to have failed, it might take unnecessary recovery actions that, although time consuming, do not adversely affect the consistency of the global state of the system. We assume a crash-recover model with transient link and router failures (although the duration of failures is unbounded). For a given producer, we will generally denote by I_{up}, or *upstream interfaces*, the interfaces along the path up to the producer, and I_{down}, or *downstream interfaces*, the other interfaces (along the paths to the consumers). In general, we will discuss the properties and behavior of our protocol in the case of a single producer. The case of multiple producers will be addressed in a separate section.

The actual consumers are connected to consumer nodes via links that are not part of the overlay network, and therefore not associated with any of the node's interface. Furthermore, to simplify the presentation of the protocol, we assume that consumer nodes are edge routers with a single interface that connects them to the overlay network (this property can always be satisfied by introducing virtual consumer nodes at the edges of the overlay). Consumers register and cancel subscriptions via their consumer nodes. A consumer cannot cancel a subscription that it did not previously register (the consumer node will filter out such requests).

3.2.2 Data models

XNet was designed to deal with XML data, the *de facto* interchange language on the Internet. Producers can define custom data types and generate arbitrary semi-structured events, as long as they are well-formed XML documents. Consumer interests are expressed using a subscription language. Subscriptions allow to specify predicates on the set of valid events for a given consumer. XNet uses a significant subset of the standard XPath [118] language to specify complex subscriptions adapted to the semi-structured nature of events.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://.../soap/envelope/"
  SOAP-ENV:encodingStyle="http://.../soap/encoding/">
    <SOAP-ENV:Header>
        <t:Transaction
          xmlns:t="some-URI"
          SOAP-ENV:mustUnderstand="1">
            5
        </t:Transaction>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <Symbol>DEF</Symbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3.1: Sample SOAP message (corresponds to example 5 of [120]).

XML

The Extensible Markup Language (XML) [119] is emerging as the universal format for exchanging structured data over the Internet and increasing amounts of data are made available in XML format. Because of its simple structure, XML is easy to interpret and process by applications. By being

vendor- and platform-neutral, as well as agnostic about how content appears, XML makes it simple to integrate existing applications and to represent data in various human-readable formats.

Technically, XML is a meta-language: it permits the creation of markup languages customized to the needs of a particular application. XML defines an unambiguous mechanism for constraining structure in a stream of information. XML documents can optionally include a type definition (DTD or XML schema), which defines the document structure by describing its legal building blocks. An XML document is *valid* if it adheres to all the rules of its type definition, and *well-formed* if it is correctly structured. Validity is a semantic constraint and is not necessary for content-based filtering and routing of XML documents.

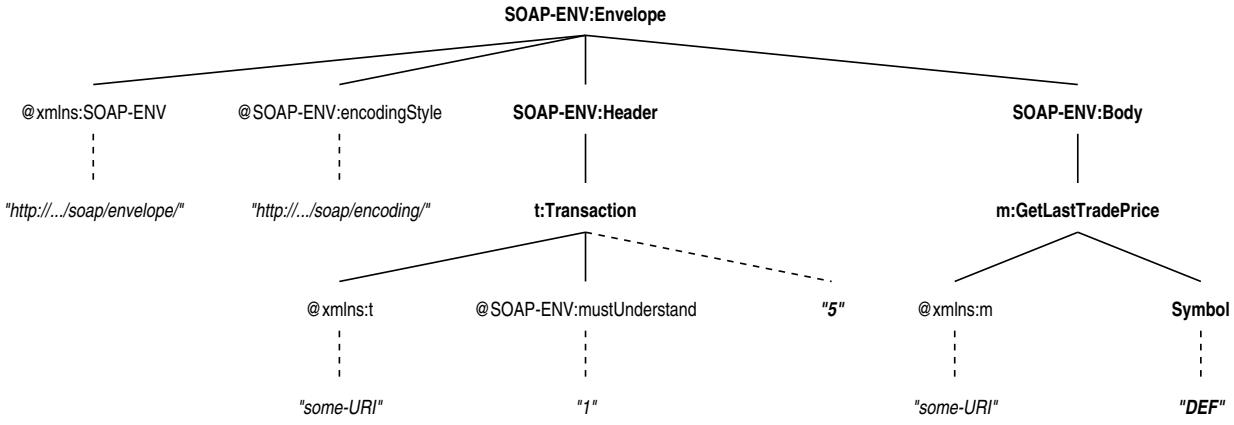


Figure 3.2: Tree representation of the XML document of Figure 3.1.

Figure 3.1 shows a sample XML document that represents a SOAP request asking the current value of the stock of a company. The document structure is specified by the means of start and end tags (tags are enclosed between `<` and `>`, and end tags start with `/`). Start tags contain an optional list of attributes, which are essentially key-value pairs. Text data can be enclosed between tag pairs, as between the `t:Transaction` and `Symbol` tags in Figure 3.1. We will refer to this text as the tag's value or content.

XML documents have a hierarchical structure. Figure 3.2 shows a tree-based representation of the XML document of Figure 3.1. Attributes are represented as children of their associated tag, and values as children of their associated attribute or tag. Attribute names are prefixed by the symbol `@` and values are represented within quotation marks.

Roughly speaking, we can distinguish between structural elements—tags and attributes—and the actual data values associated to these elements. Connections between structural elements are represented using plain lines, and between elements and values by dashed lines. Intuitively, the *type* of an XML document is defined by its tags and attributes, while the data associated to its tags and attributes define its *value*.[1] By disconnecting the dashed lines from the tree representation of an XML document, we obtain its type. Two SOAP request asking the value of different stocks would have the same type, but different values.

XPath

The structured, extensible nature of XML allows for a powerful combination of *type-based* and *value-based* content filtering. To that end, XML filters should be able to express constraints on both the type and the value of data. Several XML query and addressing languages can be used for specifying such filters. However, because of its simplicity and good standardization, the W3C XPath [118] addressing language—or a subset thereof—is most widely used for that purpose.

[1]Formally, the type an an XML document is specified by the associated DTD or schema.

XPath treats XML documents as a tree of nodes and offers an expressive way to specify and select parts of this tree. An XPath expression contains one or more location steps, separated by slashes (/). Each location step has an axis name, a node-test, and zero or more predicates (specified between brackets). The XPath axis designates a part of the document, defined from the perspective of the "context node"; the node test specifies the type and expanded-name of the nodes selected by the location step; predicates use arbitrary expressions to further refine the set of nodes selected by the location step. Predicates are generally specified as constraints on the presence of structural elements, or on the values of XML documents using basic comparison operators ($=$, $<$, $\leq$, $>$, $\geq$). XPath also allows the use of wildcard (*) and ancestor/descendant (//) operators, which respectively match exactly one and an arbitrarily long sequence of element names at a location step . The evaluation of an XPath expression yields an object whose type can be a node-set, a boolean, a number, or a string. When used to describe subscriptions, an XML document matches an XPath expression when the result is a non-empty node-set.

(i): `//m:GetLastTradePrice/Symbol`
(ii): `//m:GetLastTradePrice/Symbol[text()="DEF"]`
(iii): `/SOAP-ENV:Envelope/SOAP-ENV:Body/*//Symbol[text()="DEF"]`
(iv): `//t:Transaction/@SOAP-ENV:mustUnderstand`
(v): `//t:Transaction[@SOAP-ENV:mustUnderstand=1]`
(vi): `//Stock/Symbol[text()="GHI"]`
(vii): `//Stock/Price[text()>15]`
(viii): `//Stock[Symbol/text()="GHI"][Price/text()>15]`

Figure 3.3: Sample XPath expressions.

Some sample XPath expressions are shown in Figure 3.3. XPath expression (i) designates documents that have two consecutive nodes `GetLastTradePrice` and `Symbol` at any level in the document (the initial `//` specifies that any number of nodes can appear before the first element). XPath expression (ii) additionally constraints the value of the `Symbol` node to be equal to `"DEF"` (`text()` selects the text node below the context node). Both expressions match the XML document of Figure 3.2 and return the `Symbol` node. XPath expression (iii) designates SOAP messages that have a `Symbol` node with value `"DEF"` at least two levels deep inside the message's body (below the `Body` node). Note that this expression does not start by `//` and thus specifies an absolute path from the root of the XML document. It matches the XML documents of Figure 3.2 and 3.4(a).

Attributes are specified in XPath expressions in a very similar way to tag nodes, but are prefixed with `@`. XPath expression (iv) designates documents that have a `Transaction` node with a `mustUnderstand` attribute, and XPath expression (v) further mandates this attribute to have the value `"1"`.

XPath can be used to express more complex filters where structural constraints are not limited to single paths. Consider a filter that selects SOAP messages with quotes for symbol `"GHI"` that have a price higher than `"15"`. The first constraint can be expressed with XPath expression (vi) of Figure 3.3, and the second one with expression (vii). A simple conjunction of these two XPath expression is not sufficient, however, to obtain the desired filter: the document of Figure 3.4 (a) does contain both expressions (matching paths are highlighted), but the price that matches the second expression is not that of symbol `"GHI"`. The correct filter must further constrain the matching `Symbol` and `Price` nodes to share the same `Stock` parent node. Such structural constraints are achieved using tree-structured expressions, which are expressed in XPath by defining multiple predicates on the same node. XPath expression (viii) is one possible embodiment of the desired filter. A tree representation of that expression is shown in Figure 3.4 (b). Note that it does not match the XML document of Figure 3.4 (a).

Subscription containment

We recall the definition of subscription containment: We say that a subscription S_1 *contains* or *covers* another subscription S_2, denoted by $S_1 \supseteq S_2$, iff any event matching S_2 also matches S_1,

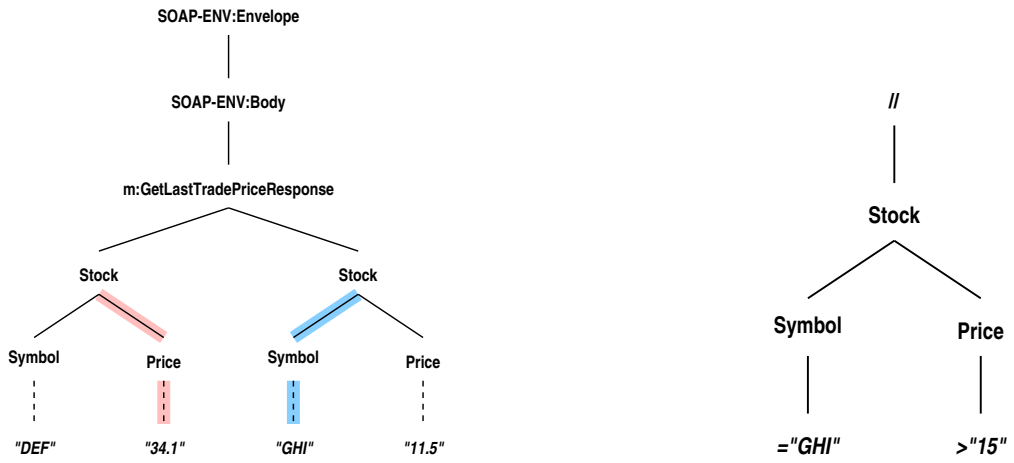


Figure 3.4: (a) SOAP message with multiple response values (b) Tree representation for XPath expression (viii) of Figure 3.3

i.e., $matches(S_2) \Rightarrow matches(S_1)$. The covering relationship defines a partial order on the set of all subscriptions.

3.3 The Filtering Engine: XTrie

XNet uses the filtering engine XTrie for efficient matching of events against large number of subscriptions. The XTrie implementation is based on a previous work done by C.-Y. Chan, P.A. Felber, M. Garofalakis, and R. Rastogi [35]. For that reason, we will only give in this section an overview of the algorithm and the data structures. A detailed study can be found in the aforementioned papers.

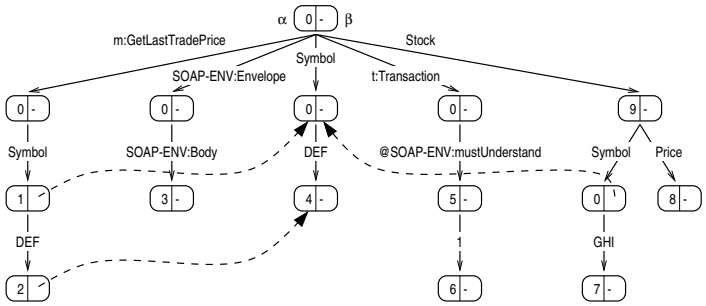
3.3.1 Overview

XTrie is a novel index structure that supports the efficient filtering of XML documents based on XPath expressions. The XTrie index structure offers several novel features that make it especially attractive for Web Services with strong scalability and performance requirements. First, XTrie is designed to support effective filtering based on complex, tree-structured XPath expressions (as opposed to simple, single-path specifications). Second, the XTrie structure and algorithms are designed to support both ordered and unordered matching of XML data. Third, by indexing on sequences of elements organized in a trie structure and using a sophisticated matching algorithm, XTrie is able to both reduce the number of unnecessary index probes as well as avoid redundant matchings, thereby providing extremely efficient filtering.

3.3.2 Trie structure

An exhaustive description of the XTrie algorithms can be found in [35]. Informally, it works as follows. XPath expressions are first normalized. During this phase, attributes and values used for equality comparisons are treated as regular nodes, with the exception that the former are prefixed by `@` and the latter are represented within quotation marks. Values used for comparisons other that equality are discarded. For instance, XPath expression (viii) of Figure 3.3 becomes `//Stock[Symbol/"GHI"][Price]`.

Normalized expressions are then decomposed into substrings. A substring is a non-empty sequence of elements that are separated by the parent/child operator (`/`), optionally prefixed by an ancestor/descendant operator (`//`) and wildcards (`*`). The decomposition of an XPath expression into substrings is generally not unique. For instance, { `//Stock`, `Symbol`, `"GHI"`, `Price` } and { `//Stock/symbol/"GHI"`, `//Stock/Price` } are two valid decompositions of XPath expression (viii).



(a)

	Parent Row	Rel Level	Rank	Num Child	Next	
1	0	$[2, \infty]$	1	0	0	`//m:GetLastTradePrice/Symbol`
2	0	$[3, \infty]$	1	0	0	`//m:GetLastTradePrice/Symbol/DEF`
3	0	$[2, 2]$	1	1	0	`SOAP-ENV:Envelope/SOAP-ENV:Body`
4	3	$[3, \infty]$	1	0	0	`*//Symbol/DEF`
5	0	$[2, \infty]$	1	0	0	`//t:Transaction/@SOAP-ENV:mustUnderstand`
6	0	$[3, \infty]$	1	0	0	`//t:Transaction/@SOAP-ENV:mustUnderstand/1`
7	0	$[3, \infty]$	1	0	10	`//Stock/Symbol/GHI`
8	0	$[2, \infty]$	1	0	11	`//Stock/Price`
9	0	$[1, \infty]$	1	2	0	`//Stock`
10	9	$[3, 3]$	1	0	0	`//Stock/Symbol/GHI`
11	9	$[2, 2]$	2	0	0	`//Stock/Price`

(b)

Figure 3.5: XTrie example for XPath expressions of Figure 3.3. (a) Trie. (b) Auxiliary table.

XTrie probes its data structures only when a substring is completely matched. Thus, longer substrings reduce the number of costly probes and improve performance. We therefore use a decomposition, called "simple decomposition", which creates the minimal number of longest substrings necessary for covering the XPath expression. In addition, to ensure correctness of the matching algorithm, the simple decomposition creates additional substrings at branching nodes when required. The simple decomposition of XPath expression (viii) is { `//Stock`, `//Stock/symbol/"GHI"`, `//Stock/Price` }, where the first substring is required for correctness. The substrings of the simple decomposition can be organized into a unique rooted tree, called the substring tree, where a substring s is the parent of a substring t if (1) s is a prefix of t or (2) the last node of s is the parent of the first node of t.

The substrings obtained from XPath expressions are then organized in a sophisticated trie structure and an auxiliary table. The trie allows for space-efficient indexing and time-efficient retrieval of XPath expressions, while the table stores additional information used for detecting valid matches. Figure 3.5 shows the trie and auxiliary table for the XPath expressions of Figure 3.3.

The auxiliary table contains one row for each substring of each indexed XPath expression (substrings are shown on the right of the table, next to their associated row). The rows are physically clustered such that all the substrings of a given XPath expression are stored consecutively. The rows are also logically partitioned into blocks containing identical substrings. This partitioning is achieved by chaining rows in a linked list using the "Next" pointer. Each row contains a pointer ("Parent Row") that holds the index of the row of its parent in the substring tree, or 0 for the root substring. "Rel Level" specifies the minimum and maximum depths at which a given substring must appear in the XML document, respective to its parent. For instance, for a range of $[2, 2]$, the last element of a given substring must occur exactly 2 levels deeper than the last element of its parent. Finally, the "Rank" value indicate the position of a substring respective to its siblings, and the "Num Child" specifies the number of children substrings. Although not shown in the picture, predicates that were discarded when normalizing the XPath expressions are attached to the row of the associated substrings.

The trie is a rooted tree constructed from the set of distinct substrings in all XPath expressions, where each edge is labeled with some element name. We call *label* of a node the concatenation of all the edge labels on the path from the root to the node. Each substring has one node in the trie, whose label corresponds to the sequence of elements in the substring. As the trie factorizes substrings with common prefixes, its size remains generally small. Each node n has two special pointers, denoted $\alpha(n)$ and $\beta(n)$, respectively shown on the right and left parts of the trie nodes. If the label of n is associated with some substring, then $\alpha(n)$ points to the first row of the linked list associated with that substring. Otherwise, the pointer is null. $\beta(n)$ points to the internal node in the trie whose label is the longest proper suffix of the label of n.

3.3.3 Matching algorithm

Informally, the matching algorithm works as follows: the XML document is first parsed using a SAX parser. This parser makes a single pass on the XML document and reports occurrences of XML elements (start tags, text, etc.) to the application using an event-driven API. SAX does not construct an in-memory representation of the document, making it an ideal candidate for streaming data. The algorithm then tries to map sequences of start tags, attributes, and text value to paths in the trie by following the edges. For each entire substring found, i.e., when reaching a node n such that $\alpha(n) \neq 0$, the auxiliary table is used to verify positional constraints with respect to previously-matched parent and sibling substrings, as well as the associated predicates, of all the rows in the linked list rooted at $\alpha(n)$. The same procedure is repeated for the nodes pointed to by $\beta(n)$. Information about partially-matches XPath expressions is kept at runtime in a data structure that stores the occurrence, depth, and scope of substrings previously encountered. When encountering end tags, runtime information is updated and substring matches that go "out of scope" are invalidated. An XPath expression is completely matched when all its substrings have been encountered and the associated constraints are validated.

We have implemented several variants of the matching algorithms optimized for different types of XPath expressions: single-branch expressions, tree-structured expressions, and ordered tree-structured expressions (branch ordering must be preserved). The details of these variants are found in [35]. Also, the performance of the XTrie algorithm is discussed later in the thesis.

Chapter 4

Content routing with XRoute

4.1 Principles

4.1.1 Overview

In this chapter, we explain how content routing is achieved in our system with the XRoute routing protocol.

Our routing protocol has been designed to achieve several goals. First, it should lead to *perfect* routing of data in the network, i.e., an event is forwarded to a link only if it leads to an interested consumer. Second, routing should ideally be *optimal*, i.e., the link cost of routing an event should be no more than that of sending the event along a multicast tree spanning all the consumers interested in the event.

Third, the protocol should take advantage of subscription *aggregation* to minimize space and processing requirements at the nodes. Informally, subscription aggregation is a mechanism that enables us to reduce the size of the routing tables by detecting and eliminating subscription redundancies; it is a key technique to scale to very large populations of consumers in a pub/sub system.

Finally, the protocol should be efficient and allow consumers to register and cancel subscriptions at any time. In particular, canceling a subscription should leave the system in the same state as if the subscription were not registered in the first place.

XRoute is an application-layer routing protocol: it allows the system to achieve the routing of events from the producer to the consumers, based on the events' content and the subscriptions registered by the consumers. For that purpose, XRoute integrates a routing algorithm to route events in a distributed manner and a subscription management protocol to maintain routing tables consistent with the consumers population. The routing algorithm consists for the most part in event filtering and forwarding and relies on the XTrie algorithm for the former. The subscription management protocol is itself composed of a subscription advertisement scheme and a routing table update algorithm.

4.1.2 The routing algorithm

Routing works in a distributed manner. Each node N in the network contains in its routing table a set of entries that represent the subscriptions that its neighbor nodes are interested in. For each subscription S, node N maintains some information in its routing Table in the form "if match S, send to $N_1, N_2, \ldots$". When a node is a consumer node, it knows the consumers which are interested in receiving events matching S.

The process starts when a producer publishes an event at its producer node. Routing then proceeds in a distributed manner as indicated in Algorithm 1. The process ends when all consumer nodes that are interested in that event have received it.

Example 1. *Consider the network in Figure 4.1(a), with two publisher nodes P_1 and P_2, and three consumer nodes C_1, C_2, and C_3. The other nodes N_1, N_2, N_3, N_4, and N_5 are internal nodes.*

Algorithm 1 Routing algorithm at node N

1: **when** receive event e from N' via interface I_{up}
2: Run XTrie algorithm to identify subscriptions in routing table that e matches
3: Forward e to all downstream interfaces that have a matching subscription.
4: **end when**

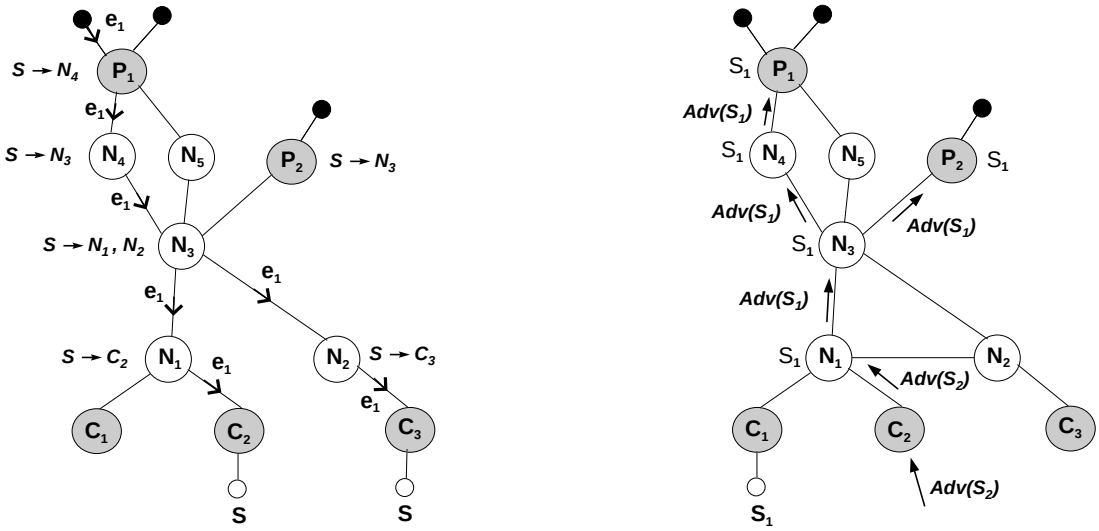


Figure 4.1: (a) A sample pub/sub network. Consumers (producers) are represented by white (black) circles. Subscriptions are represented underneath the consumers that registered them, and routing table entries are listed next to the node they are associated with. (b) Subscription advertisements are propagated upward from the consumers to the publishers. They may be transformed along the propagation paths due to aggregation (here, we have $S_1 \supseteq S_2$).

Nodes C_2 and C_3 have consumers interested in receiving events matching subscription S. Suppose that e_1, an event matching subscription S, is published at node P_1. Event e_1 will follow the path highlighted by the arrows.

4.1.3 The subscription advertisement algorithm

As previously mentioned, subscription advertisement is the mechanism that enables a router to learn about the subscriptions in which its neighbors are interested. Subscription advertisement in XNet works as follows.

When some consumer registers or cancels a subscription, it builds an advertisement corresponding to that subscription and sends it to its consumer node. Subscription advertisement then proceeds recursively as shown in Algorithm 2 and illustrated in Figure 4.1(b). The algorithm works by propagating advertisements recursively across the overlay, from the consumers toward the producers, following the best path, and updating routing tables along the way. Note that subscriptions may be transformed along the propagation path due to aggregation, i.e., a subscription received as part of an incoming advertisement may be different from the subscription carried by the resulting outgoing advertisement. The algorithm ends when the publisher node has been reached. When a subscription should be registered by multiple producers, the advertisements are sent along the paths to each of the producers. That issue is handled in a separate section.

4.1.4 Subscription Aggregation

Subscription aggregation is a key technique that allows us to minimize the size of the routing tables by eliminating redundancies between subscriptions, and hence to speed the routing process. The

Algorithm 2 Sketch of the advertisement protocol at node N

```
1: when receive adv(S) from N' via interface I_down
2:    update routing table
3:    generate outgoing advertisement adv(S')
4:    send adv(S') via I_up upward to the producer
5: end when
```

notion of subscription aggregation based on subscription containment was explained in Section 2.5.3. We recall its principle with the example illustrated in Figure 4.1(b): when an event e arrives at node N_3, it is only necessary to test e against S_1, because, by definition, any event matching S_2 also matches S_1, and any event that does not match S_1 does not match S_2 either. Hence, N_3 only needs to keep information about subscription S_1 in its routing table. Also, S_2 does not need to be propagated upstream from N_1 to N_3.

We distinguish between two forms of subscription aggregation. If S_1 and S_2 are registered through the same interface I^k (e.g., at Node N_3 in Figure 4.1(b)), we say that S_2 is *represented by* S_1 at interface I^k. If they are not registered through the same interface, we say that S_2 is *substituted by* S_1 (e.g., at Node N_1 in Figure 4.1(b)). In both situations, only S_1 is advertised upstream.

4.1.5 Impact on the routing process

When a consumer registers or cancels a subscription, the nodes of the overlay update their routing table accordingly by exchanging some pieces of information that represent the registration or cancellation of the consumer. The process starts at the consumer node and terminates at the producer node(s), following the shortest paths. As a consequence, messages published by the producers follow the reverse paths of the subscriptions, along a multicast tree spanning all interested consumers. Thus, routing is optimal. Also, there cannot be cycles because each node always receives events through its I_{up} interface located on the best path from the producer to the node, and never propagates them along that path. In the next section, we present the routing table update algorithm (RTU) that enables to update a router's routing table consequently to a received advertisement, and that makes use of subscription aggregation to minimize the size of the routing table.

4.2 XRoute: routing table update algorithm

We have seen in the previous section that a key requirement for achieving efficient routing with XRoute is to maintain accurate routing tables. Also, one design goal of XRoute is to minimize the sizes of the routing tables and to enable subscription cancellation while still ensuring perfect routing. The last point is especially challenging. In fact, ensuring perfect routing and minimizing routing tables sizes can be done relatively easily. However, enabling subscription cancellation while maintaining accurate and small routing tables is a challenging point. This is achieved in XRoute by the RTU algorithm.

4.2.1 Data formats

We first present the format of the different pieces of data that are used by the RTU algorithm.

Routing Tables

Each node N maintains a routing table that consists of a set of entries. Each entry corresponds to one *distinct* subscription (two identical subscriptions share the same entry). We will write $entry(S)$ to refer to the entry corresponding to subscription S. It maintains information about all the registrations for subscription S that have been received by node N. More precisely, the information in $entry(S)$ represents N's view of its neighbor's interests in subscription S. Moreover, $entry(S)$ also contains the information required to implement the aggregation principle introduced in Section 4.1.4.

An entry $entry(S)$ in the routing table of node N has the following format:

$$S\ ;\ (T_S^1, \cdots, T_S^n)\ ;\ R_S\ ;\ P_S$$

where S is the subscription and n is the number of interfaces of node N. T_S^k represents the population of consumers downstream interface I^k that are interested in events matching S. Each T_S^k consists of a set of two integers that we will refer to as $T_S^k.x$ and $T_S^k.z$ (to be described shortly). $\overline{T_S^k}$ is defined by $T_S^k.x + T_S^k.z$ and is always greater than or equal to 0. It is strictly greater than 0 iff there are consumers downstream interface I^k interested in receiving events matching S. R_S represents the total number of subscriptions that have been "aggregated" in S, either through representation or substitution. P_S, if non-null, points to another entry in the routing table that S is substituted by.

$T_S^k.x$ represents the population of subscriptions S downstream interface I^k, i.e., the number of consumers interested in receiving events matching S. $T_S^k.z$ corresponds to the number of subscriptions that are represented by S at interface I^k. $T_S^k.z$ is also equal to the number of subscriptions that are "aggregated" in S ,either through representation or substitution, at the node downstream interface I^k (this non-trivial property follows from the RTU algorithm and will be demonstrated shortly).

Advertisements

As mentioned previously, advertisement messages are exchanged between routers to register or cancel a particular subscription. From the point of view of node N, receiving an advertisement message $adv(S)$ from interface I^k means that a change about the population of subscriptions S has occurred downstream interface I^k. Node N must update its routing table to take this change into account; in particular, T_S^k needs to be updated. N also needs to generate and send an advertisement to the upstream neighbor node.

An advertisement message $adv(S)$ is a sequence of triples with the following format:

$$S\ ;\ n_S\ ;\ r_S$$

where S is the subscription advertised, and n_S is the number of times S should be registered ($n_S > 0$) or canceled ($n_S < 0$). r_S represents the number of subscriptions, distinct from S, that have been substituted by S downstream I^k, and that should be registered ($r_S > 0$) or canceled ($r_S < 0$) at node N. Finally, $adv(S)$ may contain additional triples, with the same format, indicating additional modifications to perform to the routing tables upstream.

In the case where $n_S < 0$, we distinguish between three different cases according to the value of $|n_S|$:

- If $|n_S| = \sum_k T_S^k.x$, we say that subscription S is being *totally cancelled* at interface I^k of node N. Practically, that means that there are no more consumers interested in subscription S downstream interface I^k.
- If $|n_S| = \sum_k T_S^k.x$ and $\forall j \neq k, \overline{T_S^j} = 0$, we say that subscription S is being *totally cancelled* at node N. In other words, there are no more consumers interested in subscription S downstream node N (all interfaces).
- If $|n_S| < \sum_k T_S^k.x$, we say that subscription S is being *partially cancelled* at node N. There are still some consumers interested in S downstream interface I^k.

4.2.2 Representation and Substitution

Before describing the RTU algorithm, we need to describe more formally the representation and substitution relations that we introduced in Section 4.1.4, and how they are implemented.

Definition 1 (Representation). *Consider entries for subscriptions S_1 and S_2 at non-consumer node N such that $S_1 \supset S_2$, $\overline{T_{S_1}^k} > 0$ and $\overline{T_{S_2}^k} > 0$, then S_2 must be* represented by *S_1 at interface I^k. This operation consists in modifying their entries as follows:*

1. $T_{S_1}^k.z \leftarrow T_{S_1}^k.z + \overline{T_{S_2}^k}$

2. $R_{S_1} \leftarrow R_{S_1} + \overline{T^k_{S_2}}$
3. $R_{S_2} \leftarrow R_{S_2} - T^k_{S_2}.z$
4. $\overline{T^k_{S_2}} \leftarrow 0$

Thereafter, we say that S_2 is represented by S_1 at interface I^k.

The representation operation implements the subscription aggregation mechanism introduced in Section 4.1.4. Indeed, having both $\overline{T^k_{S_1}}$ and $\overline{T^k_{S_2}}$ greater than zero is redundant, because it is not necessary to test an event against S_2 to know if it has to be forwarded down that interface. Therefore, when S_2 has been represented by S_1 at interface I^k, $\overline{T^k_{S_2}}$ becomes null, which is equivalent to say that no client is interested in receiving events matching S_2 downstream interface I^k. If $\overline{T^k_{S_2}}$ is null for all k, then $entry(S_2)$ can be removed from the routing table.

Note that if some subscriptions were previously represented by S_2 at interface I^k, they now become represented by S_1 at I^k. Indeed, $\overline{T^k_{S_2}}$ represents the sum of the instances of S_2 registered at I^k *and* all the subscriptions that are represented by S_2 at I^k. At the time S_2 is represented by S_1 at I^k, S_1 takes control of all instances of S_2 and all the subscriptions that it represents (steps 1 and 2 in Definition 1), and S_2 loses control of the subscriptions it used to represent (steps 3 and 4).

The representation relation enables to minimize routing table sizes. Indeed, when subscription S_2 is being represented by subscription S_1 at interface I^k, this results in the field $T^k_{S_2}$ being reset. Consequently, if the other $T^j_{S_2}$ are null, entry for subscription S_2, $entry(S_2)$, can be deleted. If one is not null, say $T^j_{S_2}$, it can be reset by a future subscription representation at interface I^j.

An example is illustrated in figure 4.2

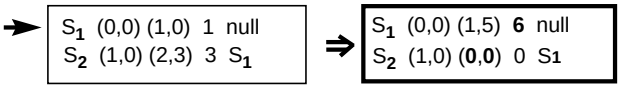


Figure 4.2: Example of a representation operation. S_2 is being represented by S_1 at interface I_2. The updated routing table is shown with a thick frame.

Definition 2 (Substitution). *Consider entries for subscriptions S_1 and S_2 at node N such that: $S_1 \supset S_2$, $P_{S_1} = null$, and $P_{S_2} = null$. Then S_2 must be* substituted by *S_1. This operation consists in modifying their entries as follows:*

1. $P_{S_2} \leftarrow S_1$
2. $R_{S_1} \leftarrow R_{S_1} + \sum_{k \leq n} T^k_{S_2}.x + R_{S_2}$

Thereafter, we say that S_2 has been substituted by S_1, and S_2 must subsequently be advertised by S_1, i.e., any incoming advertisement $(S_2; n; r)$ yields an outgoing advertisement $(S_1; 0; n + r)$. Note that a subscription may be substituted by only one other subscription.

The signification of a substitution operation can be understood by observing the following scenario. Suppose that the conditions for substituting S_2 by S_1 are met, but we do not perform the substitution operation. If an incoming advertisement for S_2 (registering n_{S_2} subscriptions) arrives at node N, the outgoing advertisement sent to the upstream neighbor node N' at interface I^j will be $adv_{up}(S_2)$. Then, S_2 will be represented by S_1 at interface I^j of N'. Thus, by substituting S_2 by S_1 at node N, we anticipate this representation. The outgoing advertisement is composed of the triple S_1 ; 0 ; n_{S_2}; it advertises S_1 and specifies that S_1 is to represent n_{S_2} additional subscriptions at interface I^j.

Although it adds some complexity to the protocol, the subscription substitution mechanism is *necessary* to guarantee perfect routing when canceling a subscription that acts as a substitute for some other subscriptions. In addition, the substitution mechanism can help save bandwidth by propagating smaller advertisements.

Note that there may be multiple substitution relations between subscriptions. That is, subscription S can be substituted by S', which is in turn substituted by S'', etc. We call such a sequence a *substitution chain*. For any subscription S_i, we denote by $h(S_i)$ the subscription at the top of the chain, i.e., the subscription S with $P_S = null$. We denote by $tree(S)$ the set of all the subscriptions S_j that have been substituted, directly or indirectly, by S (including S). Figure 4.3 shows a subscription tree, where links represent substitutions (the child is substituted by its parent). For instance, $tree(S_1)$ contains all subscriptions, $tree(S_3)$ contains S_3, S_4, and S_5, and $tree(S_5)$ only contains S_5.

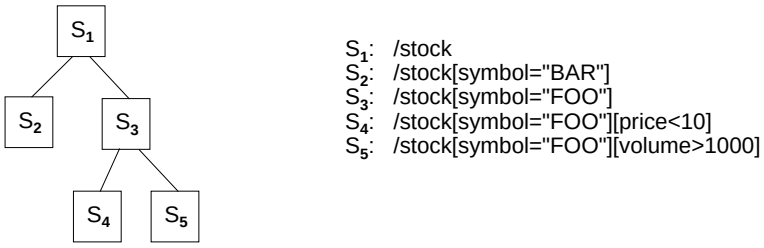


Figure 4.3: The substitution relations apply recursively. Subscriptions can be organized in a tree, where a link indicates that a child is substituted by its parent.

A substitution operation can only be performed between two subscriptions if none of them has already been substituted, in other words between two tops of chains. An example is illustrated in figure 4.4.

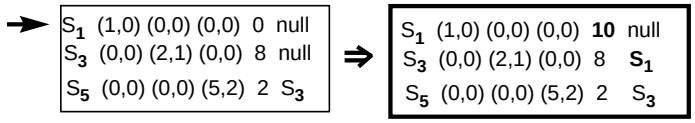


Figure 4.4: Example of a substitution operation, applied to subscriptions S_1, S_3 and S_5 of Figure 4.3. S_3 is being substituted by S_1. Note that S_5 was previously substituted by S_3. The updated routing table is shown with a thick frame.

4.2.3 Properties of the representation and substitution relations

We now give several properties of the representation and the substitution operations, which will be useful throughout the rest of the section.

Consider node N. Let node N_{down} be the node downstream interface I^k and N_{up} be the upstream neighbor node (upstream interface I_{up}).

Property 1. *When an advertisement for the registration of subscription S arrives from node N_{down} at interface I^k of node N, S* cannot *be represented by any subscription at that interface.*

Proof. Suppose that S can be represented by subscription S' at interface I^k. That means that we have $\overline{T^k_{S'}} > 0$, which means that S' has an entry at node N_{down}. But then at that node, subscription S would have been substituted by S' and no advertisement for S would have reached node N. □

Corollary: It follows from Property 1 that the only case when a subscription S_2 can be represented by another subscription S_1 ($S_1 \supset S_2$) is when $\overline{T^k_{S_1}} = 0$ and an advertisement for S_1 arrives at interface I^k. Then $\overline{T^k_{S_1}}$ becomes strictly positive and S_2 is represented by S_1 at interface I^k. In other words, an advertisement for S_2 has arrived before the advertisement for S_1.

Property 2. *At node N, $\overline{T_S^k} > 0$ iff S has not been substituted at the node downstream interface I_k (N_{down}).*

Proof. Suppose that, at node N_{down}, subscription S has been substituted by another subscription S'. We suppose that S' is not substituted (otherwise, the demonstration applies to the root of the substitution tree to which S belongs). Then, S' has an entry at node N such that $\overline{T_{S'}^k} > 0$ (because if S' is not substituted, advertisements for S' also yields to outgoing advertisements for S'). Then, at node N, S must be represented by S' and consequently we have $\overline{T_S^k} = 0$.

Similarly, if S has not been substituted at node N_{down}, then advertisements for S also yields to outgoing advertisements for S, and $\overline{T_S^k} > 0$ at node N. □

Property 3. *Suppose that S_1 is not substituted by another subscription. Then, if S_2 is substituted by S_1 at node N, S_2 is represented by S_1 at the upstream neighbor node N_{up}, at incoming interface I^j (toward N).*

Proof. Suppose that S_2 arrived at node N before S_1. Then S_2 also has an entry at the upstream node N_{up}, because it has not been substituted at node N. Because S_1 is not substituted, the outgoing advertisement advertises S_1. When it arrives at node N_{up} the conditions for representing S_2 by S_1 are met. Now suppose that S_1 arrived first at node N. Then when S_2 is substituted by S_1, according to the definition of the substitution relation, the incoming advertisement for S_2, $(S_2; n; r)$ yields an outgoing advertisement for S_1: $(S_1; 0; n + r)$. That means that S_1 is to represent $n + r$ instances of subscription S_2 at node N_{up}.

□

Corollary: Property 3 is true for any subscription S_j that belongs to the substitution tree $tree(h(S_j))$: S_j is represented by $h(S_j)$ at the upstream neighbor node, at incoming interface I^j (toward N).

Property 4. *Consider subscription S. R_S represents the number of subscriptions that are represented by S at all interfaces, plus the number of instances of each of the subscriptions on $tree(S)$, plus the number of the subscriptions that any of them represent at any interface.*

Proof. According to the definition of the representation relation, R_S is incremented by $\overline{T_{S'}^k}$ when S' is being represented by S at interface I^k. Thus R_S represents at least the instances of all the subscriptions that are represented by S at all the interfaces.

Suppose that the height of $tree(S)$ is one. In other words, no subscriptions are substituted by S. Then the only way to increase R_S is by representation relations. Thus R_S represents exactly the instances of all the subscriptions that are represented by S at all the interfaces, and the property is true ($tree(S)$ comprises S only).

Now suppose that the height of $tree(S)$ is two. In other words, if $\{Substituted\}$ is the set of all the subscriptions that are substituted by S, no subscriptions are substituted by a subscription in $\{Substituted\}$. Consider $S_1 \in \{Substituted\}$. Then $tree(S_1)$ is of height one, and R_{S_1} is exactly the number of subscriptions that are represented by S_1 at all the interfaces. When S_1 was substituted by S, R_S was increased by $\sum_{k \leq n} T_{S_1}^k.x + R_{S_1}$. Thus in R_S are included all the instances of subscription S_1 plus the instances of the subscriptions that are represented by S_1 at all the interfaces. This is true for all the subscriptions in $\{Substituted\}$, and the property is true.

Now suppose that the property is true for a height of h (recursive hypothesis). If the height of $tree(S)$ is $h+1$, then again let $\{Substituted\}$ be the set of all the subscriptions that are substituted by S. We have seen that R_S represents at least the instances of all the subscriptions that are represented by S at all the interfaces. Now for each $S_1 \in \{Substituted\}$, $tree(S_1)$ is of height at most h. When S_1 is substituted by S, R_S is increased by $\sum_{k \leq n} T_{S_1}^k.x + R_{S_1}$, that is by all the instances of subscription S_1 plus R_{S_1}. Because $tree(S_1)$ is of height at most h, according to the recursive assumption, R_{S_1} is exactly the number of subscriptions that are represented by S_1 at all the interfaces, plus the number of all the subscriptions on $tree(S_1)$, plus the number of the subscriptions that any subscription on

$tree(S_1)$ represents at any interface. Thus the property is true for a height $h+1$, and according to the theorem of recursivity, it is always true. □

Property 5. *At node N, for any subscription S, the $T_S^k.z$ field is equal to the R_S field of entry(S) at node N_{down}.*

Proof. This property comes from Property 4 and the operation of the routing protocol. Indeed, the update of the routing table is such that $T_S^k.z$ corresponds to the number of subscriptions "aggregated" in S, either through representation or substitution at node N_{down}. Then according to property 4, R_S at node N_{down} represents exactly the number of those subscriptions and is hence equal to $T_S^k.z$ at node N. □

Corollary: From this property, it follows that part of the subscriptions that are represented by S at interface I^k of node N are subscriptions S_i at node N_{down} such that P_{S_i} points to S. The other part consists of subscriptions that are represented by S at all the downstream interfaces of node N_{down}. Recursively we can completely identify them, because there are no representations at client nodes.

Property 6. *When an advertisement for the registration of subscription S_2 arrives at node N, if S_2 can be substituted by another subscription S_1, then no subscription can be substituted by S_2.*

Proof. Suppose that a subscription S_3 can be substituted by S_2. This implies that S_3 is not substituted by a subscription yet. Also this implies that $S_3 \subset S_1$ (because of the transitivity property of the containment relation). Then S_3 would have been substituted by S_1. □

4.2.4 Correctness of subscription advertisement

It trivially appears from Algorithm 2 that subscription advertisement is correct when subscriptions are not aggregated. We now show that subscription aggregation does not affect the correctness of subscription advertisement.

Consider a node N with upstream neighbor N_{up}. Suppose that S_2 is substituted by S_1 in the routing table of node N ($S_1 \supseteq S_2$). Then, because of Property 3, at node N_{up}, S_2 is represented by S_1 at the incoming interface. Now consider an event e that arrives at node N_{up}. If e matches S_1, then it is forwarded at node N, where we have at least S_1 as a matching subscription. Now if e does not match S_1, then because $S_1 \supseteq S_2$, it does not match S_2 either. Hence, event e is not forwarded to node N, where there are no matching subscriptions. Hence subscription advertisement is correct at node N_{up}. At node N, S_2 and S_1 have been registered through different interfaces, and both have an entry in the routing table. Hence, subscription advertisement is correct at node N.

Now suppose that at node N, S_2 is represented by S_1 at interface I^k. Let N_{down} be the neighbor downstream interface I^k. If S_2 has an entry in the routing table of node N_{down}, then it is substituted by S_1 and similarly to the previous case, subscription advertisement is correct at node N. If S_2 does not have an entry in the routing table of node N_{down}, then subscription advertisement is trivially correct.

In fact, there necessarily exists a node N_d downstream I^k, where S_2 has been substituted by S_1. N_d is the first node where S_1 and S_2 have been registered through different interfaces. N_d necessarily exists, because if S_2 and S_1 were always registered through the same interface, then they were registered at the same consumer node C, where S_2 have been substituted by S_1, since representation relations are not permitted at consumer nodes.

For example, consider the network illustrated in Figure 4.5, which results from the registration of subscription S_2 at node C_2, as illustrated in Figure 4.1(b). At node N_1, S_2 has been substituted by S_1. Consequently, because of Property 3, at node N_3, S_2 is represented by S_1 at the incoming interface. This results in S_2 not having an entry in the routing table of node N_3. Similarly, S_2 does not have an entry at nodes N_3, N_4, P_1 and P_2. An event e published at P_1, and that matches S_1 is forwarded to nodes N_4, N_3 and N_1. Now if e does not match S_1, then it does not match S_2 either, and is not forwarded by node P_1.

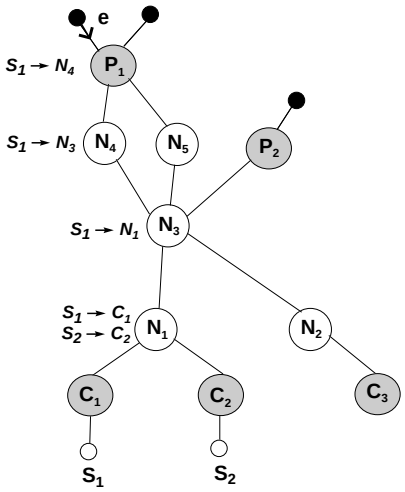


Figure 4.5: Subscription aggregation does not affect perfect routing.

4.2.5 RTU protocol description

We now describe formally the routing table update protocol. Updating the routing table constitutes the main task of the subscription algorithm. The table must be updated at node N each time an advertisement for a subscription S arrives from an interface I^k, i.e., when a change has occurred in the population of the subscriptions S downstream interface I^k. The routing table at node N must be updated so that its entries are accurate enough to enable perfect routing. Moreover, the algorithm must make full use of subscription aggregation at all times. The details of the algorithm are given in Algorithms 3, 4, 5, and 8, 7 and 6 and described in the rest of this section.

Algorithm 3 — Routing Table Update

1: **if** $P_S \neq null$ **then**
2: **for all** S' ancestor of S in $tree(h(S))$ **do**
3: $R_{S'} \leftarrow R_{S'} + n_S + r_S$
4: **end for**
5: $adv_{out} \leftarrow (h(S); 0; n_S + r_S)$
6: **else**
7: $adv_{out} \leftarrow (S; n_S; r_S)$
8: **end if**
9: $T_S^k.x \leftarrow T_S^k.x + n_S$
10: $R_S \leftarrow R_S + r_S$
11: $T_S^k.z \leftarrow T_S^k.z + r_S$
12: Send adv_{out} upstream

Dealing with Registrations

We first focus on the case of subscriptions *registrations*. Consider node N that receives from downstream interface I^k a *registration* advertisement for a subscription S: $(S; n_S; r_S)$. The routing table update consists in updating T_S^k and in trying to establish and/or modify some relations (substitution or representation) between S and the other subscriptions in the routing table of node N. The process is different according to the value of $entry(S)$ in the routing table.

First case: $entry(S)$ exists and $\overline{T_S^k} > 0$. Algorithm 3 is executed. Then, some advertisement for the registration of S has been received earlier at interface I^k. Hence, the possible aggregation relations (i.e., representation or substitution relations) between S and the other subscriptions have already been established. Consequently, the process consists in updating T_S^k (lines $9-11$) and in

Algorithm 4 — Subscription Representation

1: declare $A = 0$
2: **for all** S_j subscriptions that can be represented by S at I^k **do**
3: declare $T_j = \overline{T^k_{S_j}}$
4: Represent S_j by S at I^k
5: **if** $S_j \in tree(S)$ **then**
6: **for all** S_k ancestor of S_j in $tree(S)$ **do**
7: $R_{S_k} \leftarrow R_{S_k} - T_j$
8: **end for**
9: **else**
10: **for all** S_k ancestor of S_j in $tree(h(S_j))$ **do**
11: $R_{S_k} \leftarrow R_{S_k} - T_j$
12: **end for**
13: **if** $S_j \notin tree(h(S))$ **then**
14: append $(h(S_j); 0; -T_j)$ to adv_{out}
15: $A \leftarrow A + T_j$
16: **end if**
17: **for all** S_k ancestor of S in $tree(h(S))$ **do**
18: $R_{S_k} \leftarrow R_{S_k} + T_j$
19: **end for**
20: **end if**
21: **if** $\sum_{p \leq n} \overline{T^p_{S_j}} = 0$ **then**
22: remove $entry(S_j)$
23: **for all** S_k such that $P_{S_k} = S_j$ **do**
24: $P_{S_k} \leftarrow P_{S_j}$
25: **end for**
26: **end if**
27: **end for**
28: **for all** S_k ancestor of S in $tree(h(S))$ **do**
29: $R_{S_k} \leftarrow R_{S_k} + n_S + r_S$
30: **end for**
31: $T^k_S.x \leftarrow T^k_S.x + n_S$
32: $R_S \leftarrow R_S + r_S$
33: $T^k_S.z \leftarrow T^k_S.z + r_S$
34: **if** $h(S) \neq S$ **then**
35: $adv_{out} \leftarrow (h(S); 0; n_S + r_S + A)$ [+ appended triples]
36: **else**
37: $adv_{out} \leftarrow (S; n_S; r_S + A)$ [+ appended triples]
38: **end if**
39: Send adv_{out} upstream

the case where S is substituted by another subscription, the entries of some subscriptions in the substitution chain to which S belongs (lines $1-6$).

More precisely, the process proceeds as follows. We increment R_S and $T^k_S.z$ by r_S to take into account the r_S additional subscriptions that S represents at interface I^k (lines $10-11$). We also increment $T^k_S.x$ by the n_S additional instances of subscription S that were carried by the advertisement (line 9). Then, if S has been substituted, we increment the R field of the subscriptions ancestor of S in $tree(h(S))$ by $n_S + r_S$ (lines $2-4$). Indeed, those subscriptions are now substitutes for $n_S + r_S$ additional subscriptions.

Finally, we build the outgoing advertisement as the result of the update that has been done, and send it upstream (lines 5, 7 and 12).

Second case: $entry(S)$ exists and $\overline{T^k_S} = 0$. Algorithm 4 is executed.

Then, some advertisement for the registration of S has been received earlier, but not through interface I^k. Consequently, all the possible substitution relations have already been established. However, there may be some possible representation relations at interface I^k, between S and some other subscriptions. Moreover, we have seen in Property 1 that S cannot be represented by another subscription. Hence, we look for all the subscriptions that can be represented by S at interface I^k, and perform the representation operation (lines $2-4$). This has for consequence that some existing relations may need to be modified (lines $5-20$). This is due to the fact that a subscription can both have at the same time a substitution relation with another subscription, and multiple representation relations with other subscriptions.

When the necessary modifications have been done, we look for the subscriptions that have just been represented by S at interface I^k and that have a null entry (as a result of the representation operation), and we remove them from the routing table (lines 21−22). Consider one such subscription S_j. Then, the subscriptions that were directly substituted by S_j must now be substituted by another subscription, since S_j does not exist anymore. P_{S_j} is a potential candidate. Indeed, since S_j was represented by S, we have $S_j \subseteq S$. Also, since both S_j and S already had an entry in the routing table, S_j was necessary substituted by another subscription (S is a potential candidate). Hence, $P_{S_j} \neq null$. Consequently, all the subscriptions that were directly substituted by S_j can now be substituted by P_{S_j} (lines $23 - 25$).

Finally, we proceed as in the previous case: we update T_S^k (lines $31 - 33$) as well as the entries of the subscriptions ancestor of S in $tree(h(S))$ (lines $28-30$), and we build the outgoing advertisement and send it to the upstream interface (lines $34 - 39$).

We now detail the modifications to some existing relations that were made consequently to the representation relations that were established between S and the other subscriptions. Consider the case where a subscription S_j is to be represented by S at interface I^k. There are $T_j = \overline{T_{S_j}^k}$ instances of subscription S_j. We have two cases:

First case: $S_j \in tree(S)$. This case is illustrated in Figure 4.6(a). The T_j instances of subscription S_j are now represented by S at interface I^k. For each subscription S_k ancestor of S_j in $tree(S)$, the T_j instances of subscription S_j are no longer substituted by S_k. Thus, because of Property 4, subscription S_k must have its R field decremented by T_j (lines $5 - 8$).

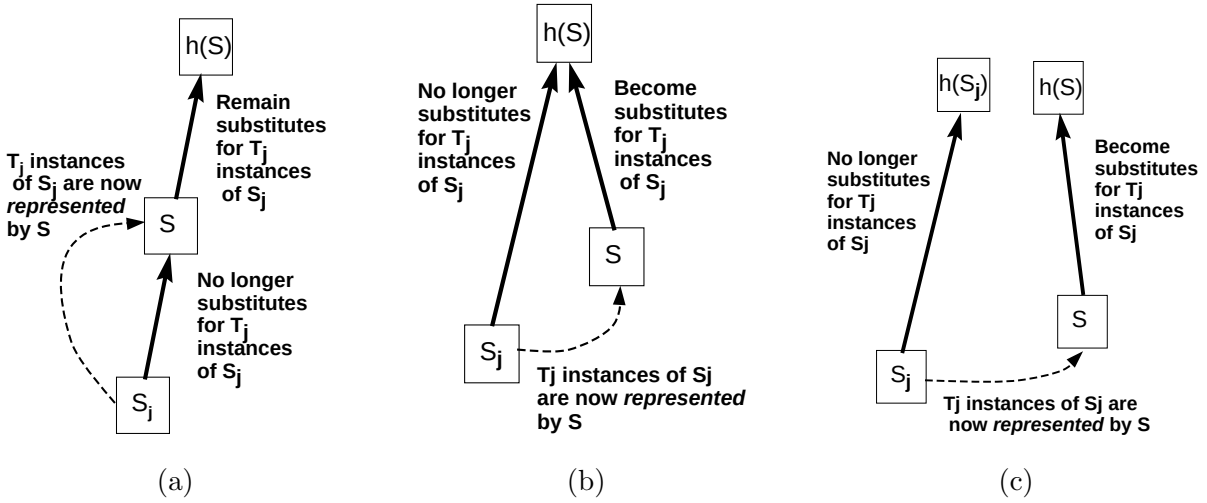


Figure 4.6: Representation of the possible substitution trees of subscriptions S_j and S. Subscriptions are represented within boxes. A plain straight arrow originating from subscription A and pointing to subscription B represents all the subscriptions ancestor of A and descendant of B (A included) in the substitution tree rooted at B. The curved dashed arrow indicates the new *representation* relation that has just been established between subscription S_j and subscription S. The modifications that are induced subsequently are indicated next to the subscriptions concerned. (a) $S_j \in tree(S)$ (b) $S_j \notin tree(S)$ and $h(S_j) = h(S)$ (c) $S_j \notin tree(S)$ and $h(S_j) \neq h(S)$

However, the subscriptions ancestor of S in $tree(h(S))$ (if any) are still a substitute for the T_j instances of subscription S_j, and do not need to have their entry modified.

Second case: $S_j \notin tree(S)$. This case is illustrated in Figures 4.6(b) and (c). Then the T_j instances of subscription S_j (that are now represented by S at I^k) also have for substitutes every subscription ancestor of S in $tree(h(S))$ (if any). Thus those subscriptions must have their R field incremented by T_j (lines $17 - 19$). Also, all subscriptions ancestor of S_j in $tree(h(S_j))$ (if any) must have their R field decremented by T_j (lines $10 - 12$).

Now, consider the additional case where S_j does not belong to $tree(h(S))$ ($h(S_j) \neq h(S)$), as illustrated in Figure 4.6(c). Then, because of Property 3, at the incoming interface of the upstream neighbor node, the T_j instances of subscription S_j are represented by subscription $h(S_j)$ (note that we

necessarily have $h(S_j) \neq S_j$, otherwise S_j would have been substituted by S). This is incompatible with the fact that those T_j instances are now represented by S at node N. Thus, we must indicate that $h(S_j)$ should represent T_j fewer instances of subscription S_j at that node, whereas $h(S)$ should represent T_j additional instances of S_j. This information is appended to the outgoing advertisement in the form of two additional triples $(h(S); 0; T_j)$ and $(h(S_j); 0; -T_j)$ (lines $13-16$, 35 and 37).

Algorithm 5 — Subscription Substitution

```
1: create a null entry(S)
2: if ∃S', S' ⊃ S, P_S' = null then
3:    substitute S by S'
4: else
5:    for all S_k that can be substituted by S do
6:       substitute S_k by S
7:    end for
8: end if
9: call Algorithm 4: "Subscription Representation".
```

Third case: $entry(S)$ does not exist. Algorithm 5 is executed. Then, no advertisement for the registration of S arrived previously at node N. Hence, we must check for both substitution and representation relations between S and the other subscriptions in the routing table.

We first create a null entry for subscription S (line 1). Then, we try to substitute S by another subscription (lines $2-3$). If that is possible, then according to property 6, no other subscription can be substituted by S. Otherwise, we try to substitute other subscriptions by S (lines $5-7$).

Subsequently, we fall in the previous case. Indeed, $entry(S)$ exists, $\overline{T_S^k} = 0$, and all the possible substitution relations between S and the other subscriptions have been established (this is necessary since $entry(S)$ is null for now). We can then execute Algorithm 4 (line 9), to proceed with the remaining routing table updates.

Additional updates: The incoming advertisement may contain additional triples $(S'; 0; U)$. These triples are generated by Algorithm 4 (line 14) at the downstream neighbor node and are such that $U < 0$ and $P_{S'} = null$. We are thus in the case where $entry(S')$ exists and $\overline{T_{S'}^k} > 0$, and we can apply Algorithm 3 for each S'.

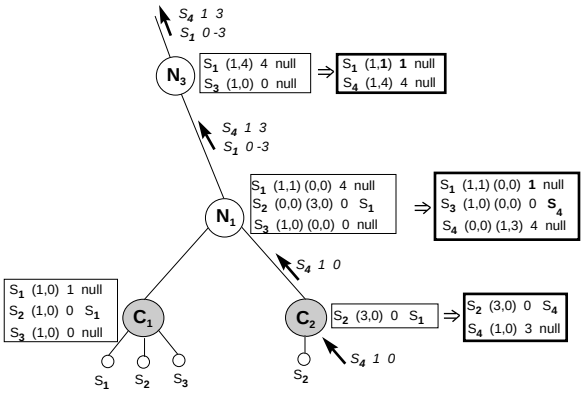


Figure 4.7: Example of the subscription algorithm. Registered subscriptions are represented below their corresponding client nodes. Routing tables (shown next to the nodes) are updated as a result of the registration of subscription S_4 (updated tables are shown with a thick frame).

Example 2. *Figure 4.7 illustrates the operation of the subscription algorithm on the pub/sub network of Figure 4.1(a). Four consumers have already registered some subscriptions. A consumer at client node C_2 registers subscription S_4, resulting in updates of the routing table at each node on the path*

from C_2 to each publisher. For the sake of clarity, we have only represented inner nodes N_1 and N_3. Also, we suppose that $S_0 \supseteq S_2$, $S_1 \supseteq S_2$, and $S_4 \supseteq S_3$, but there are no relationships between S_4 and S_1, and between S_2 and S_3.

At nodes C_2, N_1, and N_3, entry(S_4) does not exist. Thus, Algorithm 5 (which in turn calls Algorithm 4) is called to update the routing table. The following relations are established: At node C_2, S_2 is substituted by S_4. At node N_1, S_3 is substituted by S_4, S_2 is represented by S_4 at the downstream interface to C_2, and entry(S_2) is removed. At node N_3, S_3 is represented by S_4 at the downstream interface to N_1 and its entry is removed.

Case of consumer nodes

At consumer node, there is only one interface (see 3.2.1. Consumers send registrations and cancellations through that single interface. There cannot be representation relations between the subscriptions in the routing table of a consumer node. Indeed, if that happened, then we would "loose track" of the subscription represented, and we would not be able to maintain perfect routing in the case of subscription cancellations. Thus, the update of the routing table at consumer node is exactly the same as for inner nodes, except that Algorithm 4 is not called at line 3 of Algorithm 5.

Classification and dynamics of registrations

It results from the previous sections that we can classify an incoming advertisement Adv_{in} for the registration of subscription S through interface I^k, and the type of routing table updates that must be performed in four categories as follows:

- *Duplicate registration*: This corresponds to the case where *entry*(S) exists and $\overline{T_S^k} > 0$. In other words, subscription S was previously registered through interface I^k. We have seen that we do not need to compute containment relationships. The routing table at node N simply consists in updating routing table entries, which is a very fast operation.

- *Supplementary registration*: This corresponds to the case where *entry*(S) exists and $\overline{T_S^k} = 0$. In other words, subscription S was previously registered by another consumer, but through a different interface than I^k. We have seen that we may need to perform representation operations, which requires to determine the subscriptions in the routing table that are contained by S. As a consequence, the routing table update corresponding to a *supplementary registration* is significantly more complicated and costly than that of a *duplicate registration*.

- *New registration*: This corresponds to the case where *entry*(S) does not exists. We have seen that we need to determine all the possible containment relationships between S and the other subscriptions in the routing table. Consequently, the routing table update corresponds to a *new registration* is the costliest of all three registration types.

- *Additional triples :* We have seen that if additional triples $(S'; 0; U)$ are contained in adv_{in}, each of them is seen as a *duplicate registration*.

We now identify the relations between the type of an incoming registration advertisement at node N, adv_{in}, and that of the outgoing registration advertisement adv_{out} sent to the upstream node N_{up}. Those can be easily derived from the properties enounced in Section 4.2.3 and the operation of the RTU algorithm.

- If adv_{in} is a *duplicate registration* at node N: Then, adv_{out} is necessarily a *duplicate registration* for node N_{up} (whether S is directly advertised or not).

- If adv_{in} is a *supplementary registration* at node N: Then, adv_{out} is a *duplicate registration* at node N_{up}.

- If adv_{in} is a *new registration* at node N: Then, if S has been substituted by another subscription at node N, adv_{out} is a *duplicate registration* at node N_{up} (and $h(S)$ is advertised). Otherwise, adv_{out} is a *new registration.*

Dealing with Cancellations

We now deal with the case of subscription cancellations.

Problem statement. Consider node N and $entry(S)$ for subscription S its routing table. Let node N_{down} be the node downstream interface I^k and node N_{up} be the upstream neighbor node. Let $adv_{in}(S) = (S\ ;\ n_S\ ;\ r_S)$ be an advertisement for the *cancellation* of subscription S ($n_S < 0$), arriving at interface I^k from node N_{down}. If $|n_S| < T_S^k.x$, then S is being partially cancelled. This case is completely handled by Algorithm 3, except that $n_S < 0$.

Now if $|n_S| = T_S^k.x$, we consider two cases. If $T_S^k.z = 0$, then no subscriptions are represented by S at interface I^k. This case is also handled by Algorithm 3. Indeed, no subscriptions are affected by the fact that there are no more subscriptions interested in S downstream interface I^k.

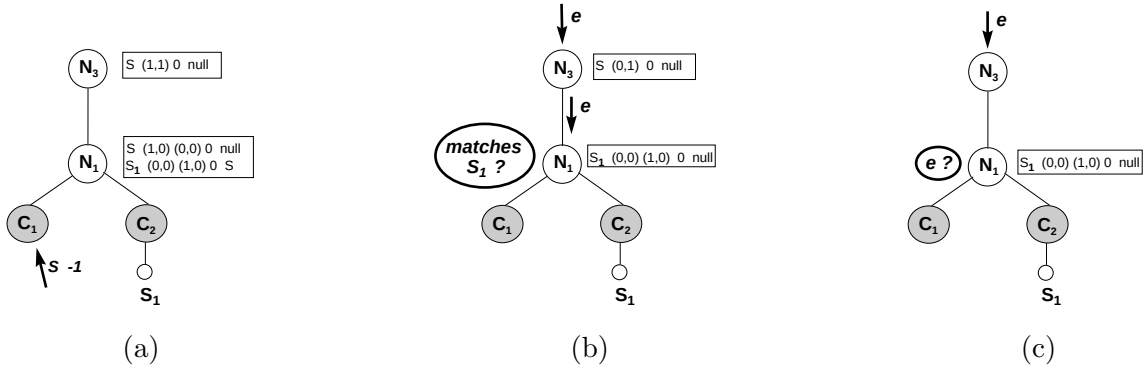


Figure 4.8: (a) Subscription S is being totally cancelled at nodes N_1 and N_3. At node N_3, subscription S_1 is represented by S_1 (b) There are no more consumers interested in events matching S at node N_3. N_3 knows that one subscription is represented by S, but it does not know if event e matches them. Consequently, e may be forwarded wrongly to node N_1 (c) If $entry(S)$ is removed, e is not forwarded to node N_1, even if it matches S_1.

Now consider the case where $|n_S| = T_S^k.x$ and $T_S^k.z > 0$, i.e., some subscriptions are represented by S at interface I^k, as illustrated in Figure 4.8(a). Now consider an event e that arrives at node N and that matches S.

If $\overline{T_S^k} > 0$, e will be forwarded downstream interface I^k. However, there are no more consumers interested in events matching S downstream I^k, and node N cannot determine whether e matches any of the subscriptions represented by S at I^k, as illustrated in Figure 4.8(b).

Now if $\overline{T_S^k} = 0$, then $entry(S)$ is removed form the routing table and e may not be forwarded downstream, as illustrated in Figure 4.8(c).

Consequently, in both cases, we may have imperfect routing, and in the worst case routing may degenerate into *flooding.* Therefore, we need to implement an additional routing table update operation to deal with this problem. This operation, which we refer to as the cancellation algorithm, is implemented in Algorithms 8, 7 and 6 and detailed in the rest of this section.

Overview of the cancellation algorithm. As previously mentioned, the cancellation algorithm handles the case where $|n_S| = T_S^k.x$ and $T_S^k.z > 0$, i.e., subscription S is being totally cancelled at interface I^k (or at all interfaces), and some subscriptions are represented by S at interface I^k. The key concept used by the cancellation algorithm to maintain perfect routing in this case is the *substitution relation.*

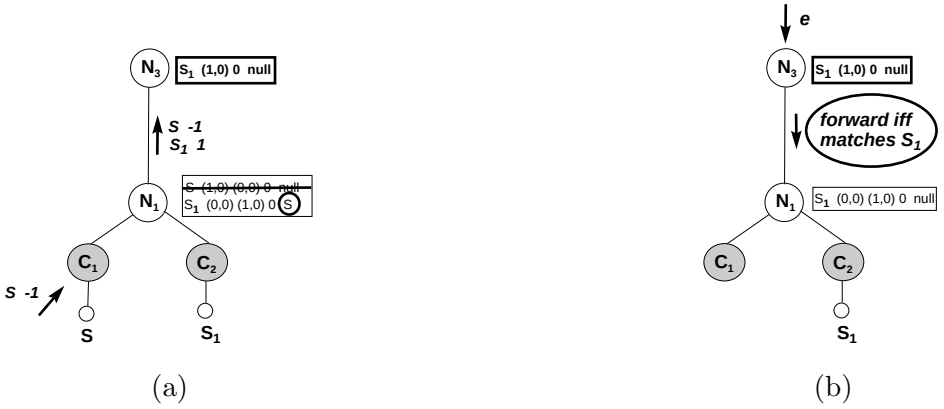


Figure 4.9: (a) At node N_1, subscription S_1 is substituted by S. Hence, at the upstream node, it is represented by it. When subscription S is being totally cancelled at node N_1, subscription S_1 can be reinserted in the routing table of node N_3 via the outgoing advertisement. (b) At node N_3, the subscription that was represented by S has been reinserted in the routing table and perfect routing maintained.

Consider node N where subscription S is being totally cancelled at interface I^k, and such that there are some subscriptions represented by S at I^k. We have explained in the previous section that routing inaccuracy may occur because node N does not know if an incoming event matches the subscriptions represented by S. The problem would be solved if those subscriptions were reinserted in node N's routing table. This is possible thanks to the substitution relation. Indeed, we have seen in Section 4.2.4 that if a subscription S_1 is represented by subscription S at interface I^k, then there is a downstream node where S_1 is substituted by S. Hence, that node can inform node N that subscription S_1 is the one that is represented by S at interface I^k (property 3). Subsequently, subscription S_1 can be properly "reinserted" in the routing table, and perfect routing maintained, as illustrated in Figure 4.9.

More formally, let $\{S_i\}$ denote the set of subscriptions that S represents at interface I^k. According to the corollary of Property 5, those subscriptions are the subscriptions of $tree(S)$ at node N_{down} (i.e., directly or indirectly substituted by S) plus those that are represented by any of them at any interface (also at node N_{down}). To prevent imperfect routing, each subscription S_i must be properly "reinserted" in the routing table of node N. This especially includes the possible substitution relationships between the subscriptions (which must always be performed).

The cancellation algorithm consists of a set of algorithms, implemented in pseudo-code in Algorithms 8, 7 and 6. It works by recursively reinserting subscriptions in the routing tables of the encountered routers so as to maintain perfect routing (as previously mentioned). It starts at the consumer node where the total cancellation of subscription S was originally issued. Indeed, if subscription S is being totally cancelled at a given node N in the system, then there is a unique consumer node downstream node N, where the last consumer interested in subscription S *totally cancelled* it. Since there are no representation relations at consumer nodes, we need a dedicated algorithm, which is given in Algorithm 6. At each inner node encountered upstream, the cancellation algorithm then proceeds as indicated in Algorithm 8. It ends when a router is encountered where $entry(S)$ is such that: $|n_S| \neq T_S^k.x$ *or* $T_S^k.z = 0$ (if there is no such router, then the algorithm proceeds until the producer node), where the cancellation of subscription S further proceeds with Algorithm 3. Algorithm 7 is a key procedure that is used by both Algorithms 8 and 6.

We now detail the operation of the cancellation algorithm.

Cancellation algorithm at consumer nodes. Consider consumer node C, where the last consumer interested in subscription S totally cancelled it. The cancellation of subscription S is handled with Algorithm 6.

Since we are at a consumer node, the total cancellation of subscription S necessarily implies the removal of its entry from the routing table (there is only one interface, see Section 4.2.5). Then, we

Algorithm 6 — Cancellation algorithm - consumer nodes

1: **if** $P_S \neq null$ **then**
2: **for all** S_k ancestor of S in $tree(h(S))$ **do**
3: $R_{S_k} \leftarrow R_{S_k} + n_S$
4: **end for**
5: $adv_{out} \leftarrow (h(S); 0; n_S)$
6: **for all** S_k such that $P_{S_k} = S$ **do**
7: $P_{S_k} = P_S$
8: **end for**
9: delete $entry(S)$
10: **else**
11: call $reinsert()$ (Algorithm 7)
12: **end if**
13: $adv_{out} \leftarrow (S; n_S; 0)$ [+ appended triples]
14: Send adv_{out} upstream

Algorithm 7 — Reinsert function: $reinsert()$

1: **for all** S_j such that $P_{S_j} = S$ **do**
2: push S_j in $L_{reinsert}$
3: $P_{S_j} \leftarrow null$
4: **end for**
5: delete $entry(S)$
6: **for all** $S_j \in L_{reinsert}$ **do**
7: **if** $\exists S', S' \supset S_j, P_{S'} = null$ **then**
8: substitute S_j by S'
9: **end if**
10: **end for**
11: **for all** $S_j \in L_{reinsert}$ **do**
12: **if** $P_{S_j} = null$ **then**
13: $count_{S_j} = \sum_k T^k_{S_j}.x$
14: append $(S_j; count_{S_j}; R_{S_j})$ to adv_{out}
15: **else if** $P_{S_j} \notin L_{reinsert}$ **then**
16: $count_{S_j} = \sum_k T^k_{S_j}.x$
17: append $(P_{S_j}; 0; count_{S_j} + R_{S_j})$ to adv_{out}
18: **end if**
19: **end for**

consider two cases according to the value of P_S:

If $P_S \neq null$ Then S has been substituted by another subscription. Then, we update the R field of the subscriptions ancestor of S to take into account the fact that they are no longer a substitute for $|n_S|$ instances of subscription S (lines $2-4$). Then, the subscriptions substituted by S can now be substituted by P_S (up one level in $tree(h(S))$ (lines $6-8$). Subsequently, they have been properly reinserted in the routing table. Since subscription S is substituted by $h(S)$, the outgoing advertisement advertises $h(S)$, and mentions that $h(S)$ is to represent $|n_S|$ fewer instances of subscription S (Property 3) (lines $2-5$). At the upstream node N_{up}, that advertisement is handled with Algorithm 3. We finally delete $entry(S)$ (line 9).

If $P_S = null$ Then, S has entry in the routing table of node N_{up} and will be totally cancelled at the incoming interface. Also, some subscriptions are represented by S at that interface. At node N_{up}, we are then also in the case of application of the cancellation algorithm. We have seen that to prevent imperfect routing, node N_{up} needs to know all the subscriptions that S represents at interface I^k. Because there are no representations at a consumer node, those subscriptions are the ones at node C that are substituted by S (plus the ones that are substituted by each of them). Those subscriptions must now be reinserted in the routing table. For that purpose, we proceed as indicated in Algorithm 7.

We build a list $L_{reinsert}$ that contains all the subscriptions that are substituted by S, and we subsequently reset their P field (as if they were no longer substituted), and we delete $entry(S)$ (lines $1-5$). We then establish the possible substitution relations between the subscriptions in $L_{reinsert}$ and the other subscriptions (lines $6-10$). Once a given subscription $S_j \in L_{reinsert}$ has been substituted, there are three possible cases:

$P_{S_j} = null$: S_j could not be substituted by any subscription. Then, for the purpose of reinserting subscription S_j in the routing table of node N_{up}, we append the triple $(S_j; count_{S_j}; R_{S_j})$ to the outgoing advertisement, as if S_j were a newly registered subscription, with $count_{S_j}$ instances (lines $13-14$). Also, S_j is to represent R_{S_j} subscriptions at interface I^k of node N_{up} (see Property 3) Besides, because S_j was formerly substituted by S, it is currently represented by S at node N_{up} (Property 3). Thus, at node N_{up}, $\overline{T^k_{S_j}} = 0$ or S_j does not have an entry, and we have $S_j \subset S$ (since S_j was previously substituted by S).

$P_{S_j} \neq null$ and $P_{S_j} \notin L_{reinsert}$: S_j has been substituted by a subscription that does not belong to $L_{reinsert}$. Then, to reinsert all the instances of S_j plus the instances of each subscription that is substituted by it (no representations at consumer nodes), we append a triple to the outgoing advertisement, that advertises P_{S_j}, and indicating that this latter represent $count_{S_j} + R_{S_j}$ additional instances of subscriptions (instances of S_j plus instances of subscriptions substituted by S_j) (lines $16-17$). Also, because P_{S_j} is not one of the subscriptions in $L_{reinsert}$, it was not substituted by S before. Then, because $P_{P_{S_j}}$ is null (a subscription is always substituted by a subscription that is not already substituted), P_{S_j} has an entry at node N_{up} such that $\overline{T^k_{P_{S_j}}} \neq 0$.

$P_{S_j} \neq null$ and $P_{S_j} \in L_{reinsert}$: This case does not have to be dealt with. Indeed, the changes made to S_j are taken into account in the first case, via the R field of subscription $P_{P_{S_j}}$, during the substitution operation.

Note that it is not possible to have both $P_{S_j} \notin L_{reinsert}$ and $P_{S_j} \in tree(S)$. Indeed, substitutions are always performed at the top of a substitution tree. If S_j has been substituted by a subscription S' that was previously in $tree(S)$, then it is necessarily one of the subscriptions in $L_{reinsert}$.

Finally, we create the advertisement for the cancellation of subscription S, along with the appended triples, and send it to the upstream interface (lines $13-14$ in Algorithm 6). Also, for a given appended triple $(S_j; count_{S_j}; R_{S_j})$, we have seen that at node N_{up} we either have $\left(\overline{T^k_{S_j}} = 0 \,\text{and}\, S_j \subseteq S\right)$ or $\overline{T^k_{P_{S_j}}} \neq 0$. Then, we have the following property: If $\overline{T^k_{S_j}} = 0$, then $S_j \subseteq S$.

Subsequently, all the subscriptions that were substituted by S have been reinserted in the routing table and we had $\sum_{S_j} count_{S_j} + R_{S_j} = R_S$ before $entry(S)$ was deleted. Because of Property 5, the

former sum is also equal to $T_S^k.z$ in the routing table of node N_{up}. Those subscriptions were appended to the outgoing advertisement. To prevent imperfect routing at node N_{up}, all $T_S^k.z$ instances must be "reinserted" in its routing table.

The cancellation algorithm then proceeds at node N_{up} and its upstream nodes as indicated in Algorithm 8.

Cancellation algorithm at inner nodes. We now detail the cancellation algorithm in the case of an inner node. The operation is given in Algorithm 8 and explained as follows.

Consider node N receiving the advertisement for the cancellation of subscription S, $adv_{in}(S)$, from the node downstream interface I^k, N_{down}, and such that $T_S^k.x = |n_S|$ (all instances of S are to be cancelled at interface I^k) and $T_S^k.z > 0$. Let $\{L\}$ be the set of the subscriptions appended to $adv_{in}(S)$ (as part of the optional triples appended to an advertisement). Each triple comes in the form $(S_j; n_{S_j}; r_{S_j})$. We suppose (recursive assumption) that the instances of the subscriptions in $\{L\}$ represent the instances of the subscriptions that are represented by S at interface I^k of node N. Also, we suppose that for each subscription S_j in $\{L\}$, we have the following property: If $\overline{T_{S_j}^k} = 0$, then $S_j \subseteq S$. Also we suppose that those subscriptions are such that at node N_{down}, their P field is null.

Consider a given subscription $S_j \in \{L\}$. S_j cannot be represented at interface I^k, by a subscription other than S. Indeed, suppose that S_j can be represented by S' at I^k. Then S' has an entry at node N_{down} and $S_j \subseteq S'$. Then S_j would have been substituted by S' at node N_{down}. This contradicts the recursive assumption. In addition, no subscriptions can be represented by S_j at interface I^k. Indeed, suppose that S'' can be represented by S_j at interface I^k. If $\overline{T_{S_j}^k} \neq 0$, then the representation would have been done already. Now if $\overline{T_{S_j}^k} = 0$, then because of the recursive assumption, $S_j \subseteq S$ and S'' would have been represented by S already (S is being cancelled downstream interface I^k, hence $\overline{T_S^k} \neq 0$). Consequently, the only possible relations between the subscriptions in $\{L\}$ are substitution relations. Also, note that because of the recursive assumption, there are no possible relations between two subscriptions in $\{L\}$. If that were the case, one would be contained by the other and would have been substituted downstream, which contradicts the recursive assumption.

To cancel subscription S, we must first proceed with the "reinsertions" of each subscription S_j in $\{L\}$ that were advertised in the additional triples $(S_j; n_{S_j}; r_{S_j})$.

If S_j does not have an entry, we create a null entry for it, and make the P field point to S (lines $8-11$). This is possible because of the recursive assumption (we have $S_j \subseteq S$ if $\overline{T_{S_j}^k} = 0$). Then for every subscription in $\{L\}$, we store those that belong to $tree(h(S))$ in a list, L_{keep}, and remove them from L (lines $12-15$).

For every subscription S_j in L_{keep}, we update its $T_{S_j}^k$ field (S_j is necessarily substituted by another subscription) and its R field to take into account the n_{S_j} additional instances of S_j and the r_{S_j} additional instances of subscriptions that it must represent (lines $19-22$). Besides, all those additional instances of subscriptions now also have for substitutes every subscription ancestor of S_j in $tree(h(S_j))$. We then update their R field accordingly (lines $23-25$). *keep* is a counter that represents all the instances of each subscription S_j in L_{keep} plus the instances of additional subscriptions that S_j represents (line 19).

The subscriptions that remain in $\{L\}$ are the ones that already have an entry and that do not belong to $tree(h(S))$. We have seen that there are no possible representation relations between the subscriptions in $\{L\}$ and the others. In addition, each subscription S_j already has an entry. Thus there are no possible substitution relations with S_j. Since $S_j \notin tree(h(S))$, the changes induced by the triple $(S_j; n_{S_j}; r_{S_j})$ and those induced by the cancellation of subscription S are independent from each other. Triple $(S_j; n_{S_j}; r_{S_j})$ can be completely dealt with by Algorithm 3 (lines $24-26$). The changes induced are propagated upwards via the unique triple $(h(S_j); 0; n_{S_j} + r_{S_j})$ appended to the outgoing advertisement. The subscription advertised, $h(S_j)$ is not substituted. Thus, at the upstream node N_{up}, incoming interface p, it is such that $\overline{T_{h(S_j)}^p} \neq 0$. However, $n_{S_j} + r_{S_j}$ still

Algorithm 8 — Cancellation algorithm - inner nodes

declare L_{keep}
declare L
declare $L_{reinsert}$
for all $S_j \in adv_{in}(S)$ **do**
  push S_j in L
end for
for all $S_j \in L$ **do**
  if S_j does not have an entry **then**
    create a null entry for S_j, $entry(S_j)$
    $P_{S_j} \leftarrow S$
  end if
  if $S_j \in tree(h(S))$ **then**
    push S_j in L_{keep}
    remove S_j from L
  end if
end for
declare $keep = 0$
for all $S_j \in L_{keep}$ **do**
  $keep \leftarrow keep + n_{S_j} + r_{S_j}$
  $T^k_{S_j}.x \leftarrow T^k_{S_j}.x + n_{S_j}$
  $T^k_{S_j}.z \leftarrow T^k_{S_j}.z + r_{S_j}$
  $R_{S_j} \leftarrow R_{S_j} + r_{S_j}$
  for all S_k ancestor of S_j in $tree(h(S_j))$ **do**
    $R_{S_k} \leftarrow R_{S_k} + n_{S_j} + r_{S_j}$
  end for
end for
for all $S_j \in L$ **do**
  call algorithm 3: "Routing Table Update"
end for
for all S_k ancestor of S in $tree(h(S))$ **do**
  $R_{S_k} \leftarrow R_{S_k} + n_S - T^k_S.z$
end for
if $P_S \neq null$ **then**
  $adv_{out} \leftarrow (h(S); 0; n_S - T^k_S.z + keep)$
  if $\forall p, \overline{T^p_S} = 0$ **then**
    for all S_k such that $P_{S_k} = S$ **do**
      $P_{S_k} = P_S$
    end for
    delete $entry(S)$
  else
    $R_S \leftarrow R_S - T^k_S.z$
    $T^k_S \leftarrow (0, 0)$
  end if
else
  if $\forall p, \overline{T^p_S} = 0$ **then**
    call $reinsert()$ (see Algorithm 7)
    $adv_{out} \leftarrow (S; n_S; 0)$ [+ appended triples]
  else
    $adv_{out} \leftarrow (S; n_S; -T^k_S.z + keep)$
    $R_S \leftarrow R_S - T^k_S.z$
    $T^k_S \leftarrow (0, 0)$
  end if
end if
Send adv_{out} upstream

represents the instances of subscription S_j that were previously represented by S at interface I^k.

Now, we update the entries of the ancestors of S, if any, in a similar way to that in algorithm 4. $|n_S|$ instances of subscription S have been canceled. Thus those are no longer substituted by the ancestors of S. Also, $T_S^k.z$ subscriptions are no longer represented by S at interface I^k, they are no longer substituted by its ancestors. Thus we decrement the R field of every ancestor of S by $|n_S| + T_S^k.z$ (lines $27 - 29$) and decrement that of S by $T_S^k.z$, if its entry does not have to be deleted (lines 43 and 53).

To further proceed with the cancellation of subscription S, we consider different cases depending on the value of P_S.

1. *Case 1: P_S is not null*

   Then, if $\exists p \neq k$ such that $\overline{T_S^p} \neq 0$, $entry(S)$ does not have to be deleted. As previously mentioned, we update R_S to account for the $T_S^k.z$ fewer instances of subscriptions that S represent (line 41), and we reset T_S^k (lines 42).

   Now if $\forall p, \overline{T_S^p} = 0$, then $entry(S)$ has to be deleted. But before doing this, we must take into account the possible subscriptions that are substituted by S. Because P_S is not null, we can just make the P of those subscriptions point to P_S. Then we delete $entry(S)$ (lines $36 - 39$).

   Subscription $h(S)$ has seen its R field decremented by $|n_S| + T_S^k.z$ and incremented by $keep$. $T_S^k.z$ represents all the instances of the subscriptions that were represented by S at interface I^k, and also substituted by $h(S)$ (see Property 4). $keep$ represents the instances of the subscriptions that were represented by S at interface I_k, and that are still substituted by $h(S)$ (since they were reinserted in $tree(h(S))$). Hence, $T_S^k.z - keep$ represents the instances of the subscriptions that are no longer in $tree(h(S))$. Consequently, the outgoing advertisement advertises $h(S)$, and mentions that $h(S)$ is to represent $n_S - T_S^k.z + keep$ fewer instances of subscriptions (line 34). At the upstream node N_{up}, that advertisement is handled with Algorithm 3.

2. *Case 2: P_S is null*

   We consider the two following subcases:

   First case: $\exists p \neq k$ such that $\overline{T_S^p} \neq 0$. Then $entry(S)$ is not deleted. This implies that at node N_{up} we are no longer in the case where the cancellation algorithm must be applied. This case is similar to the previous one (where $P_S \neq null$), except that subscription S is advertised (instead of $h(S)$). The advertisement indicates that there are n_S fewer instances of subscription S, and that this latter is to represent $-T_S^k.z + keep$ fewer subscriptions (line 49). We then update $entry(S)$ (line $50 - 51$).

   Second case: $\forall p, \overline{T_S^p} = 0$ Then, $entry(S)$ is to be deleted, and at node N_{up}, we will be in the same situation as node N, where the cancellation algorithm must be applied. This implies that node N_{up} must know the subscriptions that S represents at the incoming interface. But because $\forall p, \overline{T_S^p} = 0$, the only subscriptions that S represented at node N were the ones at interface I_k. Those were the subscriptions in $\{L\}$ and they have been reinserted in the routing table of node N either by Algorithm 3 or as described in Algorithm 8 and the changes to propagate upstream were appended to the outgoing advertisement. Now the only subscriptions that S represents at the incoming interface of node N_{up} are the ones that are substituted by S at node N. We have seen that there are no possible representation relations between the subscriptions in $\{L_{keep}\}$ and the others. Hence, we are in a situation similar to the one that we would have if N were a consumer node. We can then call the *reinsert* function in Algorithm 7 to reinsert all the subscriptions that are directly substituted by S and append the result to the outgoing advertisement (lines $47 - 48$). As in the case of a consumer node, the triples appended satisfy the recursive assumption. The other triples appended to the outgoing advertisements were generated by Algorithm 3. The subscriptions advertised also satisfy the recursive assumption. Finally, all the triples appended to the outgoing advertisement satisfy the recursive assumption.

In particular, they account for the subscriptions that are represented by S at the incoming interface of node N_{up}. Then at node N_{up}, the recursive assumption is true and Algorithm 8 is called.

Finally, the outgoing advertisement is sent to the upstream interface (line 54).

Recursion termination. The process goes on recursively, until the root node or a node N_{final} is reached, where $entry(S)$ does not have to be deleted and the cancellation algorithm does not have to be called. At each encountered router N, the subscriptions that were represented by S have been reinserted in the routing table and perfect routing maintained.

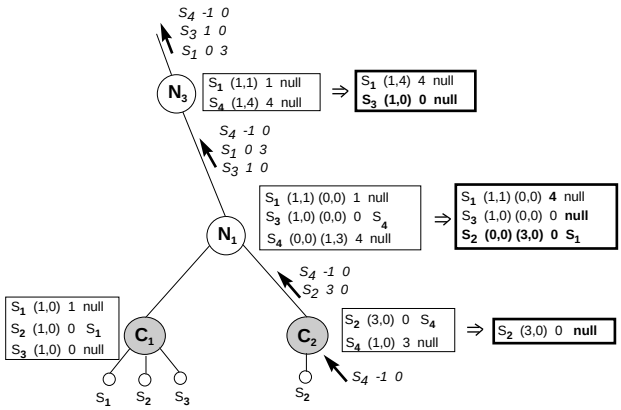


Figure 4.10: Example of the cancellation algorithm, where the initial state is equal to the final state of Figure 4.7.

Example 3. *Figure 4.10 illustrates the operation of the cancellation algorithm on the pub/sub network of Figure 4.1(a). Four consumers have already registered some subscriptions. The consumer at client node C_0 cancels all instances of subscription S_0 that it previously registered. This results in updates of the routing table at each node on the path from C_2 to each publisher. For the sake of clarity, we have only represented inner nodes N_1 and N_3. Also, we recall that: $S_0 \supseteq S_2$, $S_1 \supseteq S_2$, and $S_4 \supseteq S_3$, but there are no relationships between S_4 and S_1, and between S_2 and S_3.*

At node C_2, subscription S_2 is reinserted in the routing table. This results in triple $(S_2; 3; 0)$ being appended to the outgoing advertisement.

At node N_1, S_2 is first reinserted in the routing table. It is subsequently substituted by subscription S_1. This results in triple $(S_1; 0; 3)$ appended to the outgoing advertisement. Then, all the subscriptions that are substituted by S_0 are to be reinserted, only S_3 is concerned here . It could not be substituted, triple $(S_3; 1; 0)$ is appended to the outgoing advertisement.

At node N_3, triple $(S_1; 0; 3)$ is handled by the registration algorithm, which results in the same triple appended to the outgoing advertisement. Then, S_3 is reinserted in the routing table which results in triple $(S_3; 1; 0)$ being appended. As no subscriptions are substituted by S_0, the process stops here and $entry(S_0)$ is deleted.

Finally, all the routing tables have been updated. At each routing table, the subscriptions that were represented by S_0 have been reinserted in the routing table to preserve perfect routing.

Dynamics of cancellations

Consider an advertisement for the cancellation of subscription S at node N, arriving through interface I^k. It results from Section 4.2.5 that we can distinguish between two different cases according to the value of $entry(S)$ and the type of routing table update that must be applied.

- If S is *totally cancelled* at interface I^k *and* $T^k.z \neq 0$: then Algorithm 8 is called to update the routing table.
  - If S is *totally cancelled at node N:*
    - If S is substituted: we do not have to compute containment relationships. The overall routing table update is a fast operation.
    - If S is not substituted: we have to compute a potentially high number of containment relationships. The overall routing table update is significantly costlier.
  - If S is not *totally cancelled at node N:* then the overall routing table update is a fast operation.
- If S is *partially cancelled* at node N: then Algorithm 3 is called to update the routing table update, as in the case of a *duplicate registration.* The operation is very fast.

We now identify the implications between the types of routing table updates performed at node N and the ones that will be performed at the upstream node N_{up}. Let Adv_{out} be the advertisement sent to N_{up} as the result of the routing table update at node N.

- If S is *totally cancelled* at interface I^k *and* $T^k.z \neq 0$:
  - If S is *totally cancelled at node N:*
    - If S is substituted: Algorithm 3 is called at node N_{up}. Adv_{out} is seen as a *duplicate registration.*
    - If S is not substituted: Algorithm 8 is called at node N_{up}. The routing table update is significantly costlier.
  - If S is not *totally cancelled at node N:* Algorithm 3 is called at node N_{up}. Adv_{out} is seen as a *duplicate registration.*
- If S is *partially cancelled* at node N: Algorithm 3 is called at node N_{up}. Adv_{out} is seen as a *duplicate registration.*

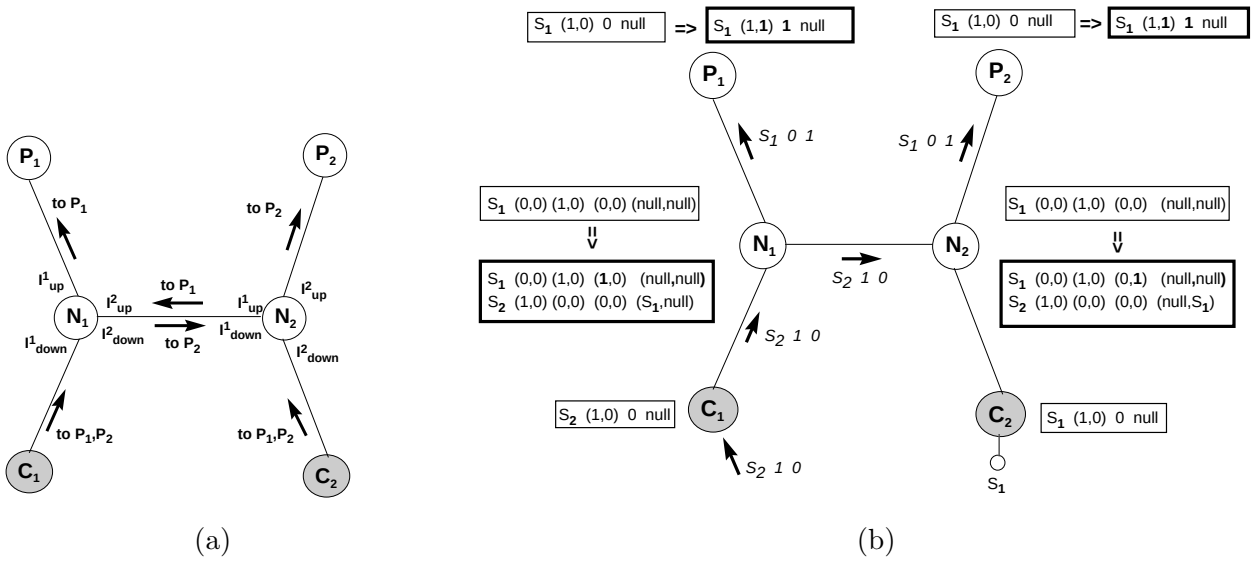


Figure 4.11: (a) A sample network topology with two consumer nodes and two producer nodes. The best paths that lead to each producer are indicated with arrows. Each node's interfaces are indicated next to them. Although simple, this topology encompasses all the cases that can be encountered with respect to a network with multiple producers (b) Registration of subscription S_2 at node C_1.

4.2.6 Extension to the case of multiple producers

Overview

In this section, we explain how the case of multiple producers is handled. Generally speaking, the case of multiple producers is similar to the case of a single one, except that for a given router N, there may be several *upstream* interfaces, each one leading to one or more producers. It is always possible for each router to maintain a separate routing table for each producer. If the producers publish events with different formats, in our case XML documents with different DTDs, there is no other solution than maintaining a different routing table for each producer. Indeed, subscription aggregation does not apply between subscriptions that correspond to different DTDs. Consequently, in this section, we will focus on the case where there are multiple producers that publish events of the same type (DTD). Then, we do not have to maintain a separate routing table. We can use a single routing table, with only minor modifications to the format of its entries, and to the RTU algorithm.

Formats

Consider a router N, with n total interfaces. Let p be the number of upstream interfaces, that is, an interface that leads to one or more producers. An entry for subscription S, $entry(S)$, in the routing table of node N has the following format:

$$S\ ;\ (T_S^1, \cdots, T_S^n)\ ;\ (R_S^1, \cdots, R_S^p)\ ;\ (P_S^1, \cdots P_S^p)$$

In other words, the format is the same as in the case of a single producer, except that there are separate R and P field for each upstream interface.

The format of advertisements is unchanged.

Key concepts

The case of multiple producers is very similar to the case of a single one. In fact, consider an advertisement for a subscription S that arrives at node N. For node N, there are p upstream interfaces that lead to different producers. This corresponds to p different spanning trees, each rooted at a producer node. Node N can process the advertisement as in the case of the previous section, when considering each upstream interface, one at a time. For each of them, I_{up}^g, node N does not consider in the routing table the T^g field of the entries, and only considers the R^g and P^g fields. The routing table update is almost similar to the case of a single producer, at the differences that we will explain shortly. When this has been done, it sends the corresponding advertisement to interface I_{up}^g.

RTU protocol

The main changes concern the substitution trees. In the previous case of a single producer, we had one substitution tree for a given subscription S. Now, considering router N with p upstream interfaces, we have p substitution trees for each subscription. We will refer to $tree^g(S)$ as the substitution tree that corresponds to upstream interface I_{up}^g. Also, $h^g(S)$ denotes the root of substitution tree $tree^g(S)$. Recall that a substitution tree is derived only from the R fields of the subscriptions' entries (and not a specialized structure).

Consider node N receiving an advertisement for subscription S. The process of updating the routing table proceeds as in the case of a unique producer. The establishment of possible subscriptions relations and the modification of existing ones proceed as explained in Section 4.2.5 , except that each different upstream interface I_{up}^g must be considered independently. In other words, node N considers each spanning tree independently of the others. A different outgoing advertisement is sent to each upstream interface I_{up}^g. We will refer to it as adv_{out}^g.

Thus, when considering interface I^g_{up}, for each subscription S_j, only $tree^g(S_j)$ must be considered. Also, all the T^g fields of the subscriptions' entries must be ignored (since we consider the spanning tree rooted upstream interface I^g_{up}). Each change that was made to a substitution tree $tree^g(S_j)$ yields to one or more triples appended to advertisement adv^g_{out}.

The RTU algorithms are very similar to those introduced in the case of a single producer. Nevertheless, for completeness, we included their extension to multiple producers in Appendix A, in Algorithms 28, 30, 27, 31, 29. The cancellation algorithm in the case of consumer nodes is unchanged (as in Algorithm 6).

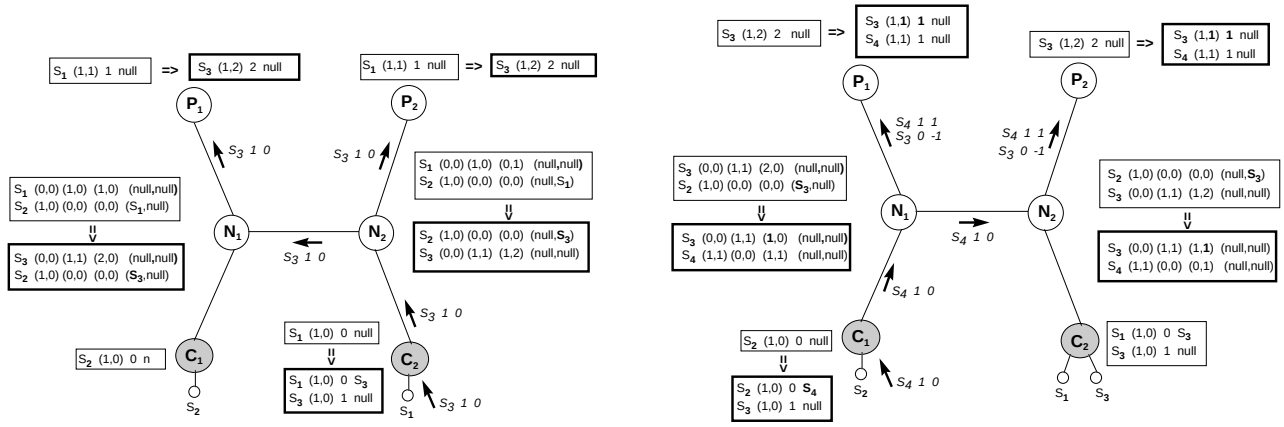


Figure 4.12: (a) Registration of subscription S_0 at node C_2 (b) Registration of subscription S_3 at node C_1

Examples

In this section, we illustrate the operation of the RTU algorithm when applied to the network of Figure 4.11(a). We have two consumer nodes C_1 and C_2, 4 inner nodes N_1, N_2, P_1 and P_2. P_1 and P_2 are producer nodes. Next to a node's link, we indicated the interface that connects it to its neighbor. For example, for node N_1, I^1_{down} (I^2_{down}) is the downstream interface from $C_1(N_2)$. I^1_{up} (I^2_{up}) is the upstream interface to producer node P_1 (P_2). Note that I^2_{down} and I^2_{up} are physically the same interface, but are different interfaces in the routing table of node N_1.

Subscription S_1 was already registered at consumer node C_2. We then consider 3 subscriptions S_2, S_3 and S_4 such that: $S_2 \subseteq S_1 \subseteq S_3$ and $S_1 \subseteq S_4$. There are no containment relationships between S_3 and S_4. We then apply the following scenario sequentially:

- S_2 is registered at node C_1. The routing table updates are illustrated in Figure 4.11(b)
- S_3 is registered at node C_2 (Figure 4.12(a)).
- S_4 is registered at node C_1(Figure 4.12(b)).
- S_3 is totally cancelled at node C_2(Figure 4.13(a)).
- Finally, S_4 is totally cancelled at node C_1(Figure 4.13(b)).

The scenarios are illustrated in the associated figure and commented below. In each figure, registered subscriptions are represented below their corresponding client nodes. Routing tables (shown next to the nodes) are updated as a result of the registration of a subscription (updated tables are shown with a thick frame).

When S_2 is registered at node C_1, at node N_1, S_2 is substituted by S_1 at interface I^1_{up}. It is not substituted at interface I^2_{up}. Indeed, when considering interface I^2_{up}, node N_1 ignores the T^2 fields of the subscriptions entries. Consequently, $entry(S_1)$ does not appear. From the point of view of the

spanning tree rooted at P_2, there are no consumers downstream N_1 interested in S_1. At node N_2, S_2 is substituted by S_1 at interface I^2_{up}. At nodes P_1 and P_2, S_2 is represented by S_1 at the incoming interface.

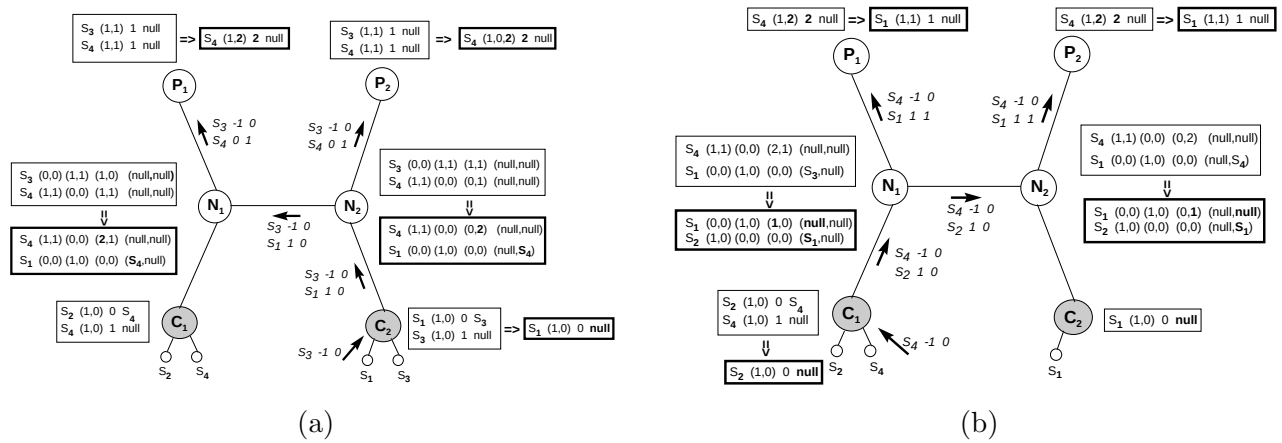


Figure 4.13: (a) Subscription S_0 is being totally cancelled at node C_2. To maintain perfect routing, subscription S_1 is reinserted in the routing table of nodes N_2 and N_1 (b) Subscription S_3 is being totally cancelled at node N_1. To maintain perfect routing, subscription S_2 is reinserted at nodes N_1 and N_2, and subscription S_1 is reinserted at nodes P_1 and P_2. The same system's state as in Figure 4.11(b) is reached.

When S_3 is registered at node C_2, S_1 is substituted by S_3. (recall that subscriptions are not represented at consumer nodes). At node N_2, S_1 is subsequently represented by S_3 at interface I^2_{down}. When considering interface I^2_{up}, S_2 is then substituted by S_3 (since $entry(S_1)$ is removed). However, S_2 is not substituted to S_3 when considering I^1_{up}. At nodes P_2, N_1 and P_1, S_1 is represented by S_3 at the incoming interface. At node N_1, S_2 is also substituted by S_3 at interface I^1_{up}.

When S_4 is registered at node C_1, S_2 is substituted by S_4. At nodes N_1 and N_2, S_2 is represented by S_4, at the incoming interface. At node N_1, when considering interface I^1_{up}, S_2 was previously substituted by S_3. Consequently, the outgoing advertisement specifies that S_3 should represent one fewer subscription and S_4 an additional one. The same procedure applies for node N_2. However, at node N_1, when considering interface I^2_{up}, S_2 was not substituted. Thus, $tree^2(S_3)$ was not modified, and the outgoing advertisement does not contain any additional triples.

Now consider the case where subscription S_3 is totally cancelled at node C_2. Since S_1 was substituted by S_3, it must be reinserted at node N_2. Subsequently, when considering interface I^2_{up}, it is substituted by S_4. This yields to triple $(S_4\,;\,0\,;\,1)$ appended to the outgoing advertisement to P_2. When considering interface I^1_{up}, S_1 is not substituted. S_1 is then directly advertised via a triple appended to the outgoing advertisement to N_1, so as to be reinserted. The same procedure applies at node N_1.

We finally consider the case where subscription S_4 is totally cancelled at node C_1. At node N_1, subscription S_2 is reinserted in the routing table and S_4 removed. Subsequently, when considering interface I^1_{up}, S_2 is substituted by S_1. Since the latter was substituted by S_4, it has to be reinserted at node P_1. Also, it should represent one additional subscription (since S_2 has been substituted to it). This results in triple $(S_1\,;\,1\,;\,1)$ appended to the outgoing advertisement to P_1. Now when considering interface I^2_{up}, S_2 and S_1 are not substituted. Hence, S_2 is directly advertised in the outgoing advertisement for N_2. At node N_2, the process is similar to that of node N_1, when interface I^1_{up} is considered. Finally, the subscriptions have been reinserted as necessary. Unsurprisingly, the system's state is then exactly the same as in Figure 4.11(b).

Algorithms

The definitions of the representation and substitution operations are modified as indicated in Definitions 3 and 4.

Definition 3 (Representation). *Consider entries for subscriptions S_1 and S_2 at non-consumer node N such that $S_1 \supset S_2$, $\overline{T^k_{S_1}} > 0$ and $\overline{T^k_{S_2}} > 0$, then S_2 must be* represented by *S_1 at interface I^k. This operation consists in modifying their entries as follows:*

1. $T^k_{S_1}.z \leftarrow T^k_{S_1}.z + \overline{T^k_{S_2}}$
2. $R^g_{S_1} \leftarrow R^g_{S_1} + \overline{T^k_{S_2}}$ *for all upstream interface* $I^g_{up} \neq I^k$
3. $R^g_{S_2} \leftarrow R^g_{S_2} - T^k_{S_2}.z$ *for all upstream interface* $I^g_{up} \neq I^k$
4. $\overline{T^k_{S_2}} \leftarrow 0$

Thereafter, we say that S_2 is represented by S_1 at interface I^k.

Definition 4 (Substitution). *Consider entries for subscriptions S_1 and S_2 at node N such that: $S_1 \supset S_2$, $P^g_{S_1} = null$, and $P^g_{S_2} = null$. Then S_2 must be* substituted by *S_1 at upstream interface I^g_{up}. This operation consists in modifying their entries as follows:*

1. $P^g_{S_2} \leftarrow S_1$
2. $R^g_{S_1} \leftarrow R^g_{S_1} + \sum_{k \leq n} T^k_{S_2}.x + R^g_{S_2}$

Thereafter, we say that S_2 has been substituted by S_1 at upstream interface I^g_{up}, and S_2 must subsequently be advertised by S_1 at the node upstream that interface, i.e., any incoming advertisement $(S_2; n; r)$ yields an outgoing advertisement $(S_1; 0; n + r)$ towards interface I^g_{up}. Note that a subscription may be substituted by only one other subscription at a given upstream interface (but may be substituted by a subscription at one upstream interface and by another subscription at another upstream interface).

The RTU algorithm extended to the case of multiple producers is described in appendix in Algorithms 28, 30, 27, 31, 29 (the cancellation algorithm in the case of consumer nodes is identical to Algorithm 6). I^k is the incoming interface, i.e., where the advertisement comes from. Also, in the algorithms, each time a specific upstream interface I^g_{up} is considered, then all the T^k fields of the subscriptions entries are ignored, where $I^g_{up} = I^k$. If there is no downstream interface I^k that is equal to I^g_{up}, then all fields are considered.

Chapter 5

Efficient subscription management with XSearch

5.1 Motivations

Efficient subscription management is critical for the overall performance of the system and to guarantee short registration delays to consumers.

When a consumer registers or cancels a subscription, the nodes of the overlay update their routing table accordingly by exchanging some pieces of information that represent the registration or cancellation of the consumer. The process starts at the consumer node and terminates at the producer node(s), following the shortest paths, and updating the routing tables along the way.

Updating routing table entries is a very fast operation and can be implemented easily using hash tables. However, the routers in our system do not only store and update subscriptions' entries, they also reduce the size of their routing tables as much as possible by using elaborate *aggregation* techniques, which are based on the detection and the elimination of subscription redundancies, as explained in Section 4.1 and 4.2.

Subscription aggregation allows us to dramatically improve the routing efficiency of the system both in terms of throughput and latency, because the time necessary to filter a message is proportional to the number of entries in the routing tables. On the other hand, aggregation also adds significant complexity and overhead to the routers, because they need to identify the containment relationships between incoming subscriptions and all the entries of their routing tables. In fact, the cost of subscription management in our system mainly results from those extensive covering checks that have to be performed by the routers when a subscription is registered or canceled.

To determine whether a given tree-structured subscription—also called "tree pattern" henceforth—contains another subscription, we can use the algorithm proposed in [34], which has a time complexity of $O(|S_1|.|S_2|)$, where $|S_1|$ and $|S_2|$ are the number of nodes of the two subscriptions being compared. Obviously, when an incoming subscription S must be tested for containment against all the other subscriptions $\{S_i\}$ in the routing table, iterative execution of the algorithm is clearly inefficient, since it would run in $\mathrm{O}(\sum_{i \leq n} |S_i|.|S|)$ time (where $|S_i|$ is the number of nodes of subscription S_i).

We have therefore designed a novel algorithm, termed XSearch, which efficiently identifies all the possible containment relationships between a given subscription and a possibly large set of subscriptions. This algorithm is described in the rest of this section.

5.2 Problem Statement

Consider a tree pattern S and a set $\mathcal{R}$ of n tree patterns, $\mathcal{R} = \{s_1, \cdots, s_n\}$, which we will refer to as the search set. Our algorithm runs in two different modes according to the relationships that we want to identify:

- *Contained mode* identifies the set $\mathcal{R}_{\supseteq}$ of all the tree patterns in $\mathcal{R}$ that are contained by S.

- *Contain mode* identifies the set $\mathcal{R}_{\subseteq}$ of all the tree patterns in $\mathcal{R}$ that contain S.

We refer to XSEARCH$_{\supseteq}$ and XSEARCH$_{\subseteq}$ as the algorithm running in *contained* and *contain* mode, respectively.

5.3 Data models

5.3.1 Definitions and Notations

Let u be a node of a tree pattern S; we denote by $label(u)$ the label of that node and by $child(u)$ the set of the child nodes of u in S. Recall that the label of node u can either be a wildcard ($*$), an ancestor/descendant operator ($//$), or a tag name. We define a partial ordering $\preceq$ on node labels such that if x and x' are tag names, then (1) $x \preceq * \preceq //$ and (2) $x \preceq x'$ iff $x = x'$.

We assume that a node u with label // has a unique child u'. This is due to the constraints on the format of XPath expressions (normalization). However, it is just a structural constraint, and not a restriction on the expressiveness of subscriptions. Indeed, nodes can have several children with label //.

5.3.2 Factorization Trees

Algorithm 9 $add(s, t, u)$

```
1: if ∃t' ∈ child(t) such that label(t') = label(u) and s ∉ sub(t') then
2:    sub(t') = sub(t') ∪ s
3: else
4:    create t' ∈ child(t) such that label(t') = label(u) and sub(t') = {s}
5: end if
6: for all u' ∈ child(u) do
7:    add(s, t', u')
8: end for
```

Our XSEARCH algorithm does not operate directly on the set of tree patterns $\mathcal{R}$, but on a "factorization tree" built from the set $\mathcal{R}$ and defined as follows. The factorization tree of $\mathcal{R}$, denoted $T(\mathcal{R})$, is a tree where each node t has two attributes: a label $label(t)$ similar to that of a node of a tree pattern, and a set of tree patterns $sub(t)$, which is a subset of $\mathcal{R}$. The root node r_T of $T(\mathcal{R})$ has no label and $sub(r_T) = R$. Initially, $T(\mathcal{R})$ consists of only its root node r_T. We incrementally add each tree pattern $s \in R$ to $T(\mathcal{R})$ with the recursive $add(s, r_T, r_s)$ function shown in Algorithm 9, where r_s is the root node of tree pattern S. The removal of a tree pattern from $T(\mathcal{R})$ is performed in a similar manner using Algorithm 10. Note that, to keep the presentation simple, we omitted the special case of the root node of the factorization tree in the addition and removal algorithms. By construction, $sub(t)$ is a set that contains all the tree patterns in $\mathcal{R}$, that comprise the single-path structure that starts at node r_T and ends at node t in $T(\mathcal{R})$ (r_T excluded).

Algorithm 10 $remove(s, t)$

```
1: for all t' ∈ child(t) such that s ∈ sub(t') do
2:    sub(t') = sub(t') \ {s}
3:    remove(s, t')
4:    if sub(t') = ∅ then
5:       remove t' from child(t)
6:    end if
7: end for
```

Intuitively, a factorization tree enables us to remove the redundancies between the tree patterns in $\mathcal{R}$ by "factorizing" identical branches. Thus, $T(\mathcal{R})$ is a compact representation of the tree patterns in $\mathcal{R}$. Figure 5.1 shows an example with six tree patterns and the corresponding factorization tree. It is important to note that the factorization tree is not unique; depending on the insertion order of the tree patterns, we can have distinct, equivalent trees. This does not affect the correctness of our XSEARCH algorithm, nor its performance.

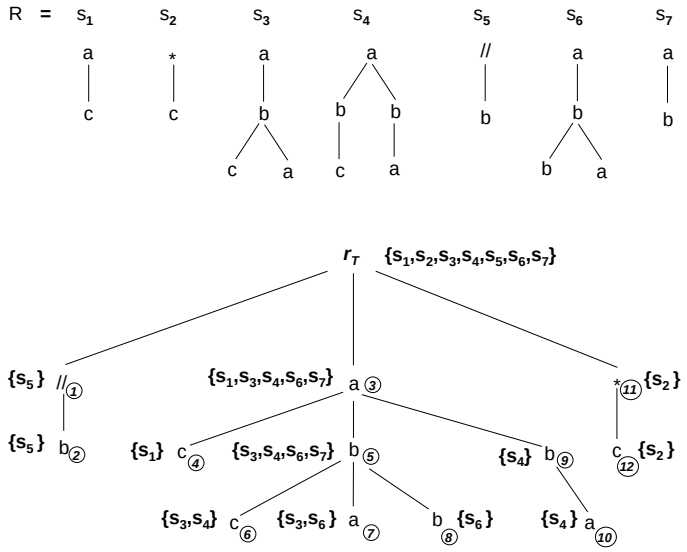


Figure 5.1: Six tree patterns and a corresponding factorization tree, where a node is represented by its label. Each node is associated with a set of tree patterns, shown between brackets.

Algorithm 11 XSEARCH$_{\supseteq}(t, u)$

```
1: if t is a leaf then
2:    XSEARCH⊇(t, u) = ∅
3: else
4:    if label(u) ≠ "//" then
5:       if u is a leaf then
6:          XSEARCH⊇(t, u) = ⋃_{label(t')⪯label(u)}^{t'∈child(t)} sub(t')
7:       else
8:          XSEARCH⊇(t, u) = ⋃_{label(t')⪯label(u)}^{t'∈child(t)} ⋂_{u'∈child(u)} XSEARCH⊇(t', u')
9:       end if
10:   else
11:      S_0 = XSEARCH(t, u')
12:      S_≥1 = ⋃_{t'∈child(t)} XSEARCH(t', u)
13:      XSEARCH⊇(t, u) = S_0 ∪ S_≥1
14:   end if
15: end if
```

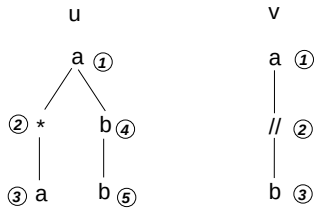
5.4 The Search Algorithm

We first describe the XSEARCH algorithm in contained mode. Consider a subscription set $\mathcal{R}$ and a corresponding factorization tree, $T(\mathcal{R})$. Let S be a single tree pattern. The algorithm works recursively on the nodes of S. When executed with the root nodes of $T(\mathcal{R})$ and S, XSEARCH$_{\supseteq}(r_T, r_s)$ returns the set $R_{\supseteq}$ of all tree patterns that are contained by S.

The search process is described in pseudo-code in Algorithm 11. Intuitively, it tries to locate the paths in $T(\mathcal{R})$ that are contained by S; the tree patterns that the union of those paths represent are also contained by S (lines 6 and 8). The process is slightly more complex when encountering an ancestor/descendant operator (//), because we need to try to map it to paths of length 0 (line 11) or ≥ 1 (line 12).

To better illustrate the workings of the XSEARCH$_{\supseteq}$ algorithm, consider the example runs shown in Figure 5.2. Two tree patterns, u and v, are matched against the factorization tree $T(\mathcal{R})$ of Figure 5.1. The nodes of u, v, and $T(\mathcal{R})$ are numbered in the figures for clarity; we refer to node number i of u, v, and $T(\mathcal{R})$ by u_i, v_i, and t_i, respectively. The different steps of the algorithm are detailed in the two execution traces (the $\hookrightarrow$ symbol represents recursive invocations of the algorithm).

The second algorithm, XSEARCH$_{\subseteq}$, is described in Algorithms 12 and 13 and works in a very similar manner. The major difference is that the algorithm works recursively on the nodes of $T(\mathcal{R})$,



$\textsc{XSearch}(r_T, u_1) = \textsc{XSearch}(t_3, u_2) \cap \textsc{XSearch}(t_3, u_4)$
$\hookrightarrow \textsc{XSearch}(t_3, u_2) = \textsc{XSearch}(t_4, u_3) \cup \textsc{XSearch}(t_5, u_3) \cup \textsc{XSearch}(t_9, u_3)$
  $\hookrightarrow \textsc{XSearch}(t_4, u_3) = \emptyset$
  $\hookrightarrow \textsc{XSearch}(t_5, u_3) = \{s_3, s_6\}$
  $\hookrightarrow \textsc{XSearch}(t_9, u_3) = \{s_4\}$
$\hookrightarrow \textsc{XSearch}(t_3, u_2) = \{s_3, s_6\} \cup \{s_4\} = \{s_3, s_4, s_6\}$
$\hookrightarrow \textsc{XSearch}(t_3, u_4) = \textsc{XSearch}(t_5, u_5) \cup \textsc{XSearch}(t_9, u_5)$
  $\hookrightarrow \textsc{XSearch}(t_5, u_5) = \{s_6\}$
  $\hookrightarrow \textsc{XSearch}(t_9, u_5) = \emptyset$
$\hookrightarrow \textsc{XSearch}(t_3, u_4) = \{s_6\}$

Finally: $\textsc{XSearch}(r_T, u_1) = \{s_3, s_4, s_6\} \cap \{s_6\} = \{s_6\}$

$\textsc{XSearch}(r_T, v_1) = \textsc{XSearch}(t_3, v_2)$
$\hookrightarrow \textsc{XSearch}(t_3, v_2) = S_0 \cup S_{\geq 1}$
$\hookrightarrow S_0 = \textsc{XSearch}(t_3, v_3)$
$\hookrightarrow S_{\geq 1} = \textsc{XSearch}(t_4, v_2) \cup \textsc{XSearch}(t_5, v_2) \cup \textsc{XSearch}(t_9, v_2)$
  $\hookrightarrow \textsc{XSearch}(t_3, v_3) = sub(t_5) \cup sub(t_9) = \{s_3, s_4, s_6, s_7\}$
  $\hookrightarrow \textsc{XSearch}(t_4, v_2) = \emptyset$
  $\hookrightarrow \textsc{XSearch}(t_5, v_2) = S'_0 \cup S'_{\geq 1}$
    $\hookrightarrow S'_0 = \textsc{XSearch}(t_5, v_3)$
    $\hookrightarrow S'_{\geq 1} = \textsc{XSearch}(t_6, v_2) \cup \textsc{XSearch}(t_7, v_2) \cup \textsc{XSearch}(t_8, v_2)$
      $\hookrightarrow \textsc{XSearch}(t_5, v_3) = sub(t_6) = \{s_6\}$
      $\hookrightarrow \textsc{XSearch}(t_6, v_2) = \emptyset$
      $\hookrightarrow \textsc{XSearch}(t_7, v_2) = \emptyset$
      $\hookrightarrow \textsc{XSearch}(t_8, v_2) = \emptyset$
    $\hookrightarrow S'_0 = \{s_6\}$
    $\hookrightarrow S'_{\geq 1} = \emptyset$
  $\hookrightarrow \textsc{XSearch}(t_5, v_2) = \{s_6\}$
  $\hookrightarrow \textsc{XSearch}(t_9, v_2) = S''_0 \cup S''_{\geq 1}$
    $\hookrightarrow S''_0 = \textsc{XSearch}(t_9, v_3)$
    $\hookrightarrow S''_{\geq 1} = \textsc{XSearch}(t_{10}, v_2)$
      $\hookrightarrow \textsc{XSearch}(t_9, v_3) = \emptyset$
      $\hookrightarrow \textsc{XSearch}(t_{10}, v_2) = \emptyset$
    $\hookrightarrow S''_0 = \emptyset$
    $\hookrightarrow S''_{\geq 1} = \emptyset$
  $\hookrightarrow \textsc{XSearch}(t_9, v_2) = \emptyset$
$\hookrightarrow S_0 = \{s_3, s_4, s_6, s_7\}$
$\hookrightarrow S_{\geq 1} = \{s_6\}$
$\hookrightarrow \textsc{XSearch}(t_3, v_2) = \{s_3, s_4, s_6, s_7\} \cup \{s_6\} = \{s_3, s_4, s_6, s_7\}$

Finally: $\textsc{XSearch}(r_T, v_1) = \{s_3, s_4, s_6, s_7\}$

Figure 5.2: Two $\textsc{XSearch}_{\supseteq}$ example runs.

trying to find paths in S that are contained by the tree patterns in $T(\mathcal{R})$. The recursive function in Algorithm 12 returns the subscriptions that do *not* contain S. A subscription t contains S if each branch of S is contained by some branch of t (line 12). Subscriptions that have longer (line 2) or incompatible (line 6) paths cannot contain S, whereas shorter paths (line 9) are acceptable. Finally, when encountering an ancestor/descendant operator (//), we need to try to map it to paths of length 0 (line 16) or ≥ 1 (line 17). Note that we implicitely introduce an artificial root node in the tree-structured subscriptions (denoted r_s for subscription S) in order to simplify the description of the algorithm. When called with the roots of the factorization tree and a subscription S, Algorithm 13 recusively searches for subscriptions that do not contain S and return the complement set with respect to $\mathcal{R}$.

We have illustrated the workings of the $\textsc{XSearch}_{\subseteq}$ algorithm in Figure 5.3, where tree pattern u of Figure 5.2 is matched against the factorization tree $T(\mathcal{R})$ of Figure 5.1.

5.5 Considerations

5.5.1 Complexity

Both Algorithms 11 and 12 perform in $O(|T(R)| \cdot |s|)$ time, where $|T(R)|$ is the number of nodes in the factorization tree and $|s|$ that in the expression being tested. This quadratic time complexity is due to the fact that each node in $T(\mathcal{R})$ and S is checked at most once. As for the space complexity, the size of the factorization tree $T(\mathcal{R})$ grows linearly with the number of tree patterns in the search set $\mathcal{R}$. However, by construction, the factorization tree typically requires much less space than would be needed to maintain the whole search set $\mathcal{R}$, that is, $|T(R)| \ll \sum_{s_i \in R} |s_i|$ when $|R|$ grows to large values (see Section 6.2).

Algorithm 12 XSEARCH$_{\subseteq}(t, u)$

```
1: if u is a leaf then
2:   XSEARCH⊆(t, u) = sub(t)
3: else
4:   if label(t) ≠ "//" then
5:     if ∄u' ∈ child(u), label(u') ⪯ label(t) then
6:       XSEARCH⊆(t, u) = sub(t)
7:     else
8:       if t is a leaf then
9:         XSEARCH⊆(t, u) = ∅
10:      else
11:        XSEARCH⊆(t, u) =
12:        ⋂_{label(u')⪯label(t)}^{u'∈child(u)} ⋃_{t'∈child(t)} XSEARCH⊆(t', u')
13:      end if
14:    end if
15:  else
16:    S0 = ⋃_{t'∈child(t)} XSEARCH(t', u)
17:    S≥1 = ⋂_{u'∈child(u)} XSEARCH(t, u')
18:    XSEARCH⊆(t, u) = S0 ∩ S≥1
19:  end if
20: end if
```

Algorithm 13 XSEARCH$_{\subseteq}(r_T, r_s)$

```
1: XSEARCH⊆(r_T, r_s) = sub(r_T) \ ⋃_{t'∈child(t)} XSEARCH⊆(t', r_s)
```

5.5.2 Correctness

In [80], the authors have shown that the containment problem is coNP-complete. Our XSEARCH algorithm (as well as that of [34]) is sound but not complete, i.e., it may fail to detect some containment relationships in rare pathological cases, but all the relationships that it reports are correct. Consequently, a router may fail to aggregate some valid subscriptions, but correctness is never violated. Some pathological cases where XSEARCH fails to report covering relationships are identified in the next section.

5.6 Proofs

In this section, we prove that Algorithms 11 returns sets $R_{\supseteq}$, except in some rare pathological cases.

5.6.1 Definitions and notations

Let t be a node in $T(\mathcal{R})$. We use the notation $r_T \rightarrow t$ to denote the single path tree pattern that comprises all the nodes from r_T to t in $T(\mathcal{R})$. We say that a subscription $S_i \in R$ is an *extension* of $r_T \rightarrow t$ iff S_i comprises the single path tree pattern $r_T \rightarrow t$, structurally. For example, '$/a/b/c/d$' is an extension of '$/a/b$'. Also, '$/a[./e/f]/b$' is an extension of '$/a/b$'.

We then have Property 7:

Property 7. *By construction, sub(t) is a set that contains all the tree patterns in $\mathcal{R}$ that are* extensions *of $r_T \rightarrow t$.*

Consider a subscription $S_i \in R$, and a node $t \in T(R)$ such that $S_i \in sub(t)$. Then, t is a node in S_i. We refer to $tree^{S_i}(t)$ as the tree pattern that consists of the tree of nodes *in subscription* S_i, rooted at node t. For example, in figure 5.1, $tree^{S_6}(t_5) = /b[./a]/b$. If u is a node in subscription S, then $tree^{S}(u)$ simply refers to the tree of nodes in S, rooted at node u. Finally, for a given node a with a child a', we note $/a/tree(a')$ to refer to the tree pattern that consists of node a with unique child a', and with descendants the tree of nodes rooted at node a'.

We now present Property 8, that concerns the ancestor/descendant operator (//). The property directly comes from the definition of the // operator, i.e., it matches a path of nodes of any length.

$\text{XSEARCH}(r_T, r_u) = \{S_1 \cdots S_7\} \setminus$
$(\ \text{XSEARCH}(t_1, r_u) \cup \text{XSEARCH}(t_3, r_u) \cup \text{XSEARCH}(t_{11}, r_u))$
$\hookrightarrow \text{XSEARCH}(t_1, r_u) = S_0 \cap S_{\geq 1}$
$\hookrightarrow S_0 = \text{XSEARCH}(t_2, r_u) = \{S_5\}$
$\hookrightarrow S_{\geq 1} = \text{XSEARCH}(t_1, u_1)$
$\hookrightarrow \text{XSEARCH}(t_1, u_1) = S'_0 \cap S'_{\geq 1}$
$\hookrightarrow S'_0 = \text{XSEARCH}(t_2, u_1) = \emptyset$
$\hookrightarrow S'_{\geq 1} = \text{XSEARCH}(t_1, u_2) \cap \text{XSEARCH}(t_1, u_4)$
$\hookrightarrow \text{XSEARCH}(t_1, u_2) = S''_0 \cap S''_{\geq 1}$
$\hookrightarrow S''_0 = \text{XSEARCH}(t_2, u_2) = \{S_5\}$
$\hookrightarrow S''_{\geq 1} = \text{XSEARCH}(t_1, u_3) = \{S_5\}$
$\hookrightarrow \text{XSEARCH}(t_1, u_2) = \{S_5\}$
$\hookrightarrow \text{XSEARCH}(t_1, u_4) = S^3_0 \cap S^3_{\geq 1}$
$\hookrightarrow S^3_0 = \text{XSEARCH}(t_2, u_4) = \emptyset$
$\hookrightarrow S^3_{\geq 1} = \text{XSEARCH}(t_1, u_5) = \{S_5\}$
$\hookrightarrow \text{XSEARCH}(t_1, u_4) = \emptyset$
$\hookrightarrow S'_{\geq 1} = \emptyset$
$\hookrightarrow S_1 = \emptyset$
$\hookrightarrow \text{XSEARCH}(t_1, r_u) = \emptyset$
$\hookrightarrow \text{XSEARCH}(t_3, r_u) = \text{XSEARCH}(t_4, u_1) \cup \text{XSEARCH}(t_5, u_1) \cup \text{XSEARCH}(t_9, u_1)$
$\hookrightarrow \text{XSEARCH}(t_4, u_1) = \{S_1\}$
$\hookrightarrow \text{XSEARCH}(t_5, u_1) = \text{XSEARCH}(t_6, u_4) \cup \text{XSEARCH}(t_7, u_4) \cup \text{XSEARCH}(t_8, u_4)$
$\hookrightarrow \text{XSEARCH}(t_6, u_4) = \{S_3, S_4\}$
$\hookrightarrow \text{XSEARCH}(t_7, u_4) = \{S_3, S_6\}$
$\hookrightarrow \text{XSEARCH}(t_8, u_4) = \emptyset$
$\hookrightarrow \text{XSEARCH}(t_5, u_1) = \{S_3, S_4, S_6\}$
$\hookrightarrow \text{XSEARCH}(t_9, u_1) = \text{XSEARCH}(t_{10}, u_4)$
$\hookrightarrow \text{XSEARCH}(t_{10}, u_4) = \{S_4\}$
$\hookrightarrow \text{XSEARCH}(t_9, u_1) = \{S_4\}$
$\hookrightarrow \text{XSEARCH}(t_3, r_u) = \{S_1, S_3, S_4, S_6\}$
$\hookrightarrow \text{XSEARCH}(t_{11}, r_u) = \{S_2\}$
$\hookrightarrow \text{XSEARCH}(r_T, r_u) = \{S_1 \cdots S_7\} \setminus \{S_1, S_2, S_3, S_4, S_6\}$
Finally: $\text{XSEARCH}(r_T, r_u) = \{S_5, S_7\}$

Figure 5.3: An $\text{XSEARCH}_{\subseteq}$ example run.

Property 8. *Consider tree patterns $tree(x)$ and $tree(u)$, where $label(u) = //$. Let u' be the unique child of node u (see Section 5.3.1). Then, we have:*

$$tree(x) \subseteq tree(u) \Leftrightarrow \begin{cases} tree(x) \subseteq\ tree(u') \\ or \\ \exists x' \in child(x),\ tree(x') \subseteq\ tree(u) \end{cases}$$

5.6.2 XSearch$_{\supseteq}$ proof

In this section, we prove that $\text{XSEARCH}_{\supseteq}(t, u)$ returns $R_{\supseteq}$, except in some pathological cases. Recall that given a subscription S, we want to find all subscriptions in $\mathcal{R}$ such that they are contained by S. t is a node in $T(\mathcal{R})$ and u a node in S. We proceed by ascendant recursion on the depth of nodes in $T(\mathcal{R})$ or nodes in S. The recursive property P is the following:

Recursive property *P*: Let $S_i \in R$ be an extension of $r_T \rightarrow t$. Then:

$$\exists t' \in child(t), \text{ such that } S_i \in sub(t') \text{ and } tree^{S_i}(t') \subseteq tree^S(u)$$
$$\Leftrightarrow$$
$$S_i \in \text{XSEARCH}_{\supseteq}(t, u)$$

We first prove the recursive property on recursion initiations, that is, when t or u is a leaf node.

If t is a leaf node: Then, t is also a leaf in S_i. There are no $t' \in child(t)$ such that $tree^{S_i}(t') \subseteq tree^S(u)$. Since $\text{XSEARCH}_{\supseteq}(t, u) = \emptyset$, Property P is satisfied.

If u is a leaf node: Suppose that there exists $t' \in child(t)$ such that $S_i \in sub(t')$ and $tree^{S_i}(t') \subseteq tree^S(u)$. Then, $label(t') \preceq label(u)$, and $S_i \in \text{XSEARCH}_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} sub(t')$.

Now suppose that $S_i \in \text{XSEARCH}_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} sub(t')$. Then, there exists $t' \in child(t)$ such that $label(t') \preceq label(u)$ and $S_i \in sub(t')$. Then, $tree^{S_i}(t') \subseteq tree^S(u)$ (u is a leaf, hence $tree^S(u)$ only consists of node u).

Finally, property P is verified.

We now consider the general case. We have to prove property P for a node t in $T(\mathcal{R})$ and u in S, assuming that it is true for nodes of higher depth in $T(\mathcal{R})$ and S.

General case: $label(u) \neq //$ We first focus on the case where $label(u) \neq //$.

Assume that there exists $t' \in child(t)$ such that $S_i \in sub(t')$ and $tree^{S_i}(t') \subseteq tree^S(u)$. Then, we have $label(t') \preceq label(u)$. Then, because $tree^{S_i}(t') \subseteq tree^S(u)$ and $label(u) \neq //$, we have:

$$\forall u' \in child(u), \exists t'' \in child(t') \text{ such that } S_i \in sub(t'') \text{ and } tree^{S_i}(t'') \subseteq tree^S(u')$$

In other words, each subtree rooted at each child of node u must contain some subtree rooted at t in S_i. The fact that $label(u) \neq //$ is important, because otherwise some child nodes of node t in S_i could be matched by $//$.

Then, because of the recursive property P, we have:

$$\begin{array}{l} \forall u' \in child(u), S_i \in \text{XSearch}_{\supseteq}(t', u') \\ \Leftrightarrow \\ \forall u' \in child(u), S_i \in \bigcap_{u' \in child(u)} \text{XSearch}_{\supseteq}(t', u') \end{array}$$

Finally, we have:

$$S_i \in \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} \bigcap_{u' \in child(u)} \text{XSearch}_{\supseteq}(t', u') = \text{XSearch}_{\supseteq}(t, u)$$

and the "left to right" implication of Property P is satisfied.

Now assume that:

$$S_i \in \text{XSearch}_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} \bigcap_{u' \in child(u)} \text{XSearch}_{\supseteq}(t', u')$$

Then, we have:

$$\begin{array}{l} \exists t' \in child(t), \text{ such that} \\ label(t') \preceq label(u) \text{ and } \forall u' \in child(u), S_i \in \text{XSearch}_{\supseteq}(t', u') \end{array}$$

Let us consider such a $t' \in child(t)$, with $label(t') \preceq label(u)$. Then, because of recursive property P, we have:

$$\forall u' \in child(u), \exists t'' \in child(t'), \text{ such that } S_i \in sub(t'') \text{ and } tree^{S_i}(t'') \subseteq tree^S(u')$$

Hence, $tree^{S_i}(t') \subseteq tree^S(u)$. Additionally, we trivially have $S_i \in sub(t')$ (since $S_i \in sub(t'')$), and the "right to left" implication of property P is satisfied.

Finally, property P is satisfied.

We now consider the case where $label(u) = //$.

General case: $label(u) = //$ Then, node u has a unique child u' (see Section 5.3.1).

Suppose that there exists $t' \in child(t)$ such that $S_i \in sub(t')$ and $tree^{S_i}(t') \subseteq tree^S(u)$. Then, because of Property 8, we either have:

- $tree^{S_i}(t') \subseteq tree^S(u')$

  OR

- $\exists t'' \in child(t'), tree^{S_i}(t'') \subseteq tree^S(u)$

If $tree^{S_i}(t') \subseteq tree^S(u')$, then because of recursive Property P, $S_i \in \text{XSEARCH}_{\supseteq}(t, u')$.

Now if $\exists t'' \in child(t'), tree^{S_i}(t'') \subseteq tree^S(u)$, then because of recursive Property P, $S_i \in \text{XSEARCH}_{\supseteq}(t', u)$. Hence, we have: $\exists t' \in child(t)$ such that $S_i \in \text{XSEARCH}_{\supseteq}(t', u)$, i.e. $S_i \in \bigcup_{t' \in child(t)} \text{XSEARCH}_{\supseteq}(t', u)$.

Finally, $S_i \in \text{XSEARCH}_{\supseteq}(t, u') \cup \left(\bigcup_{t' \in child(t)} \text{XSEARCH}_{\supseteq}(t', u)\right) = \text{XSEARCH}_{\supseteq}(t, u)$.

Now assume that $S_i \in \text{XSEARCH}_{\supseteq}(t, u)$. Then, we either have:

- $S_i \in \text{XSEARCH}_{\supseteq}(t, u')$

  OR

- $S_i \in \bigcup_{t' \in child(t)} \text{XSEARCH}(t', u)$

If $S_i \in \text{XSEARCH}_{\supseteq}(t, u')$, then because of recursive Property P, we have: $\exists t' \in child(t)$ such that $S_i \in sub(t')$ and $tree^{S_i}(t') \subseteq tree^S(u')$. Then, because of Property 8, we have $tree^{S_i}(t') \subseteq tree^S(u)$.

Now if $S_i \in \bigcup_{t' \in child(t)} \text{XSEARCH}_{\supseteq}(t', u)$, then there exists $'t \in child(t)$ such that $S_i \in \text{XSEARCH}_{\supseteq}(t', u)$. Because of recursive Property P, for one such t', we then have:

$$\exists t'' \in child(t') \text{ such that } S_i \in sub(t'') \text{ and } tree^{S_i}(t'') \subseteq tree^S(u)$$

Because of Property 8, we then have: $tree^{S_i}(t') \subseteq tree^S(u)$.

In either case, the "right to left" implication of Property P is verified.

Finally, property P is verified.

As a consequence, property P is verified at the root nodes of $T(\mathcal{R})$ and u. In other words, Algorithm 11 identifies all the subscriptions in $\mathcal{R}$ that are contained by S, except in some rare pathological cases that we identify in the next section.

5.6.3 Completeness

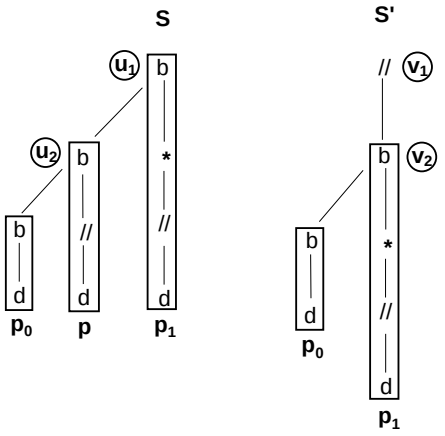


Figure 5.4: A pathological case where XSEARCH is incomplete.

As previously mentionned, the XSEARCH algorithm is sound but not complete, i.e., it may fail to detect some containment relationships in rare pathological cases, but all relationships that it reports are correct. Those cases were first identified by the authors of [80]. They occur when certain combinations of wildcards and ancestor/descendant operators appear in the XPath expressions. One such case is shown in Figure 5.4. Two expressions S and S' are represented. Note that path p_0 (p_1) is obtained when the // in path p is matched by a path of length 0 (≥ 1). We have $S \subseteq S'$. Indeed, consider a document D that matches S. If the // of path p in S is matched by a path of length 0 in

D, then path p_0 is matched (and p_1 in S' is matched by the same path in S). Hence, D matches S'. Now, if the // of path p in S is matched by any path of length greater than 1 in D, then path p_1 in S' is matched (and p_0 in S' is matched by the same path in S). Hence, D matches S'. Hence, in any case, a document that matches S also matches S', i.e., $S \subseteq S'$. XSEARCH fails to identify this containment relationship. This is due to the fact that in the XSEARCH algorithms, when node v_1 in S' is matched with a path of length 0, then path p_1 in S is found to be contained by the same path p_1 in S', but path p in S is not contained by path p_0 in S'. Hence, solution S_0 in the algorithms is empty (or returns S' if XSEARCH$_{\supseteq}$ is executed). Now when node v_1 in S' is matched with a path of length greater than 1 in the algorithms, path p_0 in S is contained by the same path in S', but path p is not contained by p_1 in S'. Hence, solution $S_{\geq 1}$ is empty (or returns S' if XSEARCH$_{\supseteq}$ is executed). Finally, XSEARCH does not report that S is contained by S'.

In fact, the failure of XSEARCH in those rare pathological cases is due to that of the 'left to right' implication of Property 8. Indeed, in the example we have $tree(u_1) \subseteq tree(v_1)$ (i.e. $S \subseteq S'$), but we have $tree(u_1) \not\subseteq tree(v_2)$ and $tree(u_2) \not\subseteq tree(v_1)$.

5.7 Extension to handle value constraints in tree patterns

The factorization tree data model and the XSEARCH algorithm operates on tree structured subscriptions, or tree patterns. However, as described in the previous sections, they only consider the structure of the tree patterns, and not their content. In this section, we describe an extension of our algorithm that enables us to handle the structure as well as the content, of tree patterns, i.e., the constraints on element values.

We consider a tree pattern S as a tree of node as described in Section 5.3.1, but in addition to that, each node $u \in s$ may contain one or more predicates on the value of u, and which we will refer to as $P_s(u)$. Those predicates may be combined together with the logical operators $\wedge$ (and) or $\vee$ (or).

The XSEARCH algorithm can be used with only minor modifications to the data models and to Algorithms 11 and 12.

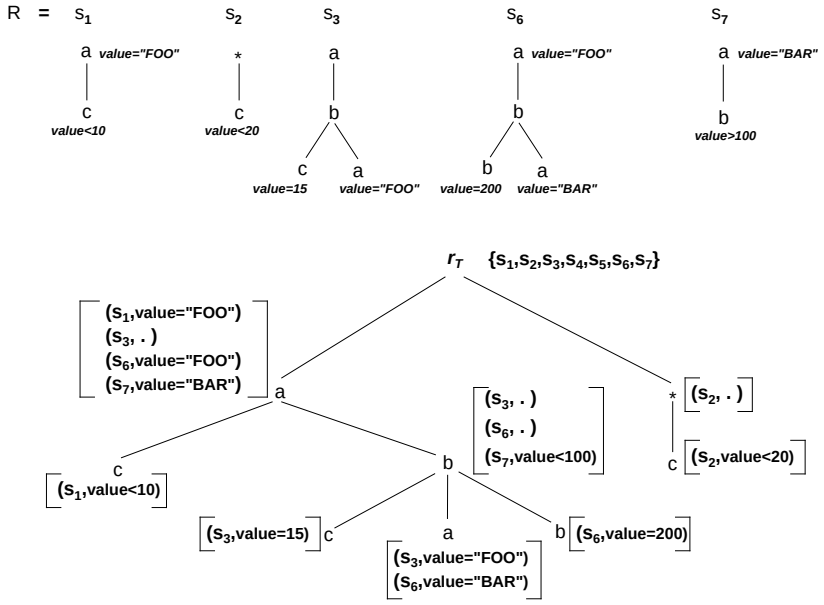


Figure 5.5: Six tree patterns and a corresponding factorization tree, where a node is represented by its label. Each node is associated with a set of tree patterns along with their predicates, shown between brackets.

5.7.1 Factorization tree

The factorization tree $T(\mathcal{R})$ that corresponds to the set of tree patterns $\mathcal{R}$ is modified as follows. For each node t in $T(\mathcal{R})$, $sub(t)$ is no longer a set of tree patterns as described in Section 5.3.1, but rather a set of pairs in the form $\big(S_i, P_{S_i}(t)\big)$. More formally, $sub(t) = \{(S_i, P_{S_i}(t))\}$ where each S_i is one of the subscriptions that were in $sub(t)$ in Section 5.3.1. We can also write: $sub(t) = \big\{\big(S_i, P_{S_i}(t)\big)\big\}_{S_i \text{ comprises } r_t \to t}$. The root node is an exception, $sub(r_T)$ is still the whole set $\mathcal{R}$.

An example is given in Figure 5.5.

Algorithm 14 XSEARCH$_{\supseteq}(t, u)$

if t is a leaf **then**
  XSEARCH$_{\supseteq}(t, u) = \emptyset$
else
  if $label(u) \neq$ "//" **then**
    if u is a leaf **then**
      XSEARCH$_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} \{S_i \in sub(t'), P_{S_i}(t') \preceq P_s(u)\}$
    else
      XSEARCH$_{\supseteq}(t, u) =$
      $\bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} \Big(\big(\bigcap_{u' \in child(u)} \text{XSEARCH}_{\supseteq}(t', u') \big) \cap \{S_i \in sub(t'), P_{S_i}(t') \preceq P_s(u)\} \Big)$
    end if
  else
    $S_0 =$XSEARCH(t, u')
    $S_{\geq 1} = \bigcup_{t' \in child(t)}$XSEARCH$(t', u)$
    XSEARCH$_{\supseteq}(t, u) = S_0 \cup S_{\geq 1}$
  end if
end if

5.7.2 Algorithms

The extended versions of the XSEARCH algorithms are presented in Algorithms 14 and 15 and explained as follows. We extend the $\preceq$ relation to predicates as follows: $P_1 \preceq P_2 \Leftrightarrow (n \text{ verifies } P_1 \Rightarrow n \text{ verifies } P_2)$.

Algorithm 15 XSEARCH$_{\subseteq}(t, u)$

if u is a leaf **then**
  XSEARCH$_{\subseteq}(t, u) = sub(t)$
else
  if $label(t) \neq$ "//" **then**
    if $\nexists u' \in child(u), label(u') \preceq label(t)$ **then**
      XSEARCH$_{\subseteq}(t, u) = sub(t)$
    else
      if t is a leaf **then**
        XSEARCH$_{\subseteq}(t, u) = \bigcap_{label(u') \preceq label(t)}^{u' \in child(u)} \{S_i \in sub(t), \overline{P_s(u') \preceq P_{S_i}(t)}\}$
      else
        XSEARCH$_{\subseteq}(t, u) =$
        $\bigcap_{label(u') \preceq label(t)}^{u' \in child(u)} \Big(\big(\bigcup_{t' \in child(t)} \text{XSEARCH}_{\subseteq}(t', u') \big) \cup \{S_i \in sub(t), \overline{P_s(u') \preceq P_{S_i}(t)}\} \Big)$
      end if
    end if
  else
    $S_0 = \bigcup_{t' \in child(t)}$XSEARCH$(t', u)$
    $S_{\geq 1} =$XSEARCH(t, u')
    XSEARCH$_{\subseteq}(t, u) = S_0 \cap S_{\geq 1}$
  end if
end if

Algorithm 16 XSEARCH$_{\subseteq}(r_T, r_s)$

XSEARCH$_{\subseteq}(r_T, r_s) = sub(t) \setminus \bigcup_{t' \in child(t)}$ XSEARCH$_{\subseteq}(t', r_s)$

XSearch$_{\supseteq}(t, u)$. The solutions are the same as in Algorithm 11, except that we must check that the conditions concerning the predicates are verified. That is, for each solution returned by Algorithm 11, we only keep the subscriptions such that their predicates at node t' is contained by the predicate of S at node u, that is: $\{S_i \in sub(t'), P_{S_i}(t') \preceq P_s(u)\}$ (lines 6 and 9 in Algorithm 14).

XSearch$_{\supseteq}(t, u)$. Similarly, the solutions returned by Algorithm 15 are the same as for Algorithm 12 except that we must include the subscriptions such that their predicates at the given node are not verified, that is: $\{S_i \in sub(t), \overline{P_s(u') \preceq P_{S_i}(t)}\}$ (lines 9 and 12 in Algorithm 15). Recall that XSEARCH$_{\subseteq}$ returns the subscriptions that do *not* contain S.

Chapter 6

Reliability and performance evaluation

6.1 Reliability in XNet

6.1.1 Motivations

In this section, we specifically address the issue of *reliability* in our XNet XML content network.

We have implemented several mechanisms to ensure reliable operation of our XNet system despite the occurrence of router or link failures. The primary objective of these mechanisms is to implement *reliable subscription advertisement*, that is, to maintain a *consistent shared state* in the system in spite of transient failures.

A secondary goal is to ensure reliable delivery of producer events; although desirable, this feature is of lesser importance because undelivered messages have no impact on the consistency of the content routing system.

The mechanisms described in this section take different approaches to failure recovery and offer various tradeoffs in terms of cost and benefits. They are also complementary in that they can be easily combined within the same network. We present two recovery-based approaches to reliability, which strive to maintain a consistent global state upon failure. We then discuss a third approach, orthogonal to the other two, which uses redundancy to *mask* problems and provide continuous service despite failures.

Note that, given a spanning tree rooted at a producer, the failure of a router directly affects the neighboring routers downstream from the failed node as they cannot anymore propagate subscription registrations and cancellations towards the producer at the root of the tree. In contrast, the failure of a link only affects the router downstream from the failed link; we can therefore consider router failure as a generalization of link failures, and we will only consider the former type of failures in the rest of the section.

Also, we only focus on the case of *routing* node failures that can be dealt with transparently by the infrastructure. The failure of a producer node will prevent the distribution of events and force the publisher application to switch over to another node. Similarly, the failure of a consumer node will affect all the attached consumers and must be handled explicitly by the subscriber application.

6.1.2 The *Crash/Recover* Scheme

Overview

The *Crash/Recover* scheme has been designed to cope efficiently and locally with temporary router or link failures. It relies on the assumption that a faulty link or router will recover after a short time. The *Crash/Recover* scheme implements *reliable* subscription advertisement. During the downtime period, the producers and consumers can still publish and subscribe to events, i.e., the failure is transparent. After the faulty router or link recovers, the system must reach the same consistent state as if no failure had occurred.

The *Crash/Recover* scheme relies upon a few key mechanisms to cope with transient failures. First, a recovery database is maintained in stable storage on each router. When the router fails, it can recover its state before the crash. Second, the use of the TCP protocol ensures the reliable and ordered delivery of subscriptions and documents. Third, a retransmission buffer coupled with a selective positive acknowledgment scheme is implemented between a router R and its upstream router U. Its purpose is to save the changes that occurred during the downtime of U so that, when it recovers, it can catch up and "roll forward" to a consistent state that corresponds to the current consumer population. Finally, sequence numbers are embedded in all messages to detect duplicates upon recovery and guarantee routing table consistency.

Algorithm 17 On receiving $Adv(sn)$ from interface i

```
1: if 0 < sn ≤ hr_i then                                   {Duplicate advertisement}
2:    Send Ack(sn) down interface i
3: else if sn = hr_i + 1 then                              {Expected advertisement}
4:    hs ← hs + 1
5:    Update routing table with XRoute and generate Adv_out(hs)
6:    hr_i ← hr_i + 1
7:    RetrBuf ⟵^append Adv_out(hs)
8:    Backup log and routing table in recovery database
9:    Send Ack(sn) down interface i
10:   Send Adv_out(hs) upstream
11: end if
```

Algorithm 18 On receiving $Ack(sn)$ from upstream

```
1: if Adv_out(sn) is found in RetrBuf then
2:    Remove Adv_out(sn) from log
3:    Backup log in recovery database
4: end if
```

Algorithm 19 On receiving $Back$ from upstream

```
1: Send RetrBuf upstream
```

Algorithm

The pseudo-code of the *Crash/Recover* protocol is given in Algorithms 17, 18, 19, and 20. Consider router R with n downstream interfaces. Let D_i be the router downstream interface i. Each time D_i sends an advertisement to router R, it includes in it a strictly increasing sequence number (unique between R and D_i). Let hr_i be the highest sequence number received from D_i, i.e., R has received from D_i all the advertisements with sequence number $sn \leq hr_i$. Similarly, hs is the highest sequence number that router R sent to its upstream router. Sequence numbers are used for the positive acknowledgment mechanism and to filter out duplicate advertisements that may be received after a link or router failure.

Each router R maintains a *log* that stores the latest non-acknowledged advertisements sent to its upstream router, as well as the current values of hs and $hr_1 \cdots hr_n$. The log and the routing table of router R are backed up in a *recovery database* (see Figure 6.1), which is written atomically to stable storage as soon as its state is updated (line 8 in Algorithm 17 and line 3 in Algorithm 18).

When router R receives an advertisement $Adv(sn)$ from interface i, it first checks if the advertisement is a duplicate by comparing sn with hr_i (lines 1 and 3 in Algorithm 17). If that is the case, R sends an acknowledgment to D_i and ignores the advertisement. Otherwise, we have $sn = hr_i + 1$ and we process the advertisement (it is trivial to see from the algorithm and the FIFO ordering property of TCP that we cannot have $sn > hr_i + 1$). R updates its routing table, generates an outgoing advertisement for its upstream router, increments hs and hr_i, and sends an acknowledgment D_i only after local updates have been saved on stable storage (lines 4–9 in Algorithm 17); this guarantees that D_i will resend its advertisement in case R fails before the recovery database has been updated.

Algorithm 20 On recovering from failure

1: Recover routing table and log from recovery database
2: Send $RetrBuf$ upstream
3: Send $Back$ downstream all interfaces

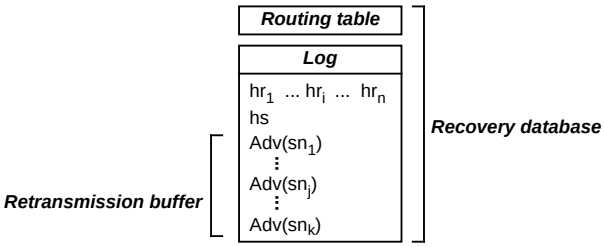


Figure 6.1: Format of the recovery database.

The retransmission buffer is a list of advertisements. Each time router R is about to send an advertisement $Adv_{out}(sn)$ to its upstream router U, it appends $Adv_{out}(sn)$ to its retransmission buffer (line 7 in Algorithm 17). When U has received it *and* has updated its routing table accordingly, it sends an acknowledgment for it back to router R (lines 2 or 9 in Algorithm 17), which removes $Adv_{out}(sn)$ from its retransmission buffer (line 2 in Algorithm 18).

If router R crashes, the advertisements that it should have received during the crash duration are not acknowledged and are thus stacked in its downstream routers' retransmission buffer. When R recovers, it first restores its state from the recovery database (line 1 in Algorithm 20). Then, it sends a $Back$ message to its downstream routers, (line 3 in Algorithm 20) to trigger the delivery of the advertisements that were stacked in their logs (Algorithm 19). From the point of view of router R and the routers upstream, everything looks as if R had never failed, except that the "missed" advertisements are received in bursts. After a certain period of time, which we refer to as the *recovery delay*, those routers have updated their routing table and the global system state reflects again the current consumer population.

The fact that the retransmission buffer is backed up in the recovery database and is retransmitted upon recovery before sending the $Back$ message (line 2 in Algorithm 20) handles the case when one of R's downstream router, D_i, fails while R is down. When recovering, D_i must first send to R the advertisements stored in its retransmission buffer before processing those received from its downstream routers, so as to preserve consistent ordering of the advertisements sent to R.

An example is illustrated in Figures 6.2(a) and (b). Router R receives S_1 from router D_1. It updates its routing table, sends an acknowledgement back to D_1, sends an outgoing advertisement for S_1 upstream and receives an acknowledgement in response. Then, router R receives advertisement for S_2 from D_1, updates its routing table and log and immediately crashes (at line 8 of Algorithm 17). Router R did not receive the advertisements sent by D_2 and D_3. Consequently, those advertisements are stacked in the retransmission buffer of the corresponding router. Also, the outgoing advertisement for subscription S_2 is stacked in router R's retransmission buffer. The recovery phase is illustrated in Figure 6.2(b). When router R recovers, it first restores its routing table and its log. Then, it immediately re-transmits the advertisements stacked in its retransmission buffer, here S_2. It then sends a $Back$ message to each downstream router which, subsequently, re-transmits the advertisements stacked in their retransmission buffer. For each of them, at the exception of S_2 received from D_1, router R updates its routing table and log, sends an acknowledgement downstream and finally sends the advertisement upstream. As for $(1 : S_2)$ received from D_1, router R detects that it is a duplicate (since an an advertisement with that sequence number was previously received from D_1). R just sends an acknowledgement back to D_1.

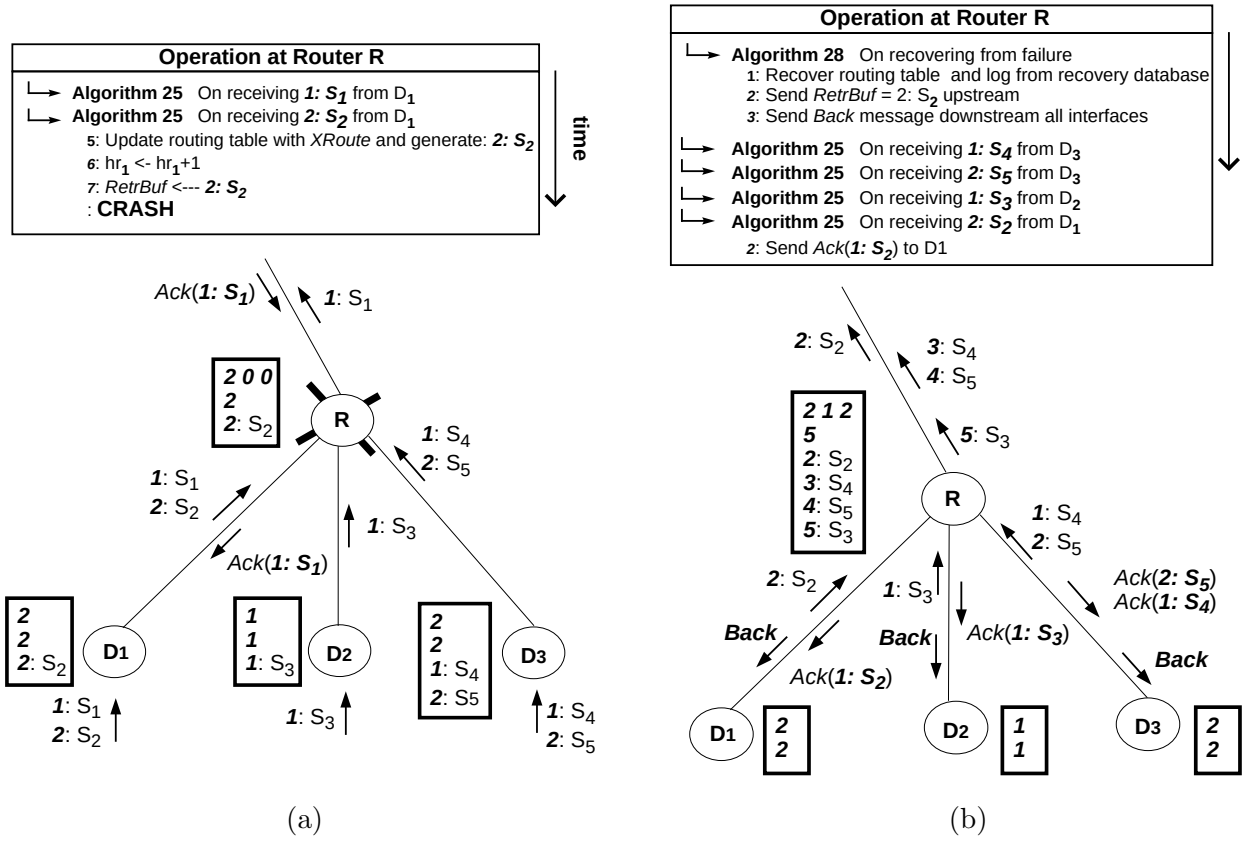


Figure 6.2: Illustration of the *Crash/Recover* scheme with a router R and three downstream routers $D_1 \cdots D_3$. Logs are shown in thick frames next to the routers. Each subscription advertised is preceded by its sequence number. The operations performed at router R are shown in the above box, in chronological order.
(a) Normal operation (b) Recovery phase

Reliable subscription advertisement

The *Crash/Recover* scheme implements *reliable* subscription advertisement. Indeed, in the absence of failures, the TCP protocol ensures the *reliable* and *ordered* delivery of advertisements. Subscription advertisement is then *correct* as explained in Section 4.2.4.

Now suppose that router R failed. Upon recovery, router R recovers the routing table that it had just before the crash occurred. Then, it receives all the advertisements that it should had received from its downstream neighbors, had it not failed. The advertisements received from a downstream router are received in the *same* order as they were received and processed at that downstream router. Hence, subscription advertisement is *correct* at router R, once it has received and processed all the advertisements received from downstream. In addition, upon recovery, and before processing any advertisements from downstream, router R retransmits all the advertisements that were not acknowledged by its upstream router R_{up} (some of them may be duplicates for R_{up}), in the same order as it (R) received and processed them. Then, it processes the advertisements received from downstream as in the case of normal operation. Consequently, subscription advertisement is correct at router R_{up}, once the advertisements sent by R have been handled. All the routers upstream R_{up} then process them and update their routing tables as if no failure occurred. Finally, after a recovery delay, that corresponds to the time necessary to handle the buffered advertisements, subscription advertisement is correct once again, and the system's state is consistent with the *actual* consumer population.

6.1.3 The *Crash/Failover* Scheme

Motivations

The *Crash/Recover* scheme was based on the assumption that a failed router R will recover after a reasonably short period of time, during which its downstream routers are buffering advertisements. However, the downtime duration of router R may be very long, causing buffers to grow huge or overflow. When R eventually recovers, many advertisements will transit along the paths from R's downstream routers to the producer nodes, potentially creating bottlenecks and delaying system recovery.

The *Crash/Failover* scheme is based on the principle that the downstream routers of a crashed router R do not wait for its recovery, but instead reconnect to another router and bring back their routing tables to a consistent state. Thus we make the assumption that every router R in the network knows at least one additional router other than its direct neighbors, to which it can connect if its upstream router fails. This scheme is very similar to primary/backup replication [82] and we will refer to the additional router as the R's *backup* router, denoted by B_R. Note that, obviously, B_R cannot be located downstream from R with respect to the producer as we must maintain a valid spanning tree after reconnection.

Protocol

The *Crash/Failover* protocol relies on the fact that every router R has a precise summary of all the subscriptions that its downstream neighbors are interested in. It can thus register/cancel any of these subscriptions at any time by sending an advertisement to its upstream router U, which see the advertisement as if it were the result of a consumer registering/canceling the subscription.

Consider a router R, its upstream router U linked to R via interface I, the set of R's downstream routers $\{D_i\}_{i\leq n}$ and their respective backup routers $\{B_{D_i}\}$. When a downstream router D_i detects that its upstream router R has failed and is unlikely to recover soon (e.g., after a reasonably long timeout), it switches over to its backup router B_{D_i} as new upstream router and registers all the subscriptions stored in its routing table, as if they had just originated from "real" consumers. Note that there are typically far less subscriptions than consumers downstream from D_i because of subscription aggregation. In fact, only the subscriptions that have not been substituted need to be registered. Consider for example an entry for subscription S in D_i's routing table, $entry(S)$, and its T field at interface I^R, T^{I^R} (I^R is the upstream interface at node D_i towards node R). Then, the advertisement sent to backup router B_{D_i} is: $(S\ ;\ n_S\ ;\ r_S)$, where $n_S = T^I.x$ and $r_S = R_S$. This advertisement accounts for all instances of subscription S, plus all instances of the subscriptions that are aggregated in S either through representation or substitution operations (see definition of R field in Property 4). At node B_{D_i}, those r_S instances of subscriptions will be represented by S at the incoming interface (as seen in Property 5 and the RTU algorithms).

Once every router D_i has reconnected to B_{D_i}, U can cancel all the subscriptions that were registered through interface I from its routing table to reestablish perfect routing on the path from the producer to U. Note that each subscription must be "totally cancelled". As we have seen in Section 4.2.5 and Algorithm 8, this may require that some subscriptions be reinserted in the routing tables of the upstream nodes. However, the present case is simpler. Indeed, consider a given subscription S that was previously registered at interface I. Since router R has crashed, from the point of view of router U, there are no more consumers interested in subscription S downstream interface I. Also, there are no more consumers interested in the subscriptions that were represented by S at interface I. Hence, those subscriptions do not have to be reinserted at node U or at the nodes upstream. At node U, only the subscriptions that are substituted to S may have to be reinserted (we have identified the cases in which this has to be done in Section 4.2.5 and Algorithm 8). Consequently, node U proceeds as if it received advertisement $(S\ ;\ n_S\ ;\ 0)$ from node R, where $n_S = -T_S^I.x$.

Reliable subscription advertisement

Clearly, after the recovery procedure has completed, the system state is again consistent with the consumer population. Indeed, router U, and all further upstream routers, have cancelled all the subscriptions that were advertised by router R. In addition, each downstream router D_i have registered all the subscriptions that have not been substituted to its backup router B_{D_i}. From the point of view of that router, those registrations are seen as "usual" registrations, i.e., as the result of a consumer registration. At router B_{D_i}, once those registrations have been processed, subscription advertisement is then correct, as explained in Section 4.2.4.

Note that, if D_i has incorrectly suspected R to have failed (e.g., because of a link failure) and has switched over to B_{D_i}, R will cancel all the subscriptions that were registered by D_i. Although resource consuming, this does not affect the *correctness* of subscription advertisement.

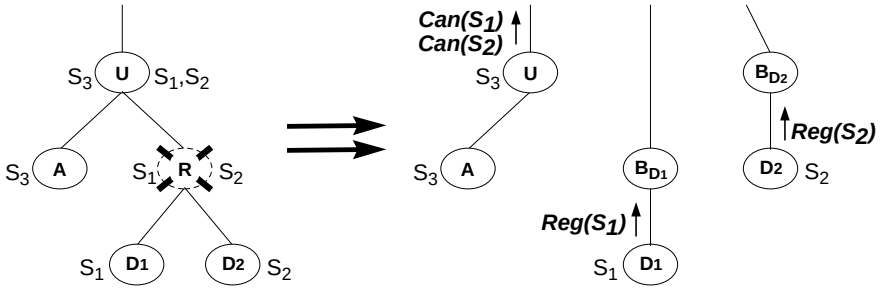


Figure 6.3: Recovering from the crash of router R with the *Crash/Failover* scheme.

Example 4. *Figure 6.3 illustrates a simple* Crash/Failover *scenario (the subscriptions that each node is interested in are represented next to the interface they came from; the state before the failure is represented on the left and recovery phase on the right). Routers D_1 and D_2 are interested in subscriptions S_1 and S_2 respectively, while router A is interested in S_3. When R crashes, routers D_1 and D_2 connect to their backup router B_{D_1} and B_{D_2} and register their subscription S_1 and S_2 ("Reg" messages). Thereafter, U can remove all subscriptions previously registered by R from its routing table and propagate the changes upstream ("Can" messages).*

Combining the two recovery protocols

The *Crash/Failover* protocol can be advantageously combined with the *Crash/Recover* protocol to deal with temporary link of node failures. If the failure duration reaches a predefined threshold, then the affected routers will switch over to a backup. The subscriptions received from downstream routers are buffered and processed after completion of the reconnection phase. Note that, in the case of simultaneous failures, it might not be possible to use the *Crash/Failover* protocol (e.g., because backup routers have also failed) and the system has to wait for some of the crashed routers to recover.

6.1.4 Masking Failures with *Redundant Paths*

The *Crash/Recover* and the *Crash/Failover* schemes suffer from two major drawbacks. First, the service is interrupted for the duration of the failure or until the overlay network has reconfigured. Second, they generate upon recovery an upstream traffic of advertisements that can be important, with each advertisement involving routing table updates at the traversed routers. To alleviate these drawbacks, we can combine these schemes with a masking strategy based on *Redundant Paths*, which improves availability by providing uninterrupted service despite failures. In particular, events can be delivered reliably and timely even though some of the routers fail.

The *Redundant Paths* strategy is based on the same principle as active replication [82]. It makes the assumption that each router R has at least one alternate route to the producer. The routing

information that corresponds to router R is replicated in the routing tables of the alternate routes. The implementation of the *Redundant Paths* strategy does not require other modifications to the XROUTE protocol than sending advertisements to all upstream routers (rather than a single one).

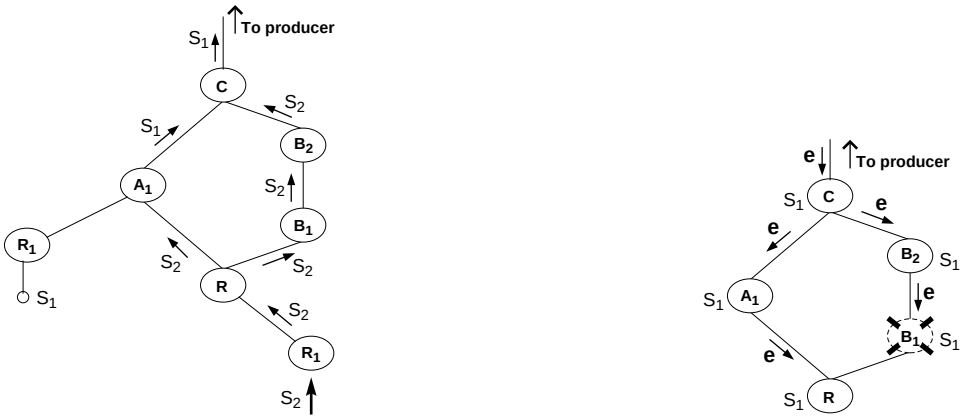


Figure 6.4: (a) routing tables replication with the *Redundant Paths* strategy (b) guarantee of event delivery with the *Redundant Paths* strategy.

An example is illustrated in Figure 6.4(a). Router R has two routes to the producer, via routers A_1 and C, and via routers B_1, B_2 and C (the remaining part is common to the two routes and is not shown in the example). When node R receives an advertisement for subscription S_2, it updates its routing table and sends an outgoing advertisement to both upstream interfaces, towards A_1 and B_1. Note that S_2 has been substituted by S_1 which was previously registered at node A_1. Subsequently, S_1 is advertised at node C. In contrast, S_2 is not aggregated on route $B_1 - B_2 - C$ and is directly advertised. Also, the order in which the advertisements from A_1 and B_2 arrive is not important. As we have seen in Section 4.2, the result at the node upstream C is the same, that is, S_2 being represented at the incoming interface.

If router R has n alternate routes to the producer, it is resilient to the failure of at least $n-1$ upstream routers (in the case of multiple producers, R should have n alternate routes to each producer, but those routes may share common sub-paths). When some routers on a route fail, the routers on the other routes are still consistent with the consumer population and R will keep receiving documents from those routes.

As previously mentioned, it is important to note that the *Redundant Paths* strategy increases the availability (liveness) of the system, but does not deal with recovery. It should be combined with the *Crash/Recover* or *Crash/Failover* protocols to ensure consistent recovery from a failure. The major drawback of the *Redundant Paths* strategy is that every subscription and event will be sent over multiple routes and thus increase bandwidth utilization. Further, routers and consumers must detect and filter out duplicate events.

Figure 6.4(b) shows an example of the event delivery guarantee property of the *Redundant Path* strategy. Router R has two routes to the producer: via routers A_1 and C, and via routers B_1, B_2 and C (the remaining part is common to the two routes and is not shown in the example). Router R is resilient to the failure of router A_1, and to the simultaneous failures of routers B_1 and B_2. In the example, router B_1 crashes and R still receives event e via the route $C \rightarrow A_1 \rightarrow R$.

6.1.5 Other issues

Reliability of Published Events

Under normal operation, the reliable delivery of published events is ensured by TCP. Guaranteed delivery in the case of failures can be implemented in the same manner as subscriptions in the *Crash/Recover* scheme, by using acknowledgments in combination with a retransmission buffer and

a persistent data storage. However, this approach has a high cost in terms of memory and bandwidth requirements as the event publishing rate is typically much higher than the subscription registration rate. Further, events published in content-based networks often need to be delivered timely or not at all, and buffering them is essentially useless; in such cases, one should use the *Redundant Paths* strategy to ensure timely event delivery despite failures. Note again that events do not modify the shared state of the system and the loss of some of them only affects the quality of service experienced by the consumers.

Case of multiple producers

We now consider the case of multiple producers. We examine the impact on each recovery strategy, and the extensions that may have to be implemented.

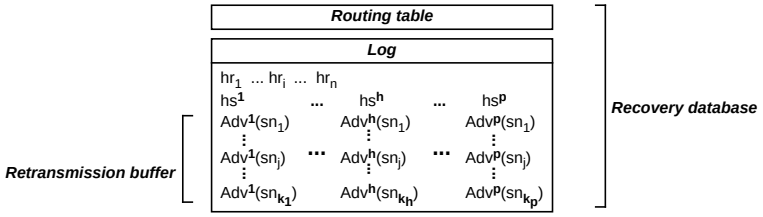


Figure 6.5: Format of the recovery database, extended to support multiple producers

***Crash/Recover* scheme.** The *Crash/Recover* scheme applies to the case of multiple producers with only minor modifications. We first need to modify the format of the recovery database to support multiple producers. Consider router R, with p upstream interfaces, each leading to different producer(s). The format of the recovery database is illustrated in Figure 6.5.

We now have p "highest sent" fields, each corresponding to a different upstream interface. hs_k is the highest sequence number sent to the router upstream interface I^k_{up}. The retransmission buffer now consists of a set of p "single" retransmission buffers, each corresponding to an interface I^k_{up}. We will refer to $RetrBuf^k$ as the retransmission buffer corresponding to interface I^k_{up} (k^{th} column of the "single" retransmission buffer).

The *Crash/Recover* protocol extended to the case of multiple producers is almost identical to the case of a single producer. We only need to consider that for each upstream interface, there is a different outgoing advertisement, retransmission buffer and hs field. Nevertheless, for completeness, we included its pseudo-code in Appendix B, in Algorithms 32, 34, 35 and 33.

***Crash/Failover* scheme.** The *Crash/Failover* scheme adapts to the case of multiple producers with only minor modifications. Consider router R with downstream routers $\{D_i\}$ and upstream routers $\{U_i\}$. For each such router D_i, R was the path leading to a set of producers $\{P_k\}$. For each such producer P_k, D_i must reconnect to a backup router that is not downstream router R with respect to producer P_k. Hence, each router must know at least p additional routers, but all of them are not necessarily different. The failover then proceeds as in Section 6.1.3. Each downstream router D_i reconnects to its backup routers and registers again the subscriptions in its routing table that were not substituted, with respect to interface I^i, which was the upstream interface at D_i towards R (see Section 4.2.6). Note that interface I^i is renumbered when the reconnection has been done. It then typically becomes the last interface.

As for routers U_i, they cancel the subscriptions that were registered by R, as explained in Section 6.1.3, and considering each upstream interface, as in Section 4.2.6.

Note that a downstream interface may be an upstream one as well. In other words, an upstream router U_i may be a downstream router D_k.

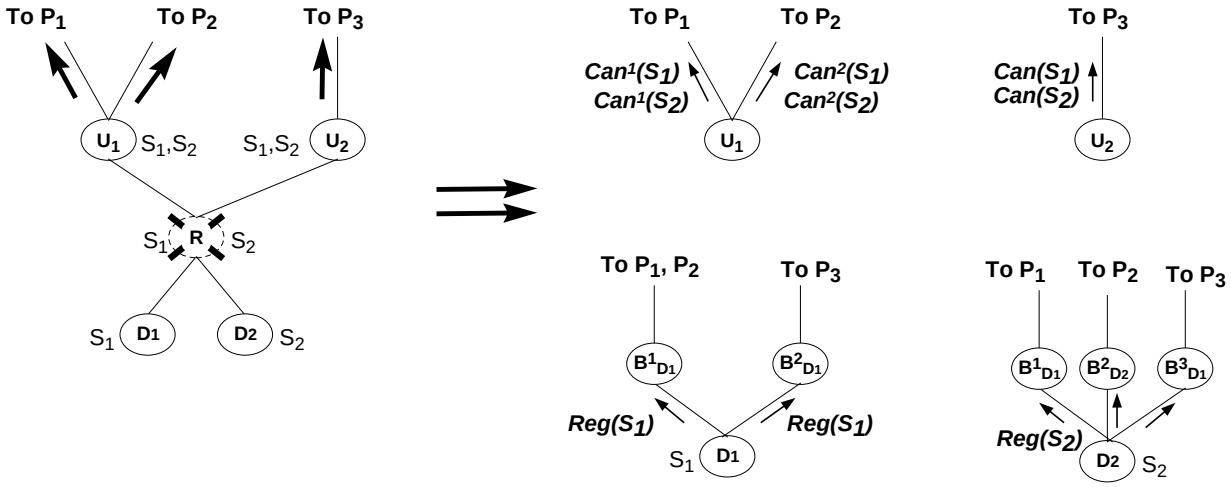


Figure 6.6: Recovering from the crash of router R with the *Crash/Failover* scheme.

An example is illustrated in Figure 6.6 When router R fails, downstream routers D_1 and D_2 connect to their backup routers. D_1 connect to router $B^1_{D_1}$ as the backup router leading to producers P_1 and P_2 and to router $B^2_{D_1}$ as the router leading to producer P_3. Router D_2 connect to $B^1_{D_2}$ ($B^2_{D_2}$, $B^3_{D_2}$) as the backup router leading to producer P_1 (P_2, P_3). Routers D_1 and D_2 then register their subscription S_1 and S_2 ("Reg" messages). Thereafter, upstream routers U_1 and U_2 can remove all subscriptions previously registered by R from their routing table, considering each upstream interface, and propagate the changes upstream ("Can" messages).

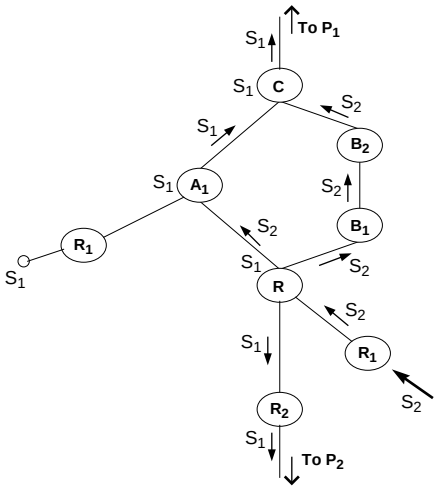


Figure 6.7: The *Redundant Paths* strategy in the case of multiple producers.

***Redundant Paths* scheme.** The *Redundant Paths* scheme easily adapts to the case of multiple producers. The only issue is that for a given router R, we must make the difference between two redundant upstream interfaces I^1_a and I^1_b, that lead to the same producer and two upstream interfaces I^1 and I^2 that lead to different producers. I^1_a and I^1_b are logically considered as the same interface. When a routing table update is made, the same outgoing advertisement is sent to I^1_a and to I^1_b, for replication purposes. In contrast, the routing table update is different when interface I^1 or I^2 is considered and a different outgoing advertisement is sent to each of them, as explained in

Section 4.2.6. The purpose is not to replicate routing tables but to propagate consumer membership changes to each spanning tree(s) that corresponds to interfaces I^1 and I^2.

An example is illustrated in Figure 6.7. At node R, there are three physical interfaces, $R \rightarrow A_1$, $R \rightarrow B_1$ and $R \rightarrow R_2$. However, $R \rightarrow A_1$ and $R \rightarrow B_1$ are logically considered as the same interface (the one that leads to producer P_1). Consequently, R updates its routing table and sends the same outgoing advertisement to A_1 and B_1, for replication purposes. Note that subscription S_1 was ignored since from the point of view of the spanning tree rooted at P_1, there are no consumers interested in S_1 downstream router R. As a consequence, S_2 is not substituted and is directly advertised upstream (see Section 4.2.6). In contrast, interface $R \rightarrow R_2$ leads to a different producer. S_1 has an entry when considering that interface, and subscription S_2 is substituted by it. S_1 is subsequently advertised towards P_2.

6.2 Performance evaluation

6.2.1 Overview

A major part of our efforts were devoted to building working prototypes and conducting extensive experimental evaluation of our XNet XML content routing network and its various components. The results are presented in this section.

We first assess the performance of the XRoute protocol when deployed on a simulated network. We have seen in the previous sections that XRoute achieves perfect routing while minimizing routing tables sizes and enabling subscription cancellation. Our main focus is to show that XRoute effectively succeeds in minimizing routing tables sizes. Indeed, keeping routing tables small is essential to implement efficient filtering of messages against large number of subscriptions: as the filtering speed typically decreases linearly with the number of subscriptions (whether matching subscriptions sequentially, or using sophisticated algorithms as in [35]), small routing tables can dramatically improve the overall performance of a content network in terms of routing, where filtering plays a major part. As a consequence, we first analyze the sizes of the routing tables in our system when imposing different consumer loads. We then study the performance of the simulated network in terms of routing.

Note that we do not evaluate the performance of the XTrie algorithm here. A complete study has been done in [35].

We then evaluate experimentally the XSearch algorithm, which plays a major role in the XNet system by enabling it to handle large and dynamic consumer populations. We then assess the performance of our XNet system in terms of subscription management, when deployed on a simulated network.

Finally, we analyze the efficiency of our techniques in a large scale experimental deployment on the PlanetLab testbed [97]. We show that XNet does not only offer good performance and scalability with large consumer populations under normal operation, but can also quickly recover from system failures.

6.2.2 Parameters of the data used in the experiments

In this section, we present the parameters of the data that we used in all the experiments that we performed.

To evaluate the behavior of our XNet system or to assess the performance of a particular component or technique, we had to generate consumer subscriptions and producer events. As previously mentioned, subscriptions in our system are expressed using the XPath [118] language, and producer events are XML documents.

To generate a set of XPath expressions, we have developed an XPath generator that takes a Document Type Descriptor (DTD) as input and creates a set of valid XPath expressions based on a set of parameters that are described in table 6.1.

Parameter	Name	Description
h	maximum depth	maximum depth of a leaf in the expression
d	ellipsis (//) maximum depth	maximum depth that an element with label // can have
$p_{//}$	ellipsis probability	probability that an element has label //
p_*	wildcard (∗) probability	probability that an element has label ∗
p_λ	branching probability	probability to have a new child
m	minimum depth	minimum depth of a leaf in the expression
θ_S	element skew	skew of Zipf distribution for element names
x	duplicates probability	probability to have duplicate expressions. $x = 1$ means no duplicates.

Table 6.1: Parameters of XPath subscriptions.

We have used the NITF (News Industry Text Format) DTD [42] to generate our sets of XPath expressions. The NITF DTD, which was developed as a joint standard by news organizations and vendors worldwide, is supported by most of the world's major news agencies and is used in several commercial applications. It contains 123 elements with 513 attributes (as of version 2.5). Note that the results of all experiments can easily be generalized to multiple DTDs. Indeed, as DTDs generally use distinct grammars, an XML document valid for a given DTD is unlikely to match a subscription for another DTD; thus, using multiple DTD essentially boils down to running separate experiments with each DTD and combining the results.

To generate XML documents, we used IBM's XML Generator tool [51], with the parameters described in table 6.2.

Parameter	Name	Description
L	document length	maximum length of XML document
T	document size	maximum number of tag pairs in document
r	r	maximum number of repeatitions of children with ∗ or + option
θ_D	element skew	skew of Zipf distribution for element names

Table 6.2: Parameters of XML documents.

6.2.3 Efficiency of the XRoute protocol

Simulation Setup

To assess the efficiency of the aggregation techniques used by our XRoute protocol, we have evaluated it on a simulated network. We have generated a network topology using the transit-stub model of the Georgia Tech Internetwork Topology Models package [125]. The resulting network topology, shown in Figure 6.8, contains 64 routers. We then added 24 consumers at the edges of the network and a single producer.

We have simulated consumer load by registering subscriptions at the consumer nodes. The subscriptions were generated with the values of the parameters as indicated in table 6.3. We have generated sets of subscriptions of various sizes (from 1000 to 200,000 subscriptions).

Parameter	Value
h	10
d	3
$p_{//}$	0.1
p_*	0.1
p_λ	0.1
m	3
θ_S	−1
x	0 or 1
N	1000 → 200000 (number of generated subscriptions)

Table 6.3: Parameter values of XPath subscriptions.

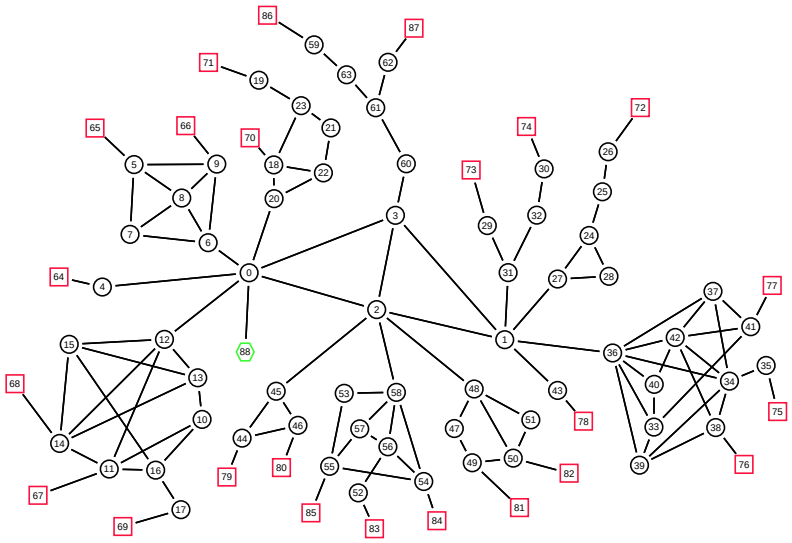


Figure 6.8: Simulated network topology with 64 routers (circles), 24 consumers (boxes), and 1 producer (hexagon).

The events published by the producer are XML NITF documents generated with the values of the parameters as indicated in table 6.4.

Parameter	**Value**
L	20
T	$\{22, 58, 108\}$ tag pairs
r	3
θ_D	0 (uniform)

Table 6.4: Parameter values of XML documents.

To handle consumer subscriptions, we implemented two routing protocols in the routers of our network. First, we implemented a *Simple* routing protocol that does not use subscription aggregation, except for suppressing multiple occurrences of a subscription. With that protocol, the size of the routing table at a node is equal to the number of disctinct subscriptions that consumers registered downstream. We will refer to this protocol as *Simple*. Second, our XROUTE routing protocol that makes extensive use of subscription aggregation to minimize the size of the routing tables. Note that appart from not using subscription aggregation, the *Simple* protocol works in a similar way as XROUTE (in particular, the subscription advertisement scheme).

To route events efficiently, we implemented the XTRIE algorithm in the routers of our network, as in Section 3.3. Events are routed as explained in Section 4.1.1.

To simulate a population of N registering consumers, we proceeded as follows. We generated a set of N random subscriptions as described earlier in the section. Note that the set contains possibly multiple occurences of each subscription. For each subscription in the set, we registered a random number of occurences (uniformly between 0 and 100) of that subscription at a random consumer node (uniformly). At that consumer node, the registration was handled using either the *Simple* protocol or our XROUTE protocol, the information was then propagated in the network and the routing tables were updated as described in Section 4.2.

We have thus simulated two systems to test the efficiency of the aggregation techniques that our protocol XROUTE uses. The first system is our XNET system in which the routers implement the XROUTE routing protocol. The second system which we will refer to as the *Simple* system is the same as XNET but in which the routers implement the *Simple* protocol instead of XROUTE.

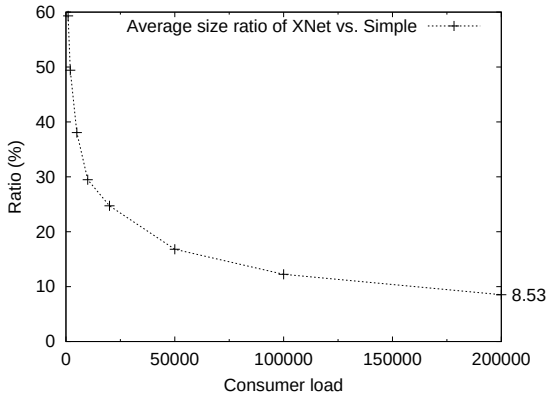


Figure 6.9: Ratio of the average routing table sizes in the XNet and the Simple system.

Consumer Population	Mean μ		Variance σ	
	μ_{XNet}	μ_{Simple}	σ_{XNet}	σ_{Simple}
1000	34	58	31	74
2000	52	105	44	130
5000	86	225	67	273
10000	120	408	88	496
20000	176	714	122	876
50000	258	1533	162	1899
100000	334	2729	206	3402
200000	415	4862	232	6113

Table 6.5: Mean value and standard deviation of the routing table sizes in the XNET and the *Simple* systems.

Routing tables sizes

For each system which was imposed a given consumer load, we computed the average size of the routing tables that we obtained, as well as the standard deviation. We then computed the ratio of the mean routing table size that we obtained in our XNET system by that obtained in the *Simple* system. The evolution of that ratio when varying the consumer load is illustrated in figure 6.9. Table 6.5 shows the evolution of the mean value μ and standard deviation σ of the routing table sizes (in number of entries) in each system when varying the consumer load.

Note that the routing tables of the consumer nodes in the network were excluded from the experiments. Indeed, we have seen in Section 4.2.5 that subscription representations are not permitted in the routing tables of the consumer nodes (but they *are* substituted). This is required by the XROUTE protocol to maintain perfect routing in case of subscription cancellation. As a consequence, to assess the efficiency of the aggregation techniques used by XROUTE, we excluded the routing tables of consumer nodes. Their sizes, wether obtained with the XROUTE or *Simple* protocol, are the same, and are approximately bounded by $\frac{N}{24}$ (since subscriptions were injected uniformely at random at the 24 consumer nodes). Note that this is an upper bound, since there are typically multiple instances of the same subscription.

Figure 6.9 and table 6.5 show that our protocol XROUTE implemented in our XNET system reduces the average routing table size significantly when compared to the *Simple* protocol. For example, for a consumer population of 200000, our protocol XROUTE yields to an average routing table size of 415 entries, when compared to 4862 for the *Simple* protocol (12 times larger). Moreover, the gap between both protocols widens significantly with large number of consumers which shows that XROUTE is more and more efficient in reducing routing table sizes. This is directly due to the fact that, with an increased number of subscriptions, the protocol is able to find more containment relationships between them, and hence perform subscription aggregation more efficienctly.

Table 6.5 shows that our protocol XROUTE yields to a routing table size variance that remains

small even with large consumer populations. The variance of the *Simple* protocol is several times larger and the gap between the two protocols gets more and more important with large populations. As a consequence, XROUTE yields to much more *homogenous* (in size) routing tables, when compared to the *Simple* protocol. This can be an issue when provisioning the network in a real system.

As a conclusion, the routing tables are both *small* and *homogeneous* in our XNET system, when compared to a system that would implement the *Simple* protocol.

Routing efficiency

In this section, we study the performance of our XNET system and that of the *Simple* system in terms of routing when varying the number of registered consumers and the sizes of the documents published by the producer. As both systems implement the XTRIE algorithm in their routers, comparing the two systems will enable us to study the impact of XROUTE in routing. All the algorithms were implemented in C++ and compiled using GNU C++ version 2.96. Performance experiments were conducted on 1.5 GHz Intel Pentium IV machines with 512 MB of main memory running Linux 2.4.18.

The protocol of the experiment that we conducted is the following. Consider a document of size T. We route this document in each system (XNET and *Simple*) which was previously imposed a consumer load of N consumers. When the document is routed in a system, it is processed by a certain number of routers in the network and it (a copy of the event) reaches a certain number of consumer nodes (recall that both systems implement perfect routing). First, for each router that routed the event, we measure the process time, that is the time that was required by this router to perform its routing task. We then compute the maximum value of the process times among those routers. This measure will be refered to as the maximum process time. Second, for each consumer node that received the event, we measure the routing delay to that node, that is the time to reach that node, from the time it was issued by the producer. Not that we did not simulate any link delays in the network. Hence, the routing delay is entirely due to the process times at the encountered routers. We then compute the average value of the routing delays of the consumer nodes that received the event. This measure will be refered to as the average routing delay. We repeat the experiment 100 times (with the same document size T) and we compute the mean values of the maximum process times and the average routing delays measured consequently. Note that for different documents, the maximum process time is not necessarily obtained for the same router. This is not a problem, since from the point of view of a producer, we are interested in a estimation of the maximum process time, and not in the location of the router. Finally we have an estimation (the mean value) of the max process time and the average routing delay when routing a document of size T in a system which was imposed a consumer load of N consumers. From the maximum process time μ_{max}, we then compute $\frac{1000}{\mu_{max}}$, which is the maximum throughput, in documents per second. This figure represents the maximum rate at which a producer can expect to be allowed to inject documents of size T in a system that comprises N registered consumers. Similarly, considering a consumer in a system that comprises N consumers in total, the average routing delay represents the latency that this consumer can expect to receive a document of interest, of size T. Because of the efficiency of the filtering algorithm XTRIE, we only focused on large consumer populations.

Figure 6.10 shows the evolution of the maximum throughput with the consumer population in each system, when routing documents comprising 22 (small), 58 (medium) and 108 (large) tag-pairs. It shows that for documents of same size, the maximum throughput obtained in our XNET system is much higher than that obtained in the *Simple* system, whatever the consumer population in a system. For example, for a consumer population of $200,000$ consumers, XNET enables the producer to inject documents of size 22 at a maximum rate of 113 documents per second when compared to a rate of 27 documenets per second in the *Simple* system, which is more than four times slower. Moreover, XNET has a very little sensitivity to the consumer population, which shows that it scales very well to large consumer populations. This is true for all document sizes.

Figure 6.11 shows the evolution of the average routing delay with the consumer population in

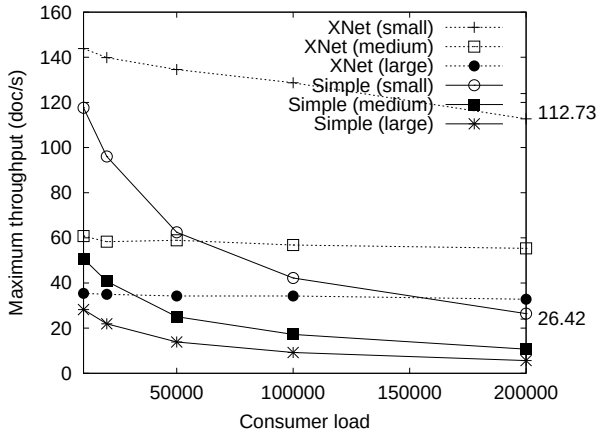


Figure 6.10: Maximum document throughput in the XNet and the Simple system.

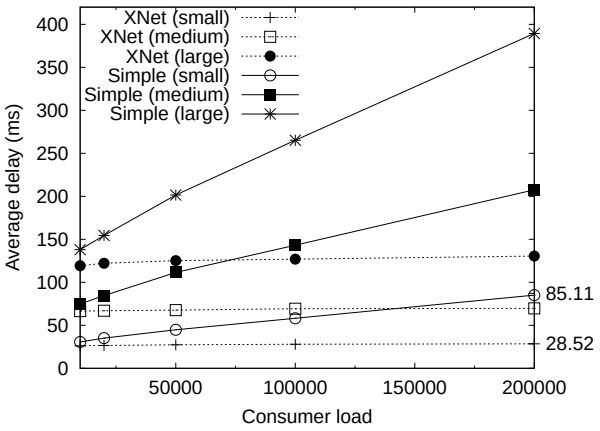


Figure 6.11: Routing delay in the XNet and the Simple system.

each system, when routing a document of a given size. As we have seen, the average delay can be interpreted as the time a consumer can expect to wait before receiving the document which it is interested in. We can see clearly that for a given document size, the delay obtained in our XNet system is several times smaller than that obtained in the *Simple* system, and the gap between the two curves widens dramatically with the number of consumers. For example, for a consumer population of $200,000$ consumers, the average routing delay for a document of size 22 would be $28.52ms$ in our XNet system when compared to $85.11ms$ in the *Simple* system. Moreover XNet is almost not sensitive to the consumer population. This shows that not only does XNet enable clients to receive documents in a much smaller delay than the *Simple* system does, but it also scales extremely well to large consumer populations, whatever the document size.

As a conclusion, our XNet system is scalable in terms of routing. It enables producers to inject messages at a high rate in the network and routes the messages to the consumers in a small amount of time. Those excellent results are due to the efficiency of the XTrie algorithm for one part, and for the other part to the XRoute protocol, which plays a major role by reducing the sizes of the routing tables significantly when compared to a protocol that does not perform subscription aggregation.

6.2.4 Efficiency of the XSearch algorithm

In this section, we evaluate the performance of the XSearch algorithm. XSearch plays a major role in our XNet system by speeding up routing tables updates. More precisely, XSearch efficiently identifies all the possible containment relationships between a given subscription and a possibly large set of subscriptions.

Note that although described in the context of content-based routing and XPath, the XSearch algorithm can be readily applied to similar subscription languages or to address different data management problems.

Experimental Setup

To evaluate the efficiency of the XSearch, we generated XPath tree patterns with the values of the parameters indicated in Table 6.6.

All the algorithms were implemented in C++ and compiled using GNU C++ version 2.96. Experiments were conducted on 1.5 GHz Intel Pentium IV machines with 512 MB of main memory running Linux 2.4.18.

Parameter	Value
h	10
d	3
$p_{//}$	$0.0 \rightarrow 0.20$
p_*	0.1
p_λ	$0.0 \rightarrow 0.20$
m	3
θ_S	-1
x	1

Table 6.6: Parameter values of XPath subscriptions.

XSearch Efficiency

We evaluated the efficiency of the XSEARCH algorithm for search sets of different sizes. For this experiment, we considered search sets with unique subscriptions, that is, a given subscription does not appear more than once in a set. Indeed, in a given router, XSEARCH is used to determine the containment relationships between a given subscription and the subscriptions in the routing table, which are all unique. Also, in this experiment, we considered tree patterns with parameters $p_{//} = 0.05$ and $p_\lambda = 0.10$.

For each search set, we generated $1,000$ additional subscriptions and, for each of them, we measured the time necessary to determine the subset of the subscriptions that contain, and are contained by, that subscription. For comparison purposes, we have also measured the efficiency of the XSEARCH algorithm against sequential execution of the containment algorithm of [34], which we call *Linear*.

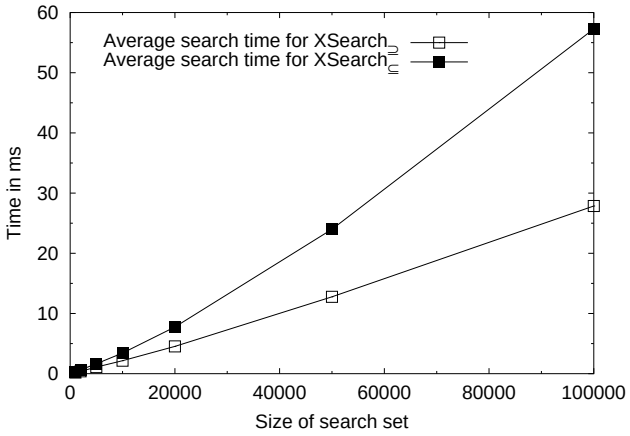


Figure 6.12: Average search time for the XSEARCH algorithm.

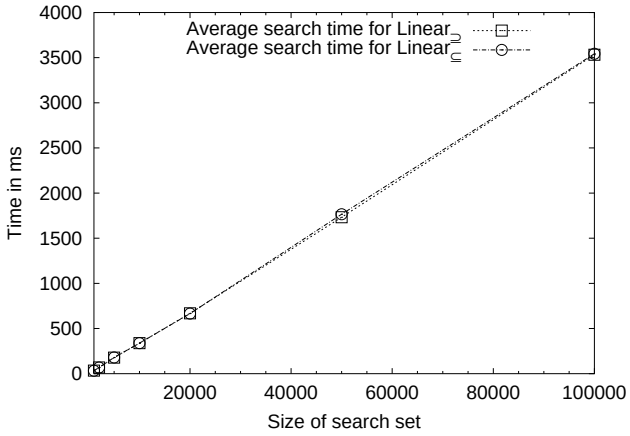


Figure 6.13: Average search time for the *Linear* algorithm.

Figure 6.12 shows the average search time of the XSEARCH algorithm and Figure 6.13 that of the *Linear* algorithm. It appears clearly that XSEARCH is extremely efficient. Even for very large search sets, we can expect an average search time of less than 50 ms. In comparison, the *Linear* algorithm yields to search times that are systematically more than two orders of magnitude higher. This result is not surprising, as the *Linear* algorithm needs to evaluate the entire subscription set R while $XSearch$ only searches through the factorization tree, which is much smaller by construction.

The second variant of the algorithm, $XSearch_{\subseteq}$, is significantly less efficient than $XSearch_{\supseteq}$ for large consumer populations. This is due to the fact that $XSearch_{\subseteq}$ works recursively on the nodes of $T(\mathcal{R})$, trying to find paths in a given subscription S that are contained by the tree patterns in $T(\mathcal{R})$. Hence, the number of traversals of the factorization tree is bounded by its size. On the contrary, $XSearch_{\supseteq}$ works recursively on the nodes of S, trying to locate paths in $T(\mathcal{R})$ that are contained by S. The number of traversals of the factorization tree is bounded by the size of S, which is most often much smaller than $T(\mathcal{R})$.

Size of search set	1,000	2,000	5,000	10,000
XSEARCH$_{\supseteq}$	0.23	0.45	1.17	2.41
XSEARCH$_{\subseteq}$	0.28	0.53	1.30	2.57

Table 6.7: Average search time of XSEARCH in ms.

We have seen in Section 6.2.3 that the sizes of the routing tables rarely exceed 1,000 entries, even for very large consumer populations, thanks to subscriptions aggregation. For completeness, we show in Table 6.7 the absolute average search time of XSEARCH for search sets of small sizes, which are most relevant in the context of content-based routing.

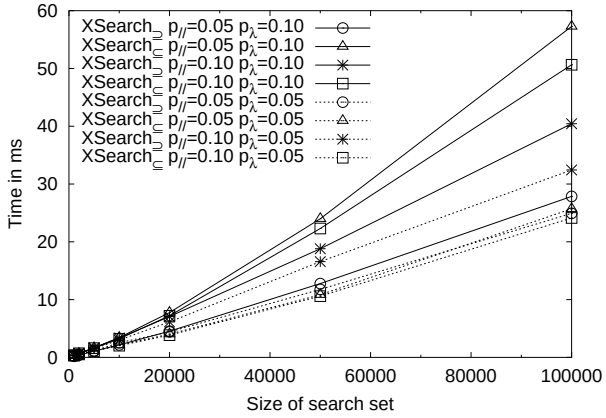


Figure 6.14: Average search time for the XSEARCH algorithm, for different values of $p_{//}$ and p_λ

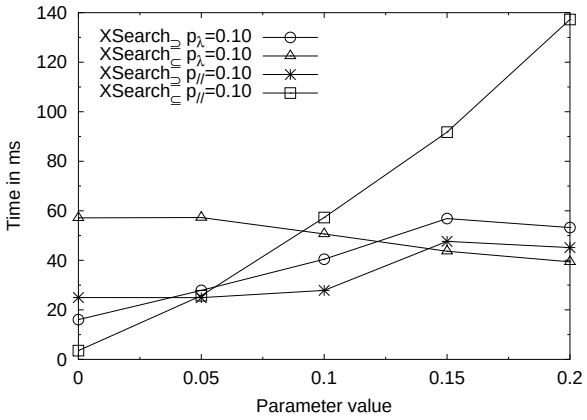


Figure 6.15: Evolution of the average search time of the XSEARCH algorithm when varying $p_{//}$ and p_λ, for a search set of 100000 patterns.

Influence of patterns parameters

In this experiment, we study the influence of parameters $p_{//}$ and p_λ on the efficency of the algorithms. We proceeded as in the previous experiment but we varied the values of parameters $p_{//}$ and p_λ in the following ranges:

- $p_{//} \in \{0.0; 0.20\}$
- $p_\lambda \in \{0.0; 0.20\}$

by steps of 0.05. Results are illustrated in Figures 6.14 and 6.15.

Figure 6.14 shows the evolution of the search time for both algorithms when varying the size of the search set and for different values of $p_{//}$ and p_λ. Figure 6.15 also shows the evolution of the search time but when varying $p_{//}$ and p_λ. The size of the search set was fixed to 100,000 patterns. We then fixed p_λ to a value of 0.10 and studied the evolution of the search time when varying $p_{//}$. We finally fixed $p_{//}$ to a value of 0.05 to study the impact of p_λ on the search time.

Influence of $p_{//}$. We observe that parameter $p_{//}$ has an opposite influence on $XSearch_{\subseteq}$ and $XSearch_{\supseteq}$, whatever the value of parameter p_λ. Indeed, a higher value of $p_{//}$ reduces the efficiency of $XSearch_{\subseteq}$ whereas it increases that of $XSearch_{\supseteq}$. This is directly due to the way the two algorithms operate when a node with label // is encountered. The explanation is the following.

On the one hand, nodes with label // may be beneficial for both algorithms in that they can reduce the number of algorithmic calls or even allow the algorithm to terminate. Indeed, for $XSearch_{\subseteq}$, consider a node u in S and t in $T(\mathcal{R})$, where $XSearch_{\subseteq}(t, u)$ is called, and such that $label(u) \neq //$. If there are more nodes t' in $child(t)$ with label //, then fewer instances of the algorithm will be

called, since only t' such that $label(t') \preceq label(u)$ are considered (line 8 in Algorithm 11). Hence, more nodes with label // in $T(\mathcal{R})$ implies fewer calls of the algorithm. For $XSearch_{\supseteq}$, the same phenomenon occurs when considering nodes in $T(\mathcal{R})$ instead of nodes in S and vice versa.

On the other hand, nodes with label // may adversely impact the efficiency of the algorithms by increasing the number of algorithmic calls. For $XSearch_{\subseteq}$, this happens when a node in S has label //. Then, $XSearch_{\subseteq}$ tries to match it to paths of any length in $T(\mathcal{R})$, until a leaf is found (in $T(\mathcal{R})$). Similarly, for $XSearch_{\supseteq}$, when a node in $T(\mathcal{R})$ is encountered with label //, $XSearch_{\supseteq}$ tries to match it to paths of any length in S, until a leaf is found (in S). As a consequence, a higher value of $p_{//}$ means a higher probability that a node has label // in S and $T(\mathcal{R})$, and hence more algorithmic calls.

The major difference between $XSearch_{\subseteq}$ and $XSearch_{\supseteq}$ is that $XSearch_{\supseteq}$ tries to map the node with label // to paths in $T(\mathcal{R})$ whereas $XSearch_{\subseteq}$ tries to map it to paths in s. On average, an individual tree pattern s is much smaller than $T(\mathcal{R})$, and a leaf is encountered much more rapidly. As a consequence, $XSearch_{\supseteq}$ benefits from a higher value of $p_{//}$ whereas $XSearch_{\subseteq}$ suffers from it (although for higher values it also seems to benefit from it).

Influence of p_λ. We first observe that a lower value of p_λ is beneficial for both algorithms. This is directly due to the fact that the more the number of children the nodes have, the more the number of algorithmic calls. In addition to that, we observe that a lower value of p_λ is much more beneficial for $XSearch_{\supseteq}$ than for $XSearch_{\subseteq}$. This can be directly explained by the fact that $XSearch_{\supseteq}$ works recursively on the nodes of $T(\mathcal{R})$ whereas $XSearch_{\subseteq}$ works recursively on the nodes of s, and $T(\mathcal{R})$ benefits from the reduction of the number of children in the tree patterns to a much larger extent than a single tree pattern s does.

$\lvert R\rvert$	1,000	2,000	5,000	10,000	20,000	50,000	100,000
$\sum_{s_i \in R} \lvert s_i\rvert$	7.6	15.8	42.1	88.1	183.3	481.8	998.6
$\lvert T(R)\rvert$	1.9	3.6	8.2	15.1	28.1	62.1	112.6

Table 6.8: Space requirements for a given subscription population R and its factorization tree $T(\mathcal{R})$, in thousands of nodes.

Space Efficiency

We have experimentally quantified the space requirements of the factorization tree with subscription sets of various sizes. For this experiment, $p_{//}$ and p_λ had fixed values of 0.05 and 0.1, respectively. The results in Table 6.8 confirm that the number of nodes in the factorization tree is indeed notably smaller than the sum of the nodes of the individual subscriptions.

Subscription management

XSearch is a vital component of our XNet system. By efficiently determining the containment relationships between subscriptions, it enables XNet to handle large and dynamic consumer populations. In this section, we specifically evaluate the performance of subscription management in XNet when deployed on the simulated network of Figure 6.8.

We proceeded as in Section 6.2.3 to simulate a population of N registered consumers. The parameters of the XPath subscriptions are the same as in table 6.6. The XRoute protocol was used to handle consumer subscriptions.

Consider a network with N registered consumers. We are interested in the average delay that a new consumer can expect when registering or cancelling a subscription (a given number of instances of that subscription) so that the whole system is updated and the consumer can receive (or stop receiving) the events that it is interested in. In other words, we are studying the delay that a new consumer can expect when *subscribing* to the system when N consumers have already registered.

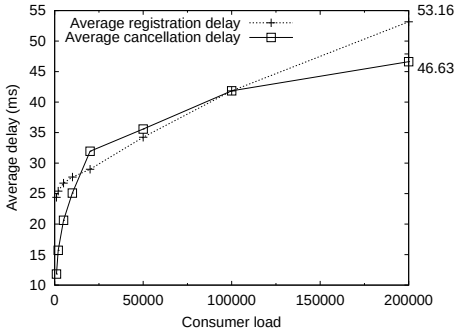


Figure 6.16: Average delay for subscriptions handling.

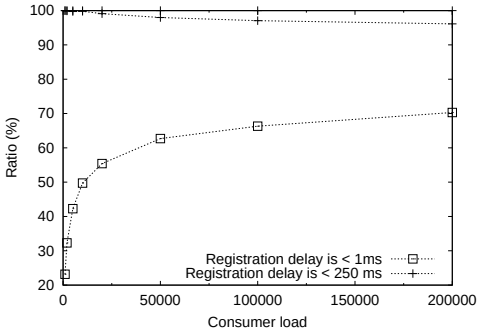


Figure 6.17: Distribution of registration delays.

The protocol of the experiment is as follows. Given a prepopulated system with N consumers, we generated $10,000$ random subscriptions (which may contain duplicates to model distinct consumers having the same interests) and injected each of them in turn at a consumer node chosen uniformely at random. After injecting a subscription, we canceled it to maintain a stable consumer population during the whole experiment. We measured for each registration the time necessary to update all the routing tables, and we computed the mean value. To study the cost of subscription cancellations, we proceeded similarly except that, for each of the $10,000$ measurements, we totally canceled a random subscription (partial cancelations are similar to *duplicate* registrations). Results are shown in Figure 6.16. As in Section 6.2.3, we did not include any link delays in this experiment.

Case of registrations. Figure 6.16 shows that the average registration delay increases with the size of the consumer population, at a decreasing rate. However, even for very large consumer populations, the average delay for a new registration remains reasonably small (53ms for $200,000$ registered consumers).

To explain this, let us consider the registration of a subscription s. We have seen in section 4.2.5 that *supplementary* and *new* registrations imply costly routing table updates, due to the fact that they require that some containment relationships between subscriptions be established. As a consequence, the higher the number of registered consumers, the larger the routing tables, and thus the slower the update of the routing tables of the routers which see the registration of s as a *supplementary* or a *new* registration. Hence the registration delay increases with the consumer load.

On the other hand, the higher the number of registered consumers, the more chances that s has already been registered previously, or has been *substituted* by another subscription. It then follows from section 4.2.5 that the higher the number of registered consumers, the higher the number of routers that see the registration of s as a *duplicate* registration. The update of the routing table of such routers is very fast.

As a consequence, by increasing the number of registered consumers, we increase the number of routers which see a *duplicate registration* but at the same time we increase substantially the processing time of the routers for which the registration of s is a *supplementary* or a *new* registration.

This behavior is illustrated in figure 6.17, where we plotted the cumulative percentage of the registrations (out of the $10,000$ registrations) that have a registration delay lower than $1ms$ and than $250ms$, for a given population size. This can be seen as an estimation of the probability (in percentage), when registering a subscription, to have a delay smaller than $250ms$ and $1ms$. We can see that for large consumer populations, it is likely that the registration delay is very low, but there is also a little chance that it is much higher. For smaller consumer populations, the registration delay is likely to be around an intermediate value.

Case of cancellations. Figure 6.16 shows that the average cancellation delay increases sharply at the beginning but then smooths down when we have reached a certain number of registered consumers. In any cases, the delay remains relatively small even for very large consumer populations

(less than $50ms$ for $200,000$ registered consumers).

To explain this evolution, suppose that we are handling the cancellation of subscription s. We have seen in Section 4.2.5 that if some subscriptions were aggregated to s at a router, we may need to "reinsert" them in the routing table, which requires to compute a potentially high number of containment relationships. Since the chances that subscriptions are aggregated to s increases with the consumer population, the cancellation delay increases as well.

On the other hand, the chances that s has been substituted by another subscription on a router (along the way to the producer node) increases with the consumer population. As we have seen in Section 4.2.5, at the routers upstream the cancellation of subscription s is seen as a *duplicate registration*, and the updates are very fast.

Also, the higher the consumer population, the more chances that subscription s has been registered at several different consumer nodes. Hence, the more the chances that the cancelation of subscription s is not total at a given router. As we have seen in Section 4.2.5, the routing table updates are then very fast at the upstream nodes. As a consequence, the average cancelation delay increases at a rate that shrinks with the consumer population, and remains small even for very large populations ($46ms$ for $200,000$ consumers).

As a conclusion, our $XNet$ system deployed on a simulated network seems to handle both subscription registrations and cancellations efficiently even for very large consumer populations. The excellent scalability of the system is mainly due to the efficiency of the XSearch algorithm, which was specifically designed to handle the most costly operation of routing tables update, that is the determination of containment relationships between subscriptions.

6.2.5 Large scale experimental deployment on the PlanetLab testbed

To assess the performance of our XNet system in a real distributed environment, we deployed application-level routers on the PlanetLab global distributed platform [97] to simulate a realistic content based network overlay at Internet scale.

Experimental setup

Network topology. The network topology consists of 21 machines of the PlanetLab network, an open distributed platform for developing, deploying, and accessing planetary-scale network services. PlanetLab was the testbed of choice for us, as it enabled us to experiment with the real conditions of the Internet, especially its unpredictability. Although we had only 22 nodes in our overlay, results are representative of larger networks: As a router only knows its direct neighbors, scalability does not directly depends on the number of routers, but on the consumer population. The machines used in the experiments were running a customized version of Linux. They all had at least 512 MB of memory and a 1.2 GHz processor, but they were used concurrently by several users running similar experiments and their load was very uneven. In practice, as the processing and memory requirements of XNet are moderate, application-layer routers can be easily deployed on low-end machines with limited resources. Each of the 21 PlanetLab machine was hosting a router. As illustrated in Figure 6.18, 12 of the routers are consumer nodes (boxes), 1 is a producer node (hexagon), and the remaining 9 are routing nodes (circles). The extension of the country where the machine is located is indicated under the node numbers and the average measured link delays are indicated next to every link (upstream delay above, downstream delay below). The routers are organized in a spanning tree rooted at the producer. Each node implements the protocols of our XNet system, that is: the XRoute routing protocol, the XSearch subscription management protocol, and the XTrie filtering algorithm.

Overlay statistics. Table 6.9 provides some network statistics about our experimental overlay. All measures are averages over several runs executed at different times. The link delay was measured

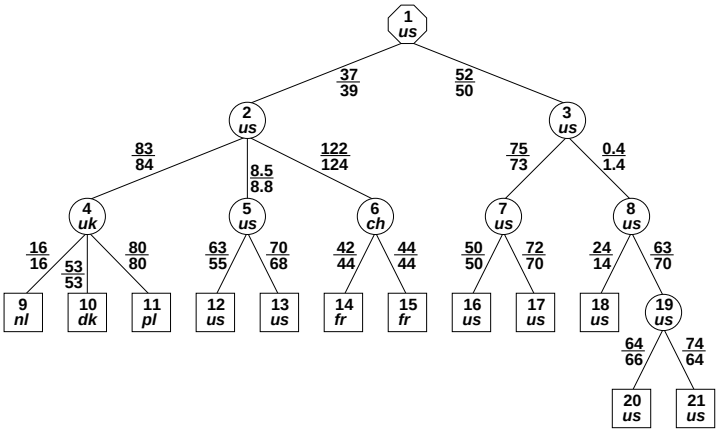


Figure 6.18: Experimental network topology.

as the round-trip time to send a packet to a machine and receive a reply over TCP (it does not include the TCP connection establishment time as we are using persistent connections). Note that we excluded the link delays between node 3 and node 8 for the computation of the minimal link delay, as those nodes are located in the same LAN.

Metrics	**Value**
Average link delay	54.135 ms
Standard deviation of link delays	28.18 ms
Maximal link delay	6 → 2: 123.67 ms
Minimal link delay	2 → 5: 8.47 ms
Average minimal routing delay	160.63 ms
Average minimal update delay (consumer → producer)	169.64 ms
Maximal producer throughput ("single-element" docs)	53.13 docs/s
Maximal producer throughput ("normal-size" docs)	30.28 docs/s
Maximal upstream (consumer) throughput	18.56 sub/s

Table 6.9: Overlay statistics.

The average minimal routing delay was computed by injecting at the producer an XML document with a single "wildcard" element matching all consumer subscriptions. Consequently, the document was forwarded to all the consumers with minimal process time at the routers. We measured the delay experienced by each consumer to receive the document and we computed the average over all consumers and over $1,000$ runs. This measure gives a lower bound on the routing delay.

We computed the average minimal update delay as the time necessary to propagate a "wildcard" subscription (requiring negligible process time at the routers) from the consumer to the producer. We computed the average over 100 runs at each consumer and over all consumers. The resulting value gives a lower bound of the update time of the network when a new consumer subscribes to the system.

The maximal producer throughput was computed by sending a burst of $1,000$ documents and measuring the delay between the time the first document was sent until the last document was received by the last consumer. We ran the test both with minimal "single-element" documents and with "normal-size" documents containing 22 tag pairs. The first measure corresponds to the maximal network throughput at the producer, while the second gives an upper bound of the producer rate with realistic event workloads.

We finally computed the average upstream throughput in the same way as we did for the maximal producer throughput: for each consumer, we registered 100 "single-element" subscriptions in a burst and measured the delay until the network has been updated. We then computed the average over all the consumers. This value gives an upper bound of the consumers' arrival rate.

Parameter	Value
h	10
d	3
$p_{//}$	0.1
p_*	0.1
p_λ	0.1
m	3
θ_S	-1
x	0 or 1
L	20
T	{22} tag pairs
r	3
θ_D	0 (uniform)
Documents arrival rate	Poisson with rate $\lambda_{doc} = 1/\text{s}$
Consumers arrival rate	Poisson with rate $\lambda_{sub} = 1/\text{s}$
Consumer population	$P = 1,000$ to $50,000$
Crash duration	$D = 1$ to 10 min
Faulty router	2 and 19; 8
Backup routers	3; 2 and 7

Table 6.10: Parameters of the experiments.

Parameters of the experiments. The parameters of our experiments are summarized in Table 6.10. Consumer subscriptions and producer events were generated as in the previous experiments, with the indicated parameter values. For the sake of simplicity, we assume that each consumer registers only one subscription: a consumer with two subscriptions is considered as two distinct consumers. The parameters λ_{doc} and λ_{sub} control the arrival rate of documents and consumers, respectively. P defines the size of the existing consumer population, i.e., the number of consumers that are registered in the system when the experiment starts. D controls the duration of a failure before recovery. Finally, we have simulated the failure of various routing nodes of the network and experimented with several configurations of backup routers.

Performance Under Normal Operation

We first evaluate the efficiency of our XNet system under normal operation, that is, with no system failures.

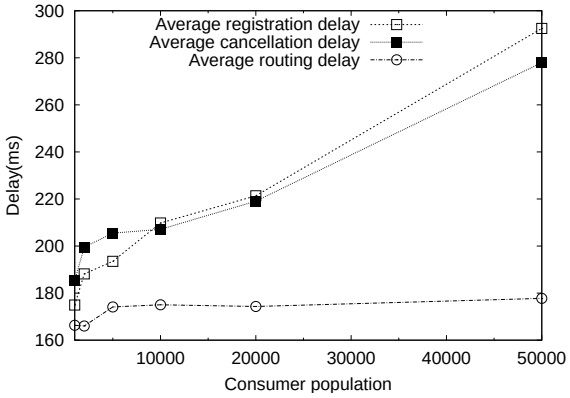


Figure 6.19: Routing/subscription delay.

Routing delay. We are interested in measuring the average routing delay, that is, the average time taken by an event to traverse the network and reach all the consumers that are interested in that event. The protocol of the experiment is the same as in Section 6.2.3: we first populate the network with random subscriptions injected at arbitrary consumer nodes until the consumer

population reaches P. We then inject events at the producer node at rate λ_{doc}. For each event, we compute the average routing delay (i.e., producer-to-consumer latency) that was experienced by each consumer node that received the event. Results are average values of $1,000$ runs and are shown in Figure 6.19. We can see that the routing delay remains small (less than 180 ms) even with large consumer populations. In fact, the routing delay is very close to the measured minimal routing delay (Table 6.9), which indicates that the delay is essentially due to the link delays and not the processing time at the routers. As previously mentionned in Section 6.2.3, the excellent scalability of the system is mainly due to the high efficiency of the filtering algorithm XTrie, combined with the efficiency of the aggregation techniques used in XRoute, which enable to minimize routing tables sizes (and hence the size of the data processed by XTrie).

Registration and cancellation delays. To assess the performance of subscription management, we proceeded as in Section 6.2.4: we measured the average delay experienced by a new consumer registering a subscription, given a preexisting population of a given size. This delay corresponds to the time necessary to update all the routers that are affected by the subscription. Given a pre-populated system with P consumers, we generated $1,000$ random subscriptions (which may contain duplicates to model distinct consumers having the same interests) and injected each of them in turn at a consumer node chosen uniformly at random, at a rate of λ_{sub}. After injecting a subscription, we canceled it to maintain a stable consumer population during the whole experiment. We measured for each registration the time necessary to update all the routing tables, and we computed the mean value. To study the cost of subscription cancellations, we proceeded similarly except that, for each of the $1,000$ measurements, we canceled a random subscription. Results are shown in Figure 6.19.

We observe that the average delay for registering or canceling a subscription increases with the size of the consumer population, but at a moderate rate. Even for large consumer populations, the average delay for a new registration or cancellation remains reasonably small (less than 300 ms). The measured minimal update delay of 170 ms (Table 6.9) indicates that link delays represent more than 75% of the overall registration or cancellation delay for the considered consumer population sizes. We also observe that the slope of the two curves decreases with the consumer population. The evolutions of the registration and cancellation delays are similar to those obtained in Section 6.2.4, and can be explained in a similar manner.

Performance of the *Crash/Recover* Scheme

Under normal operation (with no system failures), we have observed that our XNet system is efficient and scalable. We now study its behavior when faults occur. We first concentrate on the *Crash/Recover* scheme.

Consider a router R that has crashed at time t_{crash} and recovered at time $t_{recovery}$. We want to measure the recovery delay $D_{recovery}$ until the *whole system* has recovered. Indeed, during the downtime of router R, its downstream neighbors buffer the advertisements (consumer registrations or cancellations) that should be sent to R. Upon recovery, R and its upstream routers must "catch up" by handling all buffered advertisements. The recovery delay is computed as the delay between the recovery time of R ($t_{recovery}$) and the time when the whole system has been updated and reflects the current consumer population.

The protocol of the experiment is the following: considering the system with a preexisting consumer population P and under a consumer arrival rate of λ_{sub}, we kill router R_i at time t_{crash} and restart it at $t_{recovery}$. We then measure the delay $D_{recovery}$ until the system is up-to-date with no advertisement in the buffers. We are particularly interested in the ratio between $D_{recovery}$ and the crash duration $D_{crash} = t_{recovery} - t_{crash}$.

We first experimented with the failure of router 2 under various consumer populations and crash durations. We then repeated the same experiments with the failure of router 19. We chose these routers to figure out if the level of the router in the tree topology has an impact on the efficiency of the recovery mechanism.

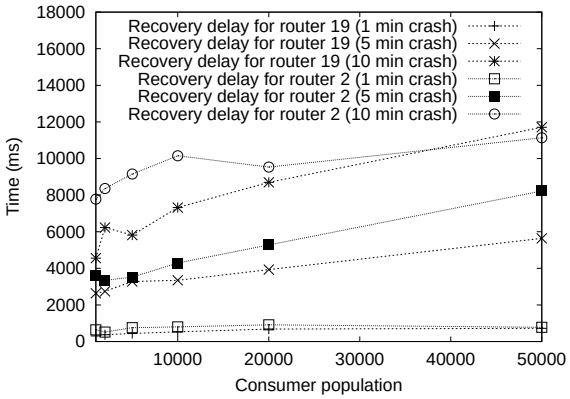


Figure 6.20: Recovery delays for routers 2 and 19 after crashes of various durations.

Figure 6.20 shows the recovery delays resulting from the crashes of routers 2 and 19, for various crash durations and consumer populations. Table 6.11 presents the absolute values, in seconds, of the recovery delay D_2 and D_{19} of routers 2 and 19, respectively, as well as the ratios R_2 and R_{19} of the recovery delay to the crash duration ($R_i = \frac{D_i}{D_{crash}}$).

A first observation is that, independently of the failing router, the crash duration, or the existing consumer population, the system is able to recover in a few seconds (typically less than 10 seconds). We can also note that, unsurprisingly, the recovery delay increases with the crash duration because the system needs to process more buffered advertisements to catch up; it does not, however, exceeds 3% of the crash duration. The recovery delay also increases with the consumer population. This is consistent with the observations made in the failure-free experiments. Finally, we observe that there is no significant difference between the recovery delay for router 2 and that for router 19. This can be explained by the fact that router 2 is a high level router and must process more buffered advertisements, but the updates of its routing table are simpler because subscriptions have likely already been aggregated along the way. Router 19 is a low level router and must process fewer advertisements, but it systematically needs to perform more costly aggregation operations (its downstream routers are consumer nodes and hence do not aggregate subscriptions). Therefore, it appears that the distance of the failing router from the producer node does not have a strong impact on the recovery efficiency of the system.

D_{crash}	P	1,000	2,000	5,000	10,000	20,000	50,000
1 min	D_2	637	514	757	799	913	778
	R_2	1.06	.85	1.26	1.33	1.52	1.29
5 min	D_2	3582	3351	3515	4290	5280	8228
	R_2	1.19	1.11	1.17	1.43	1.76	2.74
10 min	D_2	7781	8361	9153	10148	9533	11136
	R_2	1.29	1.39	1.52	1.69	1.59	1.86
1 min	D_{19}	305	374	440	530	681	721
	R_{19}	0.51	0.62	0.73	0.88	1.13	1.20
5 min	D_{19}	2638	2743	3282	3345	3932	5638
	R_{19}	0.88	0.91	1.09	1.12	1.31	1.88
10 min	D_{19}	4559	6228	5818	7322	8700	11709
	R_{19}	0.76	1.03	0.97	1.22	1.45	1.95

Table 6.11: Recovery delay as function of the consumer population and the crash duration.

Performance of the *Crash/Failover* Scheme

We finally study the overhead induced by the *Crash/Failover* scheme upon the failure of router 8, which represents a medium level router in the tree topology. We considered two different scenarios for the reconnection of the downstream routers 18 and 19 to their backup routers. In the first scenario,

the backup router for both routers 18 and 19 is router 3 (i.e., the closest non-failed upstream router). In the second scenario, router 18 is redirected to router 7 while router 19 is redirected to router 2. Figure 6.21 shows the new network topologies resulting from both scenarios.

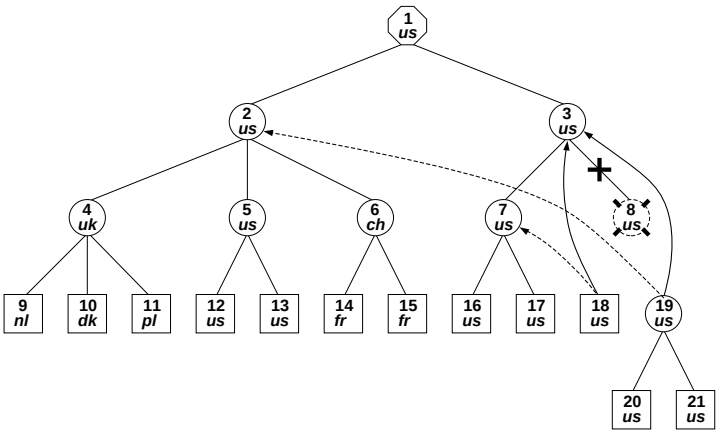


Figure 6.21: New network topologies for scenarios 1 (plain arrows) and 2 (dashed arrows).

The protocol of the experiment is the same for both scenarios. We first kill router 8 at time t_{crash}. We then redirect the downstream routers 18 and 19 to their backup routers, as explained in section 6.1.3. We measure the time $t_{recover}$ when the whole system has been updated and reflects the current consumer population. The recovery delay $D_{recovery}$ is the difference between $t_{recover}$ and t_{crash}. For each scenario, we experimented with preexisting consumer population of various sizes. Also, all the experiments were conducted under a constant consumer arrival rate λ_{sub}. Figure 6.22 summarizes the results that we obtained.

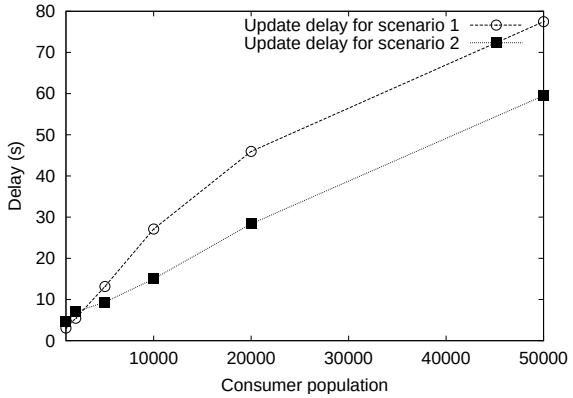


Figure 6.22: Update time for scenarios 1 and 2.

We observe that the recovery delay for both scenarios remains reasonably small, typically less than 1 minute. Also, we can see that the delay increases with the consumer population. This is explained by the fact that the routing tables grow with the consumer population and, during the recovery phase, a portion of the routing tables of routers 18 and 19 must be registered and a portion of that of router 8 must be canceled. Finally, we observe that the recovery delay for scenario 1 is significantly higher than that for scenario 2 for most consumer populations. This is due to the contention on router 3 and its upstream links, which become bottlenecks in scenario 1; in contrast, the load is split between distinct routers in scenario 2.

Discussion

By comparing the results obtained with the *Crash/Failover* and the *Crash/Recover* schemes, we can conclude that the former should be preferred only for small consumer populations and long crash periods. In systems with large consumer populations, the *Crash/Recover* scheme is more adequate provided that the crashed router eventually recovers. We do not discuss the performance of the *Redundant Paths* strategy as it does not introduce recovery overhead.

Conclusion

As a conclusion, it appears that our XNet system deployed on the Planetlab platform offers very good performance and scalability under normal operation. The routing delay remains small whatever the consumer population, which shows that our system delivers events to all interested consumers efficiently. Also, the average *subscribing* delay remains reasonably small even with large consumer populations, which shows that XNet handles large and dynamic consumer populations efficiently.

In addition, our XNet system implements several fault-tolerance schemes that ensure that the shared state of the system remains consistent with the actual consumer population. XNet can quickly recover from router or link failures by using the most appropriate scheme depending on various factors, such as the expected duration of the outage or application specific availability requirements.

Related Work

In this section, we survey the related work. We first present the most popular existing pub/sub systems that are implemented in overlay networks of distributed brokers. We investigate several aspects of pub/sub, in particular routing, subscription management, and reliability issues. We then present other works that do not specifically implement pub/sub, but that relate to the issues that were addressed in this chapter.

6.3 Content routing in most popular Publish/Subscribe systems

6.3.1 Gryphon

IBM Gryphon [18, 87, 88, 72, 15, 5, 14] uses a distributed filtering algorithm based on parallel search trees maintained on each of the brokers to efficiently determine where to route the events. Gryphon implements perfect routing and supports subscription registration and cancellations; in fact, registering (canceling) a subscription reduces to inserting (removing) it from the search tree and is thus an efficient operation. However, to maintain and update the parallel search tree, each broker must have a copy of all the subscriptions in the system. As a consequence, this approach may not scale well to large and highly dynamic consumer populations. Also, the authors do not discuss how to update the parallel search trees (and thus ensure reliable delivery) in the case of link failures or router crashes.

6.3.2 Siena

Siena [23, 24, 25, 26, 29, 31, 103, 32, 27, 109] also uses a network of event servers for content-based distribution, and their routing protocol is most similar to ours. Each event server maintains a routing table that holds a subset of the subscriptions, and the associated subscribers and neighbor routers. Messages are matched against each subscription and forwarded along the paths corresponding to matching subscriptions. Also, Siena makes use of subscription aggregation to reduce routing table sizes and subscription advertisements in the system. However, it is not clear wether subscription cancellation affects routing accuracy. In addition, we could not determine the space- and time-efficiency of the protocol, and wheter it can be extended to support more general subscription languages. In a recent paper [30], the authors of Siena introduce a novel routing scheme for content-based networking based on a combination of broadcast and selective routing. Subscription management is simple and efficient. The system handles subscription cancellations by having routers periodically request the routing table of other routers. However, it does not guarantee perfect routing in the sense that consumers may receive messages that they are not interested in. Also, the authors do not explicitly address the issue of fault tolerance in the system.

6.3.3 Jedi

Jedi [44, 22, 21, 45, 43] proposes several variations for routing events among its networked event servers; in particular, with the hierarchical approach, subscriptions are propagated both upward and downward to the children that have matching subscriptions. Subscription management is simple and efficient, but this approach may lead to large routing tables at the root and unnecessary propagation

of events upward the tree. In a recent work [92], the authors discuss how to adapt the behavior of a pub/sub system to dynamic topology reconfiguration. Their work is based on an approach which they term the "strawman approach" [29, 124], and which is similar to the *Crash/Failover* fault-tolerance scheme that we implemented in our XNet system. Indeed, the principle is to carry out the reconfiguration by using exclusively the primitives available in a pub/sub system. In particular, the reconfiguration triggered by a link removal is dealt with using subscription cancellations (as in our *Crash/Failover* scheme). Their approach aims at reducing overhead, notably by minimizing the repropagation of subscription information, while tolerating frequent reconfigurations.

6.3.4 Rebeca

Rebeca [81] is a prototype notification service that incorporates several routing strategies. Its topology is very similar to ours, i.e., a tree of brokers with a single root called the "root router." Rebeca also distinguishes between brokers that have local clients and those that do not. The system implements a self-stabilization algorithm based on subscription leasing. Routing table entries are valid as long as the lease of the corresponding subscription has not expired. This may lead to consumers receiving out-of-interest notifications. Also, this approach requires that consumers regularly renew their leases by resubscribing, making the system potentially unscalable to large consumer populations.

6.3.5 Onyx

Onyx [123] is is an ongoing research project that aims at implementing a large-scale XML dissemination system. Onyx uses the YFiler [49, 50] technology for content-driven routing. The system features the interesting concept of message transformation, where a message is transformed (data is removed or restructured) incrementally in the course of routing according to the users subscriptions encountered. Also, to boost routing, the profile (s̃ubscriptions) population is partitioned based on exclusiveness of data interestes. In Onyx, a single YFilter instance is used at each node to implement the routing table and incremental transformation. The authors do not discuss how subscription management is implemented efficiently in the system. In particular, they do not discuss the cost of routing table construction/update, especially in the case of subscription cancelation. Also, it is not clear whether subscription cancelation has an impact on routing accuracy. Finally, the authors do not address the issue of reliability in their system.

6.4 Other works

6.4.1 Content routing

In [111], the authors propose an approach for content-based routing of XML data in mesh-based overlay networks. They introduce a routing protocol that reassembles data streams sent over redundant paths to tolerate some node or link failures. Their approach provides a high level of availability but it is not clear how reliability is guaranteed during the addition and removal of subscriptions.

In [108], the authors propose to add content-based routers at specific nodes of an IP multicast tree to reduce network bandwidth usage and delivery delays. They propose algorithms for determining the optimal placement of a given number of content routers. The routing protocol merely consists of propagating subscriptions upward the tree, until they reach the producer or are subsumed by other subscriptions. Subscription cancellation is not supported.

In [11], the authors study the phenomenon of notification loss in pub/sub systems, which is defined as a notification (event) not being delivered to an interested consumer even though it was published when the consumer's subscription was duly registered in the system. The phenomenon may occur due to the diffusion delay of events and/or the registration delay of consumers subscriptions. The authors study the notification loss phenomenon by presenting a simulation study of a pub/sub system and an analytical model. The model is based on a formal framework of a distributed

computation, which abstracts a pub/sub system through the diffusion and registration delays. In particular, the authors study the evolution of the notification loss probability (or its complement) with respect to the registration and the diffusion delays.

6.4.2 Containment relationships

The subscription containment and matching techniques that we implemented in our XNet system are related to the widely studied problem of pattern and regular expression matching. There exists several indexing methods to speed up the search of textual data with regular expressions, like the bit-parallel implementation of NFA [10] and suffix trees [122]. In [36], the authors have addressed the reverse indexing problem of retrieving all the regular expressions that match a given string. They propose RE-Tree, an index structure to quickly determine the regular expressions that match a given input string, by focusing the search on only a small fraction of the expressions in the database.

In [115], Tozawa and Hagiya present a containment checking technique for XML schemas, which is based on binary decision diagrams. Little work has been done on the problem of containment checking for tree-structured XPath expressions. In fact, the problem has been shown to be coNP-complete [80]. A sound but non-complete algorithm has been proposed in [34] to determine whether a given tree-structured subscription covers another subscription, but it does not address the problem of covering relationships between large sets of subscriptions.

Part II

Semantic P2P Overlays for Publish/Subscribe Networks

Chapter 7

A P2P approach to Publish/Subscribe

After developing XNet, which is based on "traditional" techniques to implement Publish/Subscribe, we now explore a different and novel approach to building a pub/sub system based on the P2P paradigm.

7.1 Introduction

7.1.1 Motivations

Most existing pub/sub systems suffer from several drawbacks. They are usually based on a fixed infrastructure of reliable brokers, which cannot easily be modified or extended as the population of the producers and consumers evolves. Further, in most traditional pub/sub systems, the routing process is a complex and time-consuming operation. It often requires the maintenance of potentially large routing tables on each router and the execution of elaborate filtering algorithms to match each incoming document against every known subscription. The use of summarization techniques (e.g., subscription aggregation [28, 34]) alleviates those issues, but a the cost of significant control message overhead or a loss of routing accuracy. Finally, in most existing systems, the network topology has no relationships with the subscriptions registered by the consumers. As a consequence, the process of routing an event often involves a large number of routers, some of which may have no interests in the event but only act as forwarders, which may yield to a poor bandwidth usage. In the worst case, the routing process may be barely more efficient than a broadcast (which benefits from a much lower processing overhead).

To address these limitations, we have designed a pub/sub system [38] that follows a radically different approach to content-based networking.

7.1.2 Objectives

Our pub/sub system was designed to achieve several goals. First, the routing process in our system is extremely simple and has very low resource requirements. Second, by organizing peers based on their interests, content distribution is highly efficient as compared to broadcast. Finally, instead of relying on a fixed infrastructure of reliable brokers, our system is organized as a P2P network: join and leave operations, as well as peer failures, are taken care of at the design level with efficient peers management algorithms. We present in this chapter two instantiations of our system that use the same routing protocol but differ by the way peers are organized. Experimental evaluation illustrates the various trade-offs that they offer in terms of efficiency and accuracy.

7.1.3 Overview

Our system is composed of a collection of peers. Each peer is connected with a set of other peers—its neighbors—with which it exchanges messages. Each peer in the system is at the same time a

producer, a consumer, and a router. As a consumer, a peer registers certain interests that specify the types of events that it is willing to receive. As a producer, a peer can publish events to the system. Finally, as routers, peers process events received by some neighbors and forward them to some other neighbors. In addition, peers implement the peers management algorithms that handle the joins/leaves or additional/existing peers.

Our pub/sub system implements an extremely simple routing process that requires almost no routing state to be maintained at the peers. The price to pay for this simplicity is that routing may not be perfectly accurate, in that there may be false positives and/or false negatives. However, that inaccuracy can be greatly reduced by carefully organizing the peers in a hierarchy according to adequate proximity metrics.

We would like to emphasize that we propose a new P2P approach for pub/sub, which relies on a system model that differs significantly from other P2P applications like file sharing. In particular, we assume that peers are well behaved and remain online for reasonably long periods of time, and that rate of message publication is higher than the frequency of peers' arrivals or departures. Our system provides mechanisms for organizing communities of peers that wish to exchange information using the pub/sub paradigm, without reliance on central servers or fixed infrastructures.

We now describe the routing process that is used in our system for the dissemination of information.

7.2 The Routing Process

7.2.1 Protocol

The routing protocol in our system is entirely based on the principle that every peer forwards a message to its neighbors if and only if the message matches its own interests. The routing process starts when a peer P publishes a message m. We initially make the natural assumption that peers publish messages that match their own interests (we can easily relax this assumption, as will be discussed later). Since P is interested in m, it forwards it to all its neighbors. Routing then proceeds trivially as shown in Algorithm 21.

Algorithm 21 Routing protocol

```
1: Receive message m for the first time from neighbor n
2: if m matches interests then
3:    Forward m to all neighbors (except n)
4: end if
```

The intuition of the algorithm is to spread messages within a community that shares similar interests and to stop forwarding them once they reach the community's boundary. We emphasize on the fact that the routing protocol is extremely simple and requires almost no resources from the peers. It consists of a single comparison and message forwarding operation. In addition to that, it requires no routing state to be maintained in the peers in the system. Each peer is only aware of its own interests and the identity of its direct neighbors, *not* their interests.

7.2.2 Accuracy

Clearly, the aforementioned process is not perfectly accurate and may lead to a peer receive a message that it is not interested in—which we call a *false positive*— as well as missing a message that matches its subscriptions—a *false negative*. In other words, our system may deliver out-of-interest messages and may fail to deliver messages of interest. This is obviously due to the fact that a peer is not aware of the interests of its neighbors and forwards messages only based on its own interests. The challenge is thus to organize the peers so as to maximize routing accuracy. It should be noted that false positives are usually benign, because peers can easily filter out irrelevant messages, whereas false negatives can adversely impact application consistency.

7.2.3 Interest-driven Peers Organization

Consider two neighbor peers P_1 and P_2. If P_1 and P_2 have registered close interests, it means that they are interested in similar types of messages. That is, if P_1 is interested in a message, it is likely that P_2 is also interested in it, and vice versa. It follows that neighbor peers should have close interests in order to minimize occurrences of false positives and false negatives in our system. In other words, we must organize peers based on the interests they registered: proximity in terms of neighborhood should reflect the proximity of the peers' interests.

To evaluate the proximity between two registered interests I_1 and I_2, a proximity metric must be used, that is, a function $f(I_1, I_2)$ that indicates how similar I_1 and I_2 are. Unfortunately, defining a good proximity metric is a challenging problem. It very much depends on the target application, on the language used to specify interests, and most of all on the messages being distributed in the system. The problem of interest proximity has been further discussed in [34].

7.3 Organizing Peers according to Containment

7.3.1 Overview

We now describe a hierarchical organization of the peers that yields no false negatives and only a limited amount of false positives. It uses a proximity metric based on the notion of *interest containment*, which we recall in Definition 5. As previously mentioned, the containment relation is transitive and defines a partial order.

Definition 5 (Containment). *Interest* I_1 contains *interest* I_2*, or* $I_1 \supseteq I_2 \Leftrightarrow (\forall$ *message* m*,* m *matches* $I_2 \Rightarrow m$ *matches* $I_1)$

The relation of *interest equivalence*[1] in defined in a similar manner:

Definition 6 (Equivalence). *Interest* I_1 *is* equivalent *to interest* I_2*, or* $I_1 \sim I_2 \Leftrightarrow (I_1 \supseteq I_2 \wedge I_2 \supseteq I_1)$*. That is:* $\forall$ *message* m*,* m *matches* $I_2 \Leftrightarrow m$ *matches* I_1*.*

The containment-based proximity metric, which we refer to as f_c, allows us to compare interests that share containment relationships and is defined as follows. Consider the set of all registered interests $\mathcal{I} = \{I_1, I_2, \cdots, I_n\}$ that contain I. Let $\{I_i, I_j, \cdots, I_m\} \subseteq \mathcal{I}$ be the longest sequence of non-equivalent interests such that $I_i \supseteq I_j \supseteq \cdots \supseteq I_m$. Then,

$$f_c(I, I') = \begin{cases} -\infty, & \text{if } I \not\supseteq I'; \\ \infty, & \text{if } I \sim I'; \\ |\{I_i, I_j, \cdots, I_m\}|, & \text{otherwise.} \end{cases}$$

Intuitively, the objective of this metric is to favor interests that are themselves contained in many other interests, i.e., that are very specific and selective. Note that this metric is not symmetric.

The containment-based proximity metric can be used with any subscription language, provided that it defines a containment relationship. Of course, it applies best to subscription languages that are likely to produce subscriptions with many containment relationships. We wish to emphasize, however, that our routing protocol can be used with any other proximity metric, as we shall see in Section 8.

7.3.2 Network Description

Peers are organized in a *containment hierarchy tree*, based on the proximity metric f_c defined earlier. To simplify, we assume that each peer has expressed its interests by registering exactly one subscription (if that is not the case, the peer will appear multiple times in the hierarchy). The *containment*

[1] We intentionally do not use the term "equality" because some subscriptions languages allow interests to be formally different and yet match the same set of messages.

hierarchy tree is defined as follows. A peer P that registered subscription S is connected in the tree to a parent peer P_a that registered subscription S_a if S_a is the subscription in the system closest to S according to the proximity metric f_c. Given the definition of the metric f_c, this means that S_a is the deepest subscription in the tree among those that contain S. When we have more than one peer to choose from, we select as parent the peer that has the lowest number of children in order to keep the tree as balanced as possible.

We now consider the special case of peers that have registered equivalent interests in the system. From definition 6, it follows that if peer P_1 and P_2 are neighbors, P_1 would never deliver false positives to P_2, and vice versa. It is then clear that equivalent peers in the system should always be neighbors in the topology. As a consequence, in our tree topology, we organize equivalent peers together in specialized, balanced subtrees with limited degree L that we call *equivalence trees*. From the perspective of other peers in the system, an equivalence tree is considered as a single entity represented by its root node, which is positioned in the *containment hierarchy tree* using the rules described above. Non-equivalent children of the peers in an equivalence tree are always connected at its root.

Given that the containment relation is transitive, a peer contains all its descendants in the subtree rooted at itself. Since there may not be a peer in the system that contains all the others, we introduce an artificial node that interconnects all top-level peers and that we refer to as the *root node*. This node is purely virtual and is implemented by simply connecting top-level peers with each other through "sibling" links.

A simple containment hierarchy tree is illustrated in Figure 7.1. The equivalent peers P_8, P_9 and P_{10} are organized in the *equivalence tree* rooted at P_8. Note that both P_2 and P_4 contain P_3, but P_2 has a greater depth and is hence a better parent. Similarly, P_6 is connected to P_5 rather than P_1.

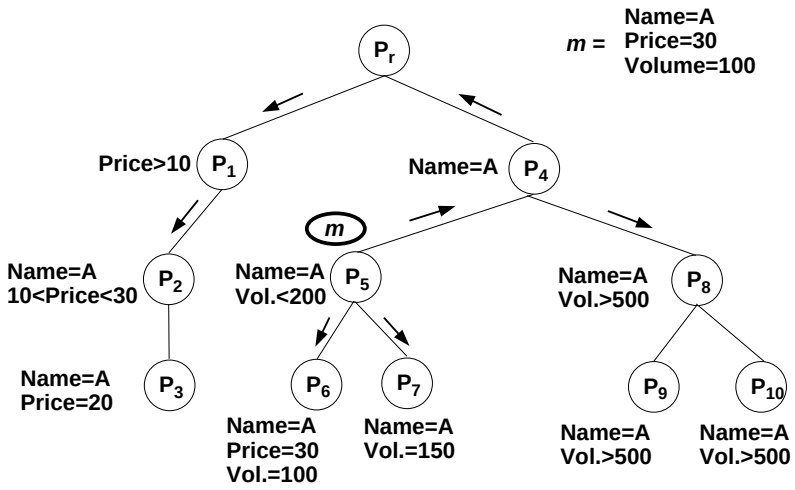


Figure 7.1: A simple pub/sub system for stock quotes with participants organized in a *containment hierarchy tree*. The subscription registered by a peer is represented next to it.

7.3.3 Impact on the routing process

Several properties concerning the routing process derive from the fact that peers are organized in a containment hierarchy tree. Those are identified and proved in this paragraph.

Property 9. *The paths followed by an event (without considering the directions) form a content distribution tree, i.e., a spanning tree rooted at the root node of the containment tree.*

Proof. This property directly comes from the routing process implemented in Algorithm 21. Indeed, consider a peer P that publishes event e. Since P is interested in e, it forwards it to all its neighbors,

including its upstream peer. Because of the containment hierarchy, this latter is also interested in e and hence forwards it to all its neighbors except P. Recursively, e reaches the root node. Now a peer that receives e from its upstream neighbor forwards it to its downstream peers if it is interested in it. Otherwise, it just discards it. Consequently, the paths followed by event e form a spanning tree, rooted at the root node of the containment tree. □

Property 10. *The leaves of a content distribution tree are either false positives or peers that are leaf nodes in the tree hierarchy. The inner nodes of the content distribution tree are all true positives (peers interested in the message and that received it).*

Proof. Consider a peer P that receives event e, such that P is a leaf in the content distribution tree of event e. Suppose that P is not a leaf in the content hierarchy tree. Then, P was not interested in e, otherwise it would have forwarded it to its downstream neighbors and would not be a leaf in the content distribution tree of event e. As a consequence, P is either a leaf in the content distribution tree, or a false positive. Now suppose that P is not a leaf in the content distribution tree, i.e., an inner node. Then, it forwarded e to its downstream neighbors. Hence, it was interested in e, and is a true positive. □

Property 11. *There are no false negatives in our system.*

Proof. Consider event e and a peer P that is interested in it. Because of the containment hierarchy tree, all ancestors of P in the hierarchy, including the root node, are also interested in e. Now consider the peer P_e that originally published e. Then if P_e is an ancestor of P, because of the routing process, e is forwarded to P. Now if P_e is not an ancestor of P, let P_c be the common ancestor of P_e and P. Because of the containment hierarchy, all ancestors of P_e are interested in e, and hence, because of the routing process, P_c receives e. Then, as previously explained, e is forwarded to P. □

In addition, the containment hierarchy tree topology enables to minimize the occurrence of false positives. Indeed, the fact that a peer P has for parent the peer of highest possible depth that contains it means that a message e has a greater chance of being discarded on the way from the root node (or the common ancestor of peer P and the peer that originally published e) to P. If event e traverses all of P's ancestors, it means that these peers were interested in the message and there is a good chance that P is also interested in it.

We wish to point out that false positives can only be avoided by having each peer know about its neighbors' interests, which conflicts with our design guidelines.

Finally, routing is *efficient* in terms of bandwidth usage. Indeed, only the leaves of the spanning tree followed by a message may be false positives. All other nodes that are involved in the routing process of event e are interested in it.

A simple example is illustrated in figure 7.1, where peer P_5 publishes message m. The path followed by m is highlighted by the arrows.

7.3.4 Maintaining the containment hierarchy tree

We have implemented several peers management algorithms to maintain the containment hierarchy tree when peers dynamically join and leave the system. Those algorithms are described in this section. We would like to point out that these algorithms are executed only when a peer joins or leaves the system, which we assume to happen at a much lower frequency than the publication of messages. As previously discussed, the algorithm executed for routing such messages is trivial and extremely efficient.

Algorithm 22 Recursive Join algorithm: On receiving $JOIN(Q, S_Q, d, s)$ at peer P

1: **if** $S_P \sim S_Q$ **then**
2: **if** P has less than L equivalent children **then**
3: Send reply $JOIN\_REPLY(P, \sim, d+1)$ to Q
4: **else**
5: Choose an equivalent child $P_i^{\sim}$, uniformly at random.
6: Send $JOIN(Q, S_Q, d+1)$ to $P_i^{\sim}$
7: **end if**
8: **end if**
9: **if** $S_P \supset S_Q$ **then**
10: Send $JOIN\_REPLY(P, \supset, d+1, \#children)$ to Q
11: **for all** Non equivalent child P_i **do**
12: Send $JOIN(Q, S_Q, d+1, \#children)$ to P_i
13: **end for**
14: **end if**
15: {*FOR REORGANIZATION AND CONNECTION BALANCING PURPOSES*}
16: **if** $S_P \subset S_Q$ **then**
17: Send reply $JOIN\_REPLY(P, \subset, d+1, s)$ to Q
18: **end if**
19: **if** there are no relations between S_P and S_Q **then**
20: **for all** non equivalent children P_i **do**
21: Send $JOIN(Q, S_Q, d+1, \#children)$ to P_i
22: **end for**
23: **end if**

Algorithm 23 Joining procedure for peer Q

1: {*FIRST PHASE: CONNECTION PHASE*}
2: **if** $JOIN\_REPLY(P, \sim, d_P, s) \in \{JOIN\_REPLY\}^{\sim}$ **then**
3: Send $JOIN\_REQUEST(Q, \sim)$ to P
4: **else**
5: **if** $\{JOIN\_REPLY\}^{\supset} \neq \emptyset$ **then**
6: Select $JOIN\_REPLY(P, \supset, d_P, s)$ with maximal d_P and minimal s
7: Send $JOIN\_REQUEST(Q)$ to P
8: **else**
9: Send $JOIN\_REQUEST(Q)$ to *Root*
10: **end if**
11: **end if**
12: {*SECOND PHASE: REORGANIZATION PHASE*}
13: **for all** $JOIN\_REPLY(P, \subset, d_P, s) \in \{JOIN\_REPLY\}^{\subset}$ **do**
14: **if** $d_P \leq d_Q$ **then**
15: Send $REORG(Q)$ to P
16: **end if**
17: **end for**
18: {*THIRD PHASE: CONNECTIONS BALANCING PHASE*}
19: Sort $\{JOIN\_REPLY(P, \subset, d_P = d_Q + 1, s)\}$, by increasing s
20: **for all** $JOIN\_REPLY(P, \subset, d_P, s) \in \{JOIN\_REPLY(P, \subset, d_P = d_Q + 1, s)\}$ **do**
21: **if** $\#children < s$ **then**
22: Send $REORG(Q)$ to P
23: **end if**
24: **end for**

Join algorithm

Let Q be a new peer that wishes to join the system and register subscription S_Q. In order to insert Q in the tree topology, the system is first *probed* to find adequate containment relationships between S_Q and the other registered subscriptions. For that purpose, Q sends a *join* message to some node R whose subscription contains S_Q (a matching node can be found by starting from any peer and moving upward the tree). As Q will be inserted in the subtree rooted at R, one will typically choose R as the root node of the system, but it may be sometimes desirable to choose another node for offloading the root. The *join* message is then propagated recursively downward the subtree and processed at each encountered peer. The joining algorithm executed at such a peer P, with registered subscription S_P, is described in Algorithm 22 and explained as follows.

If $S_P \sim S_Q$, then P is an equivalent peer. Then, Q must belong to the same equivalent tree as peer P. If P has less than L equivalent children, it can accept an additional child and hence sends a *JOIN_REPLY* message to Q, indicating that P is an equivalent peer of depth $d+1$ (lines $2-4$). Note that peers do not know their depth, but learn it when they receive *JOIN* messages. Now if the number of equivalent children that P has is above the allowed degree L, then P propagates the *JOIN* message downstream, to an equivalent peer chosen uniformly at random, and the same procedure will apply there (lines $5-6$). Choosing the peer uniformly at random allows the equivalent tree to be balanced. Also, to prevent the root of an equivalence tree to be overloaded, that node has a smaller degree than the degree L of the equivalence tree. That is, it is allowed to have only a certain number of equivalent children, smaller than L. Practically, the degree at the roots of equivalence trees is a certain fraction $\frac{L}{r}$ of the tree's degree L. Note that a peer that does not have any equivalent children is considered as the root of an equivalence tree (with only that peer). Also, a peer that has some equivalent children but whose parent is not an equivalent peer is the root of a "real" equivalence tree. Hence, only peers that have an equivalent parent have a degree of L, at most. Other peers always have less than a certain fraction of L, $\frac{L}{r}$, equivalent children. A peer learnt that its parent is an equivalent peer when it actually joined the system with the joining procedure (in Algorithm 23, to be explained shortly).

If $S_P \supset S_Q$, then P is a potential candidate for being Q's parent. Hence, it sends a *JOIN_REPLY* to Q, indicating that it contains it. It also indicates its depth and the number of children that it currently has, with the *#children* function (line 10). However, there may be better candidates downstream. Indeed, recall that according to the containment-based metric, peers have for parent a peer of highest depth, that contains it. Hence, P propagates the *JOIN* message to its (non equivalent) children, indicating in it the number of children that it currently has (lines $11-13$).

Next, we proceed with the *reorganization phase*, which might lead to moving some existing peers so as to become Q's children. Indeed, when Q has connected to a parent in the tree, some other peers may now be closer to Q than their actual parent in the tree. Hence, if neither $S_P \sim S_Q$ or $S_P \supset S_Q$, P checks if it is contained by Q. If it is the case, then P might be moved to become Q's child (lines $18-20$). Hence, P sends a *JOIN_REPLY* message to Q, indicating that it is contained by Q, along with its depth. P also indicates the number of children that its *parent* has, for connection balancing purposes. Indeed, when peer Q receives such a *JOIN_REPLY*$(\subset)$ message from P, it might decide to move P as one of its child even if P's depth remains the same, i.e., even if its position in the containment hierarchy is not improved. This will enable to unload P's parent if this latter has a large number of children. Hence, amongst the peers whose depth remains unchanged if they become one of its children, Q first reorganizes the peers whose parent has the largest number of children. It is important to note that at peer P, the parameter s included in a received *JOIN* message is always the number of children of its parent. Indeed, a peer that propagates a *JOIN* message downstream always includes the number of children that it currently has, with the *#children* function (except in an equivalence tree).

Finally, if there are no relationships between S_P and S_Q, then it is still possible for its children to be reorganized (since the containment relationship is not a total order). Consequently, P propagates the *JOIN* message to its non equivalent children (lines $21-25$).

Peer Q then uses the results of this probing phase to actually join the tree. The procedure is detailed in Algorithm 23. Peer Q first connects to a parent that is either an equivalent peer, if any (lines $2-3$), or a peer of highest depth d_P and minimal degree s, whose subscription contains S_{new} (lines $5-7$). If Q did not receive any *JOIN_REPLY* messages from peers that contain it, then it joins at the root of the tree topology (line 9).

Next, Q proceeds to the *reorganization* phase, which as previously mentioned, might lead to moving some existing peers so as to become Q's children. A peer P is reorganized if it is contained by Q and if its depth is less or equal to that of Q. Hence, Q sends a *REORG* message to all those peers (lines $14-19$). Note that Q's depth is known from the connection phase, from the d_P field that was included in the *JOIN_REPLY* message.

Finally, Q proceeds with the connections balancing phase (lines $20-27$). The peers that are contained by Q, whose parent have the same depth as Q, and such that their parent have a degree higher than Q's, are reorganized to be Q children (in a $JOIN\_REPLY(P, \subset, d_P, s)$ message, s is the number of children of P's parent). The peers whose parents have the highest degree are favoured.

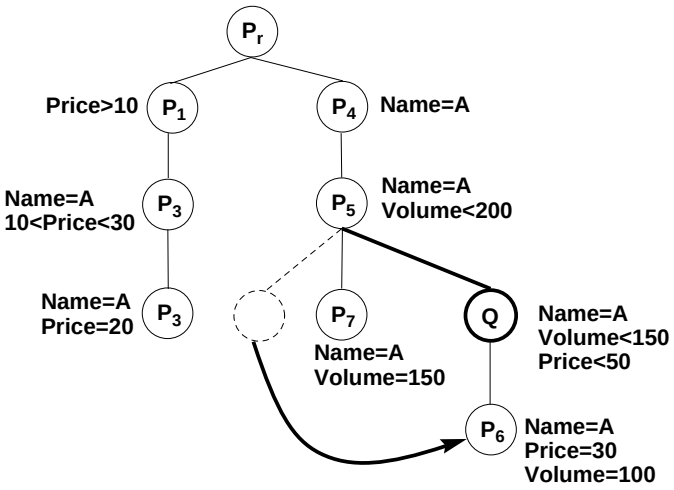


Figure 7.2: Peer Q has been inserted in the network with P_5 as its closest peer. Peer P_6 is reorganized as Q's child because the latter is a better parent than P_5.

Alternate join algorithms

The *reorganization* phase introduces significant overhead in the system, in particular because it requires additional propagations of join messages. As a consequence, we have implemented three different flavors of the join algorithm. The first variant of the algorithm, termed *full join*, was presented in Section 7.3.4 and Algorithms 22 and 23. It *always* performs all possible reorganizations to obtain the most accurate containment hierarchy tree and minimize the occurrence of false positives, at the cost of a higher complexity.

The second variant of the algorithm is termed *basic join* and *never* performs any reorganizations. It has the lowest complexity but at the same time produces less accurate containment hierarchies with poor load-balancing properties. That variant can be trivially derived from Algorithms 22 and 23 by simply ignoring the reorganization and connections balancing phases. In other words, Algorithm 22 is executed until line 16 and Algorithm 23 until line 11.

Finally, the third variant is termed *adaptive periodic join* and performs reorganizations *periodically* or to *adapt* to a particular situation. Only the recursive join algorithm (probing phase, Algorithm 22) is modified, the joining procedure remains unchanged (Algorithm 23). It works as follows. Consider peer P where the recursive joining algorithm is executed.

1. If P or its parent have exceeded a given connections limit L_{APJ}, then the *full joining* algorithm is executed.

2. Otherwise, the *full joining* algorithm is executed with probability γ_{APJ} and the *simple joining* algorithm with probability $1 - \gamma_{APJ}$. Parameter γ_{APJ} will be referred to as the *reorganization rate*.

In other words, the *adaptive periodic joining* algorithm always reorganizes the peers whose parent have exceeded the connection limit. The others are reorganized with probability γ_{APJ}. It reaches a compromise between joining complexity and routing accuracy.

An example of the *full join* algorithm is illustrated in Figure 7.2.

Leave algorithm

We now describe the process of leaving the network. When peer P with registered subscription S_P wishes to leave the system—or when it fails—each of its children has to be reconnected to another parent in the tree. If P is part of an equivalence tree, then we simply perform a *leaf promotion*: we look for a leaf in the subtree rooted at P and promote it to P's position. If P is not part of an equivalence tree, there is no trivial replacement parent for P's children. In fact, since peers are stateless in the system, the best potential replacement for P known by the peers is P's own parent. Therefore, the leave algorithm simply consists in reconnecting P's children to their grand-parent. It follows that every peer needs to know its grand-parent (or several ancestors for increased fault-tolerance); this is achieved with trivial modifications to the join algorithm and negligible additional control traffic. This algorithm, which consists in reconnecting at the parent of the leaving peer, will be referred to as the *basic leave* algorithm.

Although extremely simple, the *basic leave* algorithm may cause the accuracy of the containment hierarchy tree to degrade over time. This is due to the fact that P's parent may not be the closest peer in the system for P's children, although the accuracy of the containment hierarchy tree should not be significantly affected. In addition, P's parent may suffer from the increased number of connections that it has to manage. The arrival of new peers enables to cope with these issues in some extent, since the *full* or the *adaptive periodic* joining algorithms can enhance the accuracy of the containment tree and implement connections balancing techniques. However, to maintain an optimal tree and to handle the case where P's parent also fails, it is possible for P's children to look for another replacement parent by executing the join algorithm, typically starting from some ancestor, at the price of higher overhead. Note that if the containment tree was formerly optimal, a peer that has rejoined the system can not have a higher depth than before. Hence the other peers are not susceptible to be reorganized and it is sufficient to use the *basic join* algorithm. Also, note that if we wish to maintain an optimal tree, additional peers among P's descendants might need to re-evaluate their position as well if P's departure has decreased their depth. We will refer to this algorithm as the *full leave* algorithm.

7.3.5 Scalability issues

The instantiation of our system using the containment-based metric organizes peers in tree topologies. It follows that high-level peers, particularly the root node, receive a high number of messages. As these peers have very "broad" interests, it is not unnatural that they receive a high percentage of published messages—they are interested in those messages. They are also more exposed to control messages from the peer management algorithms, but this traffic can be reduced by confining the join and reorganization procedures within selected subtrees.

The most serious scalability issue comes from the fact that high-level peers may have a large number of neighbors to forward messages to (recall that both the routing process and the peers management algorithms are straightforward and demand very little resources). To address this problem, we have performed slight modifications to our original protocol to reduce by a great deal

the bandwidth utilization at the peers. Informally, to reduce the bandwidth usage of an overloaded peer P, we form some clusters among P's children. Messages are then propagated by P to only one peer in each cluster (chosen uniformely at random), and are further propagated inside the cluster with sibling links.

This approach enables to reduce greatly the bandwidth requirements of peers, in particular those that have a large number of children. However, the limitation of this approach is that it slows down the propagation of messages in the system. It is therefore desirable to use it only for high-level peers.

The experimental evaluation of the techniques presented in this section can be found in Section 9.1. We now present an organization of peers based on subscriptions' similarities.

7.4 Organizing Peers according to Similarity

7.4.1 Overview

Motivations

As previously mentioned, the routing protocol used to disseminate messages does not make specific assumptions about the proximity metric used to organize the peers in semantic communities. We now present a generalization of the containment-based proximity designed to alleviate two of its limitations: (1) its poor applicability to subscription language and/or consumer workloads with little or no containment relationships, and (2) its tree topologies that may be fragile with dynamic consumer populations. This generalization is based on the general principle of interest similarity.

Similarity metric

We first define the notion of interest similarity as follows.

Definition 7 (Interest similarity). *Consider two interests I_1 and I_2. Let $\mathcal{I}$ be the universe of all possible interests. We define the similarity between I_1 and I_2, noted $Sim(I_1, I_2)$, as a function from $\mathcal{I}^2$ in the interval $[0, 1]$ that returns the probability that a message m matching I_1 also matches I_2.*

We then define our proximity metric based on interest similarity, which we refer to as f_s:

Definition 8 (Proximity metric f_s). $f_s : \mathcal{I}^2 \mapsto [0, 1]$:

$$f_s(I_1, I_2) = \frac{Sim(I_1, I_2) + Sim(I_2, I_1)}{2}$$

Note that the proximity metric f_s is symmetric, that is, if I_1 is close to I_2 according to f_s, then I_2 is equally close to I_1. However, the similarity function is a priori *not* symmetric.

7.4.2 Network description

We now describe the hierarchical organization of peers based on the proximity metric f_s. A peer P with registered interest I has a set of n neighbors P_i, which are the n peers in the system with interests closest to I according to f_s (in case of equality, the peers with less connections are chosen). In turn, P is for a certain number of other peers in the system, one of the n best peers according to f_s (such that I is amongst the n interests in the system closest to their subscription according to f_s). P and those peers are thus neighbors.

Note that if the similarity metric is well designed, $f_s(I_1, I_2)$ is maximal if I_1 and I_2 are equivalent (ideally, $f_s(I_1, I_2) = 1$). Hence, as in the tree topology of Section 7.3, equivalent peers are also neighbors in this organization based on similarity.

This approach effectively organizes the peers in "interest communities," i.e., groups of peers that share similar interests. Because of the definition of the similarity function and the proximity metric

f_s, this organization optimizes routing accuracy by minimizing the number of false positives and negatives exchanged by neighbor peers. To maintain good connectivity between the communities and prevent some of them from being closed (because their interests do not compare with the other communities' interests), P also chooses r parents at random in the system, in addition to the n peers selected with f_s. Routing proceeds as described in Section 7.2.1.

7.4.3 Consequences

Obviously, if $n + r > 1$, the peers are organized in graphs instead of trees. We can also observe that, if we set $n = 1$, $r = 0$ and we define $Sim(I_1, I_2) = 1$ if $I_1 \supseteq I_2$ and $Sim(I_1, I_2) = 0$ otherwise, peers are organized using a containment-based metric similarly to the topology of Section 7.3.

The organization of peers in graphs rather than trees benefits from several advantages. It has better connectivity and is hence more resilient to failures and frequent arrivals or departures. Also, it has better flexibility and offers higher scalability since the traffic load is more evenly distributed amongst the peers. Finally, this model can be applied to any subscription languages and consumer workloads even if the subscriptions share little or no containment relationships.

However, one major drawback of this approach is that it does not prevent the occurrences of false negatives in the system (unless the definition of $Sim(I_1, I_2)$ is based on containment). This problem is alleviated by the fact that a peer has several parents in the system and, hence, a message may reach the peer via multiple paths. Also, we can enhance the routing algorithm to have even better control on false positives and false negatives. For instance, we can add an indulgence factor γ that allows a peer to forward a message even if it is not interested in it. This process, which may be performed only γ times per message, is expected to reduce the false negatives ratio, notably by improving the "traversal" of messages between communities. Another improvement is to add a random neighbor forwarding probability ρ, which controls the probability for a peer P to actually forward a message to its r random neighbors. The base case, $\rho = 100\%$, produces fewer false negatives but more false positives; lower values of ρ have the opposite effect.

In addition, the total number of neighbors that a peer has to manage is not bounded a priori. However, we expect that for large enough populations, the number of connections should be fairly distributed. That is, there is no "best peer for everybody" in the system. Besides, a given peer may simply decide to refuse additional connections, candidate peers connecting to another, less loaded peer.

7.4.4 Peers management

We have implemented several peers management algorithms to maintain the hierarchy graph when peers dynamically join and leave the system. Those algorithms are described in this section. As previously mentioned, these algorithms are executed only when a peer joins or leaves the system, which we assume to happen at a much lower frequency than the publication of messages.

Algorithm 24 Joining procedure for peer Q

```
1: Send JOIN(Q, S_Q) to some peer in the system
2: when enough JOIN_REPLY(P, f_s(S_Q, S_P), a) have been received or timer expires
3:    Build sorted list L_f with n highest (P, f_s(S_Q, S_P))
4:    Send CONNECT to each P ∈ L_f
5:    Send CONNECT to r random peers P_i ∉ L_f
6: end when
```

Join algorithm

Consider a new peer Q that wishes to join the system and register subscription S_Q. The principle of the join algorithm is similar to that of the containment hierarchy tree topology of Section 7.3, and is described in Algorithms 25 and 24.

Algorithm 25 Join algorithm: On receiving $JOIN(Q, S_Q)$ at peer P

1: Send $JOIN\_REPLY(P, f_s(S_Q, S_P), a)$ to Q
2: Forward the *JOIN* message to all neighbors except the one from which the message was originally received
3: Declare S_{f_n} such that $(P_{f_n}, f_s(S_P, S_{f_n}))$ is the last element in L_f
4: **if** $f_s(S_Q, S_P) > f_s(S_P, S_{f_n})$ **then**
5: Send *DISCONNECT* to P_{f_n}
6: Send *CONNECT* to Q
7: Insert $(Q, f_s(S_Q, S_P))$ in L_f
8: **end if**

The system is first probed to find the n best neighbors for Q. Those peers are the n peers with subscription closest to S_Q according to f_s. However, unlike the containment metric, the similarity metric f_s is not transitive. In other words, if I_1 is close to I_2 and I_2 is close to I_3, I_1 may not be close to I_3. Hence, the entire network (or a reasonably large portion of it) must be probed so as to collect proximity results (lines $1-2$ in Algorithm 25). For that purpose, Q sends a *JOIN* message to some peer in the system, which is then broadcast in the system. To avoid loops, a peer ignores additional instances of the same *JOIN* messages (at the cost of temporary and negligible state maintenance).

Q then uses the results of this probing phase to actually join the system. When it has received a sufficient number of *JOIN_REPLY* messages, or after expiration of a timer, it selects the n peers with the highest proximity results and sends them a *CONNECT* message (lines $2-4$ in Algorithm 24). Also, Q sends a *CONNECT* message to r random peers different than the previous ones (line 5 in Algorithm 24).

Next, Q proceeds with the *reorganization phase*, whose principle is the same as in Section 7.3. Each peer in the system maintains a list L_f of its n best neighbors, sorted according to the proximity results of their subscriptions with its own: $L_f = \{(P_{f_i}, f_s(S_P, S_{f_i}))\}$. The list is built when the peer originally joins the system (line 3 in Algorithm 24).

When peer Q has joined the system, some other peers may now be closer to Q than one of their actual neighbors, precisely the one with the smallest proximity result in their n best neighbors list. Hence, they disconnect from that neighbor, connect to Q instead and update their list (lines $2-7$ in Algorithm 25).

Reducing probing overhead

As previously mentioned, when a new peer Q joins the system, a large enough portion of the network must be probed so as to collect proximity results. Hence, subsequently to this probing phase, peer Q receives a potentially large number of replies. To limit the scope of this problem, we can use a slightly modified version of the join mechanism. Informally, the principle of the improved scheme is the following. Consider a peer P that receives a $JOIN(Q, S_Q)$ message from neighbor peer P', for the joining of new peer Q. Instead of immediately sending the reply with its proximity results to peer Q (as in line 1 in Algorithm 25), P waits for a reply from its neighbors (except P'). After a sufficient number of replies have been received, or after expiration of a timer, P aggregates the results so as to select only the n peers (at most) with the n best proximity results, and r other different peers (at most), chosen uniformely at random. P then sends the results to peer P'. Consequently, join replies follow the reverse path of join requests and peer Q only receives a reply from the peer in the system to which it issued its *JOIN* request.

Leave algorithm

We now describe the process of leaving the network. When peer P with registered subscription S_P wishes to leave the system—or when it fails—each of its neighbors for which P was either one of the n best peers in the system (which we refer to as a semantic neighbor) or a random neighbor has to find another neighbor.

One approach consists in having each of P's neighbors rejoin the system with the mechanisms described above. This approach, which we refer to as the *full leave* scheme enables to maintain the

most accurate topology. However, it introduces significant overhead in the system. Consequently, we propose an alternate scheme, termed the *basic leave* mechanism.

The *basic leave* mechanism works as follows. Let P' be one of P's neighbors. If P is a random neighbor for P', then P' has to find another random neighbor in the system. Then, if P leaves the system "properly", P' can simply choose one of P's semantic neighbors (one of P's best neighbors). As a result, the new random neighbor of P' belongs to the same semantic community as the former one (P). Otherwise, P' chooses one of the random neighbors of its neighbors (different from its current neighbors). Now if P is a semantic neighbor for P', P' can simply choose one peer amongst the semantic neighbors of its own semantic neighbors (different from its current neighbors). Indeed, it is highly unlikely that a better peer is found in a different semantic community.

The *basic leave* mechanism results in very little overhead in the system, since for each neighbor P' of peer P, only the neighbors of the neighbors of P' are involved in the process. Obviously, this approach may yield to a less accurate network. However, only a high number of consecutive leaves may significantly degrade the accuracy of the topology. In addition, the joining of new peers improves the overall accuracy of the system, thanks to the reorganization procedure.

7.5 Conclusion

We have studied a novel approach to pub/sub, based on the P2P paradigm, that specifically address some of the limitations of existing systems. In particular, our network does not rely on a dedicated network of content routers, nor on complex filtering and forwarding algorithms: it features an extremely simple routing process that requires almost no resources and no routing state to be maintained at the peers. The price to pay for this simplicity is that routing may not be perfectly accurate, in the sense that some peers may receive some messages that do not match their interests (false positives), or fail to receive relevant messages (false negatives). By organizing the peers according to adequate proximity metrics, one can limit the scope of this problem. We have proposed a containment-based proximity metric that allows us to build a bandwidth-efficient network topology that produces no false negatives and a minimizes the number of false positives. We have also developed a proximity metric based on subscription similarities that yields a more solid graph structure with optimized routing accuracy in terms of false negatives and false positives. The evaluation of these techniques will be presented in Section 9.1.

Chapter 8

Similarity-based proximity metric for XML documents

The work presented here is part of an effort to determine similarity between XPath subscriptions. There is room for improvements and refinements, but preliminary evaluation results are very promising and this work paves the way for future research.

8.1 Introduction

8.1.1 Motivations

In this chapter, we present the proximity metric based on subscriptions similarities that we implemented for XML documents and XPath subscriptions. We used the proximity metric to organize the peers in the pub/sub system that we presented in Chapter 7 in an efficient graph topology according to their interests, so as to minimize the occurrences of false positives and false negatives. However, our proximity metric can be used in different contexts, and to address different data management problems.

The goal of the proximity metric is to evaluate the proximity between two given XPath expressions in terms of filtering error of XML documents. That is, the error that would be induced when filtering XML documents against one expression instead of the other. For example, for a given subscription S_i, the proximity metric enables to find in a set of subscriptions the subscription that is closest to S_i, in that filtering XML documents against that subscription instead of S_i induces the minimal error.

8.1.2 Overview

Because of Definitions 7 and 8, to implement the proximity metric f_s, we need to implement a similarity function Sim. Note that the proximity metric is a symmetric function, i.e., $f_s(S_1, S_2) = f_s(S_2, S_1)$. However, the similarity function is a priori not symmetric. We have implemented a similarity function for XML documents and XPath subscriptions, which computes the probability that a message matching a subscription also matches the other. For that purpose, we introduce the notion of subscription's expansion, a data structure that is built from a subscription and that represents the possible constraints on the structures and contents that a document matching that subscription may contain.

To build the subscriptions expansions, and to compute the similarity between subscriptions, we need to have some information about the subscriptions and the XML documents. For that purpose, we suppose that we know either the *grammar* that is used to express the XPath expressions and the XML documents, or a *history* or *synopsis* of previous documents. The grammar can be either a *Document Type Descriptor*, or *DTD*, or an *XML Schema*. In our current implementation, only DTDs are supported. If the grammar is not known, then we may either use a history or a synopsis of

previous documents. A history contains *entire* XML documents that were seen previously, whereas a synopsis is a compact representation of a history, it "factorizes" documents in a tree structure annotated with some statistical information about them. We currently only support documents synopsis, and we used the definition introduced in [34]. However, our function can be trivially extended to deal with histories of documents. Note that it is often impossible to maintain a history of documents, due to the large amount of information contained in it. In contrast, a synopsis is a much more compact, and easy to maintain, data structure. If both the DTD and a documents synopsis are known, then both can be used for better accuracy. Finally, when the DTD is known, we exploit the *correlations* between elements specified in it, when building subscriptions expansions and computing the similarity between subscriptions.

8.2 Document synopsis

As mentioned above, it is simply impossible to maintain the complete history of documents H (i.e., the full set of streaming documents). Instead, we approximate H by a concise structure, which we refer to as the *document synopsis*, or simply *synopsis*. Our synopsis for H, denoted by syn_H, captures path statistics for documents in H, and is built *on-line* as XML documents stream by. The document synopsis essentially has the same structure as an XML tree, except for two differences. First, the root node of syn_H has the special label "/". Second, each non-root node t in syn_H has a frequency associated with it, which we denote by $freq(t)$. Intuitively, if $l_1/l_2/\cdots/l_n$ is the sequence of tag names on nodes along the path from the root to t (excluding the label for the root), then $freq(t)$ represents the number of documents T in H that contain a path with tag sequence $l_1/l_2/\cdots/l_n$ originating at the root of T. The frequency for the root node of syn_H is set to N, the number of documents in H.

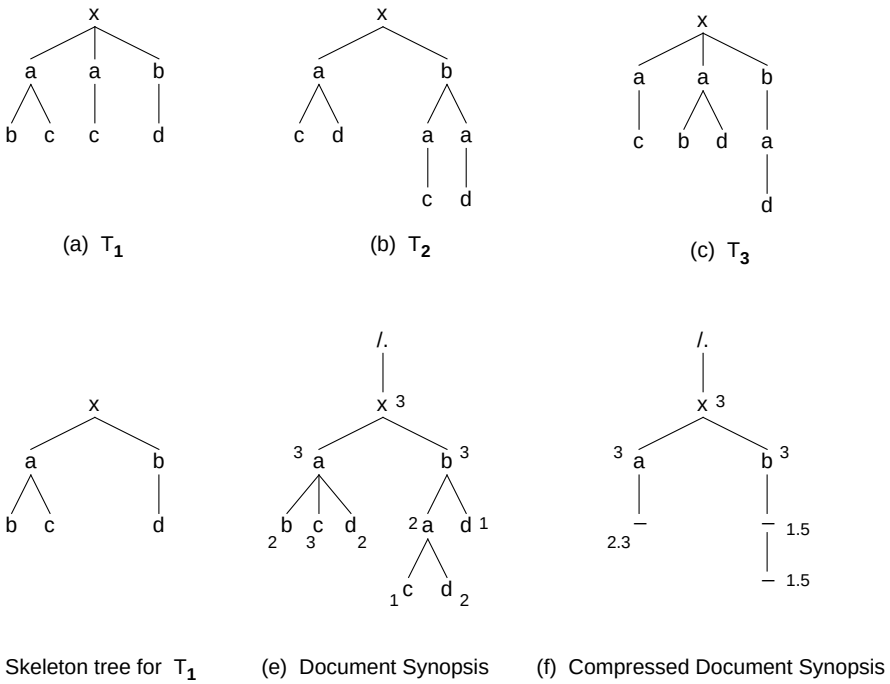


Figure 8.1: Example Documents, Skeleton Tree, Document Synopsis and Compressed Document Synopsis.

As XML documents stream by, syn_H is incrementally maintained as follows. For each arriving document T, we first construct the *skeleton tree* T_s for document T. In the skeleton tree T_s, each node has at most one child with a given tag. T_s is built from T by simply coalescing two children of a

node in T if they share a common tag. Clearly, by traversing nodes in T in a top-down fashion, and coalescing child nodes with common tags, we can construct T_s from T in a single pass (using an event-based XML parser). As an example, Figure 8.1(d) depicts the skeleton tree for the XML-document tree in Figure 8.1(a).

Next, we use T_s to update the statistics maintained in our document synopsis syn_H as follows. For each path in T_s, with tag sequence say $l_1/l_2/\cdots/l_n$, let t be the last node on the corresponding (unique) path in syn_H. We increment $freq(t)$ by 1. Figure 8.1(e) shows the document tree (with node frequencies) for the XML trees T_1, T_2, and T_3 in Figure 8.1(a) to (c). Note that it is possible to further compress syn_H by using techniques similar in spirit to the methods employed by Aboulnaga et al. [3] for summarizing *path* trees. The key idea is to merge nodes with the lowest frequencies and store, with each merged node, the average of the original frequencies for nodes in syn_H that were merged. This is illustrated in Figure 8.1(f) for the document tree in Figure 8.1(e), and with the label "-" used to indicate merge nodes. We currently only support uncompressed synopsis, However, our proximity metric can be easily extended to work when syn_H is compressed.

8.3 Correlations in DTD

In a DTD, the correlations between elements come in the form of two different constraints:

1. Constraints on the cardinality of single elements. Those constraints are expressed using regular expression operators indicated next to an element. The following operators are supported: ? which means that the considered element may appear zero or one time but not more, $*$ for any number of instances, $+$ for at least one instance, and *nothing* (i.e. no operator) for exactly one instance of the element.

2. Constraints expressed as choices in groups of elements. Those constraints are expressed using the $|$ (OR) logical operator, and come in the form $(a|b)$, which means that either element a or b may appear, but not both.

When either element a or element b may appear, but not both, we say that a and b are in *opposition*. This happens when a and b are found in a element group separated by the $|$ operator: $(a|b)$. Note that the definition extends to several different elements: $(a_1|\cdots|a_n)$. Then elements $a_1 \cdots a_n$ are in *opposition* with each other. In addition, an element may be in opposition with itself. This happens when the regular expression operator appended to an element is either ? or *nothing*, for example: $a?$. Then, element a is in opposition with itself, in the sense that no two elements with name a may appear simultaneously.

Consequently, we implemented a DTD parser, that parses a DTD, examines the aforementioned constraints, and produces an understandable data structure. The parsing involves the development of regular expression operators and the $|$ operator in element groups, according to simple rules, so as to determine the elements that are in *opposition*. Those rules are:

- As previously mentioned, if $a?$ or a (with no cardinal) is found, then element a is in opposition with itself.

- $(a\#)\&$ is always equivalent to $a*$, where $\#$ and $\&$ are two *different* regular expression operators. For example, $(a+)?$ is equivalent to $a*$. Element a is then *not* in opposition with itself.

- $(a_1|a_2)*$ is equivalent to (a_1*, a_2*) (except for the order). Elements a_1 and a_2 are *not* in opposition.

- $(a_1|a_2)?$ is equivalent to $(a_1?|a_2?)$ (except for the order). Elements a_1 and a_2 are in *opposition*.

- $(a_1|a_2)+ \neq (a_1 + |a_2+)$ and $\neq (a_1+, a_2+)$ as well. In fact, $(a_1|a_2)+$ would be equivalent to $\big((a_1|a_2), (a_1|a_2)*\big)$ but this is prohibited in most DTDs, because this is not a deterministic

expression (a given element cannot appear more than once). In any case, elements a_1 and a_2 are *not* in opposition.

- $(a_1|a_2)$. As previously mentioned, elements a_1 and a_2 are in *opposition*.

The rules exposed above trivially extend to the case of more than two elements in a group, or in the case of multiple groups. Also, the notion of elements *opposition* trivially extends to the case of multiple elements. Note that the *opposition* relation is transitive, i.e., if element a_1 is in opposition with a_2 and a_2 is in opposition with a_3, then a_1 is in opposition with a_3.

Exploiting the correlations between elements specified in a DTD enables to improve the *correctness* and the *accuracy* of our proximity metric. To the best of our knowledge, this work is the first attempt to implement a function that estimates the similarity between tree patterns in terms of matching documents *and* that takes into account the correlations between element names specified in the DTD.

An example of correlations between elements that can be found in a DTD is illustrated in Figure 8.2(a).

8.4 Subscription expansion

8.4.1 Overview

Consider subscription S. The expansion E_S of subscription S is a tree of nodes where each node N has the following attributes:

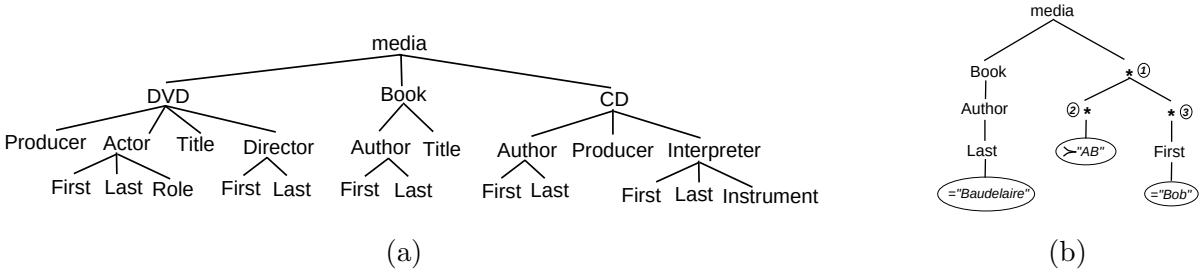
- a label $label(N)$, is a *valid* element name. Hence, $label(N)$ *cannot* be a wildcard or an ellipsis as in the case of XPath expressions.
- a probability $0 < Pr_N \leq 1$.
- a class $class(N) \in \mathbf{N}$.
- one or more predicates on the value of element with name $label(N)$.
- a set of *correlation groups*, each consists of a set of nodes among N's children.

E_S has an artificial root node termed r_{E_S}.

Roughly speaking, E_S is a compact representation of all -or some of- the possible documents that match subscription S. More precisely, it contains the possible constraints on the structure and the values that documents matching S may have. The label of a node N in E_S refers to the name of an element in a document that match S. Its probability refers to the likeliness that an element with name $label(N)$ and that verifies the predicates of node N occurs, given that all ancestor nodes have occurred. Its class identifies the node in S that it refers to: two nodes in E_S with the same parent and the same class do not occur at the same time (at least, this is not explicitly specified by subscription S). The list of correlations groups determine the children of N that are in *opposition* according to the DTD. Each correlations group contains some children that are in opposition with each others. Of course, if the DTD is not known, then the nodes in E_S do not maintain any correlation groups. Finally, the predicates describe constraints on the element's values or attributes. We currently support several common types for elements' values and attributes, some of which are *char*, *strings*, *boolean*, *date* and the usual numerical types *int*, $float$, etc. The operators are the equality ($=$) and order relation ($>$), plus the string prefix operator ($\prec$), the string postfix operator ($\succ$), and the substring operator ($\in$).

The computation of the node probabilities is based on either the knowledge of the DTD, or on a synopsis of previous documents, or both. Also, note that in the case where a subscription S contains one or more ancestor/descendant ($//$) operators, the expansion of S can be infinite if there are loops in the DTD. In that case, we truncate the expansion to a predefined length or to a minimum probability τ.

```
<!ELEMENT media ((Book,CD,DVD)*) >
<!ELEMENT Book ((Author|Title)+) >
<!ELEMENT Author ((First|Last)+) >
<!ELEMENT CD  (Author+,(Producer | Interpreter)?) >
<!ELEMENT Interpreter  ( (First|Last)+ , Instrument*) >
<!ELEMENT DVD (Title?, Actor+ , (Producer | Director)?) >
<!ELEMENT Actor ((First|Last)+,Role?) >
<!ELEMENT Director ((First|Last)+) >
```



(a) (b)

Figure 8.2: (a) A simple DTD that describes media libraries/databases. The DTD is represented as plain text (as found in a real DTD file), and as a tree below. Note the correlations between elements specified in the text representation. In particular, both elements "Producer" and "Director" cannot be found under element "DVD". Also, both elements "Producer" and "Interpreter" cannot be found under element "CD" (b) Graphical representation of subscription S, that queries for a book with author's last name "Baudelaire" and any media with an element with a value postfix (that starts with) of "AB" and an element with "Bob" as first name. The wildcards have been numbered for clarity.

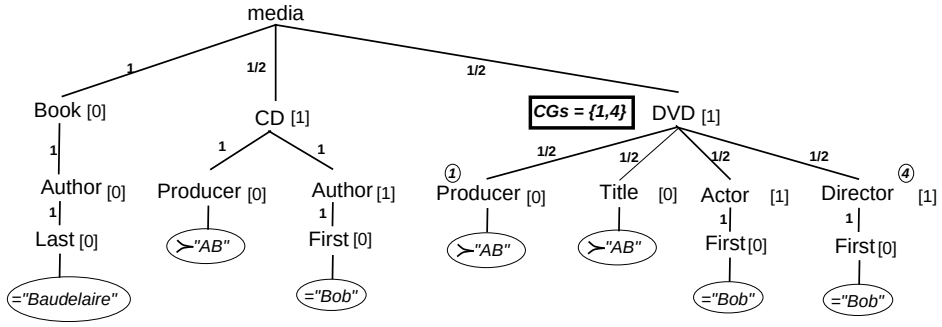


Figure 8.3: Expansion E_S of subscription S of Figure 8.3. A node is identified by its label. Its probability is indicated next to the branch linking it to its parent. Its class is indicated in brackets. Its predicates, if any, are indicated in circles below.

Example 5. *An example of the notion of subscription expansion is illustrated in Figure 8.3. The expansion of subscription S of Figure 8.2(b) is represented. Nodes "CD" and "DVD" below "media" are the possible occurrences of the first "*" in S, below "media". They refer to the same node in S and thus have the same class. In contrast, node Book refers to a different node in S (node "Book") and has a different class in E_S. Nodes "Actor" and "Director" below "DVD" are the possible occurrences of the third "*" in S, given that their parent occurred (probability of $\frac{1}{2}$). Hence, pattern /media/CD/Actor has a $\frac{1}{4}$ likeliness of occurring in a document that matches S. Similarly, nodes "Producer" and "Title" below "DVD" are the possible occurrences of the second "*" in S. In contrast, Node "Producer" below "CD" is the only possible occurrence of the same wildcard in S. Also, node "Author" is the only possible occurrence of the third "*" in S. This is due to the correlations between elements defined in the DTD of Figure 8.2(a), and will be explained shortly.*

8.4.2 Definitions

In the rest of this section, we use the same notations as in section 5.6.1: $r_{E_S} \rightarrow N$ denotes the single path pattern that comprises all the nodes from r_{E_S} to N in E_S. Similarly, $r_S \rightarrow u$ denotes the single path pattern from node r_S (subscription S's root node) to node u in subscription S.

We say that node N *matches* node u iff $label(N)$ is equal or less general than $label(u)$. Given the definition of $label(N)$, it comes that: N *matches* u iff $label(u)$ is a wildcard ($*$) or an ellipsis($//$), or otherwise if $label(N) = label(u)$. Also, we say that $r_{E_S} \rightarrow N$ *matches* $r_S \rightarrow u$ iff a document that would only consist of the tree of element names $r_{E_S} \rightarrow N$ and that verifies the node predicates, *matches* the tree pattern $r_S \rightarrow u$. We subsequently say that $r_{E_S} \rightarrow N$ represents a possible *occurence* of the single path tree pattern $r_S \rightarrow u$.

Algorithm 26 Expand recursive function: $expand(u, N)$

```
1: if u is not // or * then
2:   Create node N_1 and insert as N's child
3:   label(N_1) ← label(u)
4:   prob(N_1) ← 1
5:   Increment highest_class_at_N
6:   class(N_1) ← highest_class_at_N
7:   for all u_i ∈ children(u) do
8:     expand(u_i, N_1)
9:   end for
10: end if
11: if u is * then
12:   Increment highest_class_at_N
13:   for all possible occurrences e_i of node u do
14:     Create node N_i and insert as N's child
15:     label(N_i) ← e_i
16:     class(N_i) ← highest_class_at_N
17:     Adjust prob(N_i)
18:     for all u_j ∈ children(u) do
19:       expand(u_i, N_i)
20:     end for
21:     if some u_j ∈ children(u) could not be expanded then
22:       Delete tree(N_i) from E_S
23:     end if
24:   end for
25: end if
26: if u is // then
27:   expand(children(u), N)
28:   if depth(N) ≥ Δ or probability(r_{E_S} → N) ≤ τ then
29:     return
30:   end if
31:   for all possible element e_i child of N do
32:     Create node N_i and insert as N's child
33:     label(N_i) ← e_i
34:     class(N_1) ← highest_class_at_N
35:     Adjust prob(N_i)
36:     expand(u, N_i)
37:     if some u_j ∈ children(u) could not be expanded then
38:       Delete tree(N_i) from E_S
39:     end if
40:   end for
41: end if
```

8.4.3 Building subscriptions expansions

Expand recursive function

In this paragraph, we describe the *expand* function which builds the expansion E_S of a subscription S. The expand function works recursively on the nodes of S. For each such node u, the expand function points to a node N in E_S, such that the single-path tree of element names $r_{E_S} \rightarrow N$ is a possible occurence of the single-path pattern $r_u \rightarrow parent(u)$, where $parent(u)$ is u's ancestor in S. That is, $r_{E_S} \rightarrow N$ is a possible tree of element names in a document that matches $r_u \rightarrow parent(u)$. Then, the recursive function $expand(u, N)$ determines the possible trees of element names that start with $r_{E_S} \rightarrow N$ and that are possible occurences of the pattern $r_u \rightarrow tree(u)$.

For that purpose, we first have to find the possible occurences of the pattern $r_u \rightarrow u$ given that $r_{E_S} \rightarrow N$ is the possible current occurence of $r_u \rightarrow parent(u)$. It is then equivalent to find all the nodes N_i such that $r_{E_S} \rightarrow N \rightarrow N_i$ is a possible occurence of $r_u \rightarrow parent(u) \rightarrow u$.

Then, because of the definition of E_S, it follows that if $label(u) \neq *$ and $label(u) \neq //$, then a single node N_1 is built in E_S and appended as child of N. It has label $label(u)$ and probability 1 (lines $2-4$). Indeed, node N_1 is the only possible occurrence of node u. Its class is such that no other children of N have the same class (for that purpose, a counter is maintained and incremented at node N) (lines $5-6$). Then, $r_{E_S} \rightarrow N \rightarrow N_1$ being the current possible occurrence of $r_u \rightarrow u$, we proceed recursively to find the possible occurrences of $r_u \rightarrow u_i$ for each u_i child of u (lines $7-9$).

Now if $label(u) = *$, then a set of nodes $\{N_i \in children(N)\} = A$ with the same class is appended as N's children. Each node N_i corresponds to a possible occurrence of node u (lines $14-16$). Its probability is adjusted (line 17) to represent the probability that it occurs, that is, that $r_{E_S} \rightarrow N_i$ actually occurs given that $r_{E_S} \rightarrow N$ has. The determination of the set A and their probabilities is based on either the knowledge of the DTD and/or a synopsis of previous documents.

1. If the DTD is known, then set A corresponds to the set of the possible children of the element with name $label(N)$ in the DTD. In other words, all the nodes N_i such that $r_{E_S} \rightarrow N \rightarrow N_i$ is a *valid* tree of element names.
   - If we have no synopsis, the probability is computed as 1 over the number of possible children. That is, we suppose that the distribution of element names in documents is uniform.
   - Otherwise, we use the synopsis to compute the distribution of the elements based on the previous documents. This can be done easily by parsing the synopsis against E_S. Each node in the synopsis contains the number of documents that had the single-path tree of elements $E_S \rightarrow N$ Note that if the synopsis does not contain the pattern $E_S \rightarrow N$, then the process will stop here and N will have no children. Indeed, if the synopsis is large enough, we estimate that no documents can have a pattern equal or longer than $E_S \rightarrow N$.
2. If the DTD is not known, then we also use the synopsis to determine set A. We proceed as in the case of the DTD except that the synopsis is parsed recursively against E_S. We then proceed as previously explained for the estimation of the nodes probabilities.

Finally, for all possible occurrences $r_{E_S} \rightarrow N \rightarrow N_i$ of $r_u \rightarrow u$, we proceed recursively to find the possible occurrences of $r_u \rightarrow tree(u)$ (lines $18-20$). Note that, although $r_{E_S} \rightarrow N \rightarrow N_i$ is a possible occurrence of $r_u \rightarrow u$, it may happen that we cannot find possible occurrences of $r_u \rightarrow tree(u)$ that start with $r_{E_S} \rightarrow N \rightarrow N_i$. We then delete all the nodes that belong to $tree(N_i)$ (lines $21-23$).

Now if $label(u) = //$, then we simply proceed according to the definition of the ancestor/descendant operator $//$. That is, node u is first mapped with "nothing". Then, $r_{E_S} \rightarrow N$ is a possible occurrence of $r_u \rightarrow u$, and we then directly proceed recursively with its child to find the possible occurrences of $r_u \rightarrow tree(u)$ that start with $r_{E_S} \rightarrow N$ (line 27). Second, node u is mapped with a node with any label and child u. Then, the procedure is similar to the case where $label(u) = *$: all nodes N_i such that $r_{E_S} \rightarrow N \rightarrow N_i$ is a *valid* tree of element names are possible occurrences of $r_u \rightarrow u$ (lines $32-35$). We then proceed recursively to find the possible occurrences of $r_u \rightarrow tree(u)$ (lines 36), and we remove nodes that yield no solutions (lines $37-39$).

Note that although not described in the algorithm, element values are handled quite easily as follows: each time a node or a set of nodes that match node u are appended as node N's children, we copy the predicate values of node u at each of those nodes. Also, note that as previously mentioned, in the case where $label(u) = //$, E_S can be infinite if there are loops in the DTD. To prevent this from happening, we simply stop the procedure if node N has exceeded a given predefined depth Δ or if the probability of path $r_{E_S} \rightarrow N$ is smaller than a predefined minimum probability τ (lines $28-30$).

Determining correlation groups

Finally, if we have knowledge of the DTD, the list of correlations groups for each node N in E_S is built. Note that this phase occurs after E_S has been built with the *expand* function previously

described. This is due to the fact that until the whole E_S has been built, nodes in E_S are not permanent (since subtrees may be pruned).

For that purpose, we proceed iteratively with each child node N_i of node N. If a correlations group already exists that contains N_i, we proceed with the next node. Otherwise, we use our DTD parser to determine the nodes $N_j \in children(N)$, that are in opposition with N_i and with each others. We then append a correlation group that contains those nodes and node N_i, to node N.

Note that the determination of the correlation groups at node N may result in some nodes being deleted, along with all their descendants. This happens when a node $N_i \in children(N)$ has probability 1 and belongs to a correlation groups. Then, all the other nodes N_j in that correlations groups must be deleted, along with their descendants. Indeed, node N_i has probability 1, which means that it *must* occur. Hence, the other nodes N_j *cannot* occur and have to be removed from E_S, with their descendants. Note that if some N_j also has probability 1, then the structure $r_{E_S} \rightarrow N$ does not yield to a correct occurence and node N itself, with all its descendants, have to be pruned. Then, if node N have probability 1, then its parent also has to be pruned, and so on (unless subscription S is invalid, the process stops and does not yield to the deletion of the whole E_S).

Also, note that if we do not have knowledge of the DTD, then we have still exploited, to some extent, the correlations between elements via the *synopsis*. Indeed, this latter contains structural information about documents that were seen previously. Those documents are *valid* documents, and are conform to the correlations specified in the DTD. Hence, when we built the subscriptions expansions according to the synopsis, we used the structures encountered in the synopsis to determine the possible occurences of the nodes in S. Hence, we implicitly exploited some elements correlations.

Example 6. *Consider again the example illustrated in Figure 8.3. Node "Producer" below "CD" is the only possible occurrence of the second "*" in S. Hence, it has probability* 1*. In addition, elements "Producer" and "Interpreter" are in opposition in the DTD of Figure 8.2(a). As a consequence, node "Author" becomes the only possible occurence of the third "*" in S, since "Interpreter" cannot appear simultaneously with "Producer". In addition, elements "Producer" and "Director" under "DVD" are in opposition according to the DTD. Consequently, node "DVD" in* E_S *contains a correlation group* $CGs = \{1, 4\}$ *(indicated next to it in Figure 8.3), that indicates that node number* 1 *below "DVD", i.e. "Producer", and node number* 4*, "Director", are in opposition.*

8.5 Similarity function: principle

In this section, we present the similarity function that we implemented for XML documents and XPath tree patterns. As the details of the complete algorithm are rather intricate, we only explain in this section the principle of the similarity function in simple terms, and illustrate it with some examples. This description is sufficient for a global understanding of the similarity function.

8.5.1 Principle

The similarity function $Sim(S_1, S_2)$ works recursively on the nodes of S_2 and the nodes of S_1's expansion, E_{S_1}. Recall that $Sim(S_1, S_2)$ returns the probability that a document matching S_1 also matches S_2, and is *not* symmetric.

The principle of the similarity function is pretty similar to that of the containment algorithms that we presented in Section 5 and the one in [34]. In fact, consider a document D that matches S_1. We want to evaluate the probability that it matches S_2. The idea is to use S_1's expansion rather S_1. Indeed, E_{S_1} represents all the possible structures (with the associated contents) that a document that matches S_1 can have, as well as the probabilities of each structure. Hence, the principle is to find paths in E_{S_1} that are equivalent or contained by a given path in S_2. We then say that the path in E_{S_1} *matches* the path in S_2. We proceed with as many paths in S_2 as possible (if the "entire" tree S_2 is found, then $S_1 \subseteq S_2$ and $Sim(S_1, S_2) = 1$), and when we have several possible paths in E_{S_1}, they are redundant and we compute the maximum of their probabilities. In contrast,

the conjunction of different patterns in S_2 is computed as the product of the probabilities of their respective occurences in E_{S_1}. Hence, we first look for the longest single-path patterns in S_2 that are matched in E_{S_1}. If the remaining parts are not matched, we use a heuristics to estimate the probability that they are matched. We then compute the probability that the conjunction of those patterns is matched. Besides, to two different patterns A and B in S_2 must correspond two different matching patterns C and D in E_{S_1}, in the sense that those patterns correspond to different patterns in S_1. Because of the definition of a subscription's expansion, it is equivalent to say that all nodes in C and D that have the same ancestor must belong to different classes. In the case where C and D do not correspond to different patterns in S_1, we say that there is a *conflict*. Indeed, patterns A and B are in conflict since they are matched with the same pattern in S_1, or at least with a common sub-pattern. Each time a node in C and a node in D have the same ancestor and the same class, there is a *conflict*. Intuitively, the higher the number of conflicts, the less S_1 is similar to S_2. Hence, each *conflict* yields to a *conflicts penalty* in the computation of the similarity between S_1 and S_2.

8.5.2 Examples

Consider XPath subscription S and its expansion, illustrated in Figures 8.2(b) and 8.3, respectively. Consider the following patterns:

- $S_1 = /media/DVD/Actor/First[text() \succ \text{“}B\text{”}]$
- $S_2 = /media/DVD/Actor/First[text() =\!\!'' \ Charles\text{”}]$
- $S_3 = /media[./Book]/CD$
- $S_4 = /media/*/Author$
- $S_5 = /media/CD/Producer[text() = \text{“}AB \ Productions\text{”}]$
- $S_6 = /media[./CD]/DVD$
- $S_7 = /media/DVD[./Title]/Director$
- $S_8 = /media/DVD[./Producer]/Director$
- $S_9 = /media/DVD[./Title]/*/First$
- $S_{10} = /media/DVD[./Producer]/*/First$

The similarity between those subscriptions and subscription S will be computed as follows:

$\mathbf{Sim(S_1, S)}$ will be computed as $\frac{1}{4} = \frac{1}{2}.\frac{1}{2}.1$. Indeed, the path $/media/DVD/Actor/First$ in E_S is the only one that matches S_1 (note that predicate $text() =\!\!'' \ Bob\text{”}$ matches $text() \succ'' \ B\text{”}$). Hence, the similarity between S_1 and S is computed as the probability that pattern $/media/DVD/Actor/First$ in E_S occurs, that is, $\frac{1}{4}$.

$\mathbf{Sim(S_2, S)} = 0 = \frac{1}{2}.\frac{1}{2}.0$. Indeed, the whole pattern is matched, but not the predicate at element "First" (predicate $text() =\!\!'' \ Charles\text{”}$ is not matched by $''Bob\text{”}$).

$\mathbf{Sim(S_3, S)} = \frac{1}{2} = 1.\frac{1}{2}$. Patterns $/media/Book$ and $/media/CD$ are matched with the equivalent patterns $/media/Book$ and $/media/CD$ in E_S. Since nodes "Book" and "CD" are in different classes, each node independently has a probability of $\frac{1}{2}$ of occurring. The similarity is computed as the probability that both patterns occur, that is, the product of their probability.

$\mathbf{Sim(S_4, S)} = 1 = Max(1.1, \frac{1}{2}.\frac{1}{2})$. Pattern $/media/*/Author$ is matched with two possible redundant patterns in E_S, $/media/CD/Author$ and $/media/Book/Author$. The similarity is computed as the maximum probability of occurrence of any of them.

$\mathbf{Sim(S_5, S)} = \frac{1}{2}.1.\epsilon$, with $0 < \epsilon \ll 1$. Pattern $/media/CD/Producer$ is matched but not the predicate at element $''Producer''$. However, it may be matched, in the sense that a document that

matches the predicate $'' \succ AB''$ might also match $'' = ABProductions''$. The probability that this occurs is estimated with a heuristics as ϵ.

$\mathbf{Sim(S_6, S)} = \frac{1}{2}.\frac{1}{2}.\gamma$, with $0 < \gamma \ll 1$. Here, patterns $/media/CD$ and $/media/DVD$ are matched with patterns $/media/CD$ and $/media/DVD$ in E_S that correspond to the same pattern in S ($/media/*$). Hence, we have a conflict and the similarity is computed as the product of the probability of occurence of patterns $/media/CD$ and $/media/DVD$ plus a conflict penalty γ.

$\mathbf{Sim(S_7, S)} = \frac{1}{2} = \frac{1}{2}.(\frac{1}{2}.\frac{1}{2})$. This case is similar to the computation of $\mathbf{Sim(S_3, S)}$. Pattern $/media/DVD$ has a probability of $\frac{1}{2}$ of occurring. Nodes $Title$ and $Director$ are in different classes and each independently has probability $\frac{1}{2}$ of occurring. The similarity is computed as the probability that both patterns occur, i.e., the product of their probability.

The last three subscriptions $S_8 \cdots S_{10}$ illustrate how correlations between elements are taken into account in the computation of subscriptions similarities.

$\mathbf{Sim(S_8, S)} = 0 = \frac{1}{2}.0$. Pattern $/media/DVD$ has a probability of $\frac{1}{2}$ of occuring, but nodes "Producer" and "Director" cannot appear simultaneously, as indicated in the correlation groups CGs of node "DVD". Hence, the similarity is computed as 0. Note, that S_8 is *not* a valid expression, in the sense that it matches no documents.

$\mathbf{Sim(S_9, S)} = \frac{1}{4} = \frac{1}{2}.(\frac{1}{2}.(\frac{1}{2} + \frac{1}{2}))$. Pattern $/media/DVD$ has a probability of $\frac{1}{2}$ of occurring. Node "Title" has a probability of $\frac{1}{2}$. The sub-pattern $/*/First$ has a probability of $\frac{1}{2}.1 + \frac{1}{2}.1$, since both nodes "Actor" and "Director" match the "*" (and element First below is also matched). Hence, pattern $/media/DVD[./Title]/*/First$ has probability $\frac{1}{2}.(\frac{1}{2}.1)$ of occurring.

$\mathbf{Sim(S_{10}, S)} = \frac{1}{8} = \frac{1}{2}.(\frac{1}{2}.\frac{1}{2})$. This case is similar to the previous one, except that nodes "Producer" and "Director" are in opposition. Consequently, the "*" in S_{10} can only be matched by node "Actor", since both "Producer" and "Director" may not appear simultaneously. As a result, sub-pattern $/*/First$ only has a probability of $\frac{1}{2}$ of occurring, as opposed to 1 in the previous case. The similarity is subsequently computed as $\frac{1}{8}$. Consequently, the notion of element correlations improves correctness, since without considering correlations, $Sim(S_{10}, S)$ would have been computed as equal to $Sim(S_9, S)$, which is not correct.

8.6 Conclusion

The proximity metric presented in this chapter is very general. Although used with XPath expressions and XML documents, it can be used with different tree structured languages. Also, although used in the context of pub/sub, it can be used to address different data management problems. This work is still in the development stage, and there is room for improvements and refinements. However, preliminary results shown in the experimental evaluation of Section 9.1.3 are very promising, they are of interest in their own right, and can prove useful in other domains.

Chapter 9

Experimental evaluation

In this chapter, we present results from experimental evaluation. We first assess the performance of the pub/sub system that we presented in Chapter 7, when using both the containment and similarity metrics. We then evaluate specifically the proximity metric based on subscriptions similarities for XML documents and XPath expressions that we presented in Chapter 8. That proximity metric was used in our pub/sub system to efficiently organize peers based on the similarities of their subscriptions. However, the proximity metric can be used to address different data management problems. The evaluation that we propose here is "general purpose" and is independent of the Pub/Sub context. Finally, in this chapter, we also survey related work.

9.1 Performance of the P2P semantic overlay

In this section, we present the simulations that we conducted to test the behavior and the effectiveness of our pub/sub system. We propose an evaluation of our system when using both the containment and similarity metrics presented earlier. We are mostly interested in studying the routing process in the system. Indeed, we have seen that the cost for its extreme simplicity is that it induces a certain inaccuracy in terms of false positives and negatives but than an efficient topology enables to minimize their occurrence. We thus aim at quantifying the accuracy of our system experimentally.

In addition, we also analyze the characteristics of the system obtained, especially in terms of number of connections. Finally, we evaluate the efficiency of peers management algorithms, when applicable.

To evaluate our system when using the similarity metric, we have implemented a proximity metric for XML documents and XPath subscriptions as explained in Chapter 8.

9.1.1 Experimental setup

Peers in our system register their interests using the standard XPath language to specify complex, tree-structured subscriptions. We generated set of various sizes as described in Section 6.2.2, with the parameters indicated in table 9.1.

Parameter	**Value**
h	10
d	3
$p_{//}$	0.1
p_*	0.1
p_λ	0.1
m	3
θ_S	-1
x	0 (with duplicates)
N	$1000 \rightarrow 50000$ (number of generated subscriptions)

Table 9.1: Parameter values of XPath subscriptions.

The events published by the producer are XML NITF documents generated with the values of the parameters as indicated in table 9.2.

Parameter	Value
L	20
T	$\{22, 58, 108\}$ tag pairs
r	3
θ_D	0 (uniform)

Table 9.2: Parameter values of XML documents.

9.1.2 Containment metric

We first focus on the system when peers are organized in a containment hierarchy according to the proximity metric f_c. To evaluate the efficiency of the system, we proceeded as follows.

We first simulated networks of different sizes, with each version of the *join* algorithm presented in section 7.3.4 by sequentially adding peers with randomly-generated subscription. We used a value of $L = 100$ for the maximum degree of equivalence trees. Also, for the *adaptive periodic joining* algorithm, we used a connections limit of $L_{APJ} = 100$ and a reorganization rate of γ_{APJ} of 10%.

Routing efficiency

We first focus on the performance of the system in terms of routing. We have seen that the containment tree topology enables to suppress all occurrences of false negatives. As a consequence, we aim at quantifying experimentally the number of false positives generated by the routing process in the system. For that purpose, we proceeded as follows. In a network of a given size and generated with a given join algorithm (as previously explained), we routed $1,000$ random documents by injecting them at the root node.[1] For each document, we computed the false positives ratio as the percentage of peers in the system that received a message that did not match its interests. The results were obtained by taking the average of 1000 executions and are shown in figure 9.1, for documents of average size $T = 22$ and in Figure 9.2 , for documents of size $T = 108$.

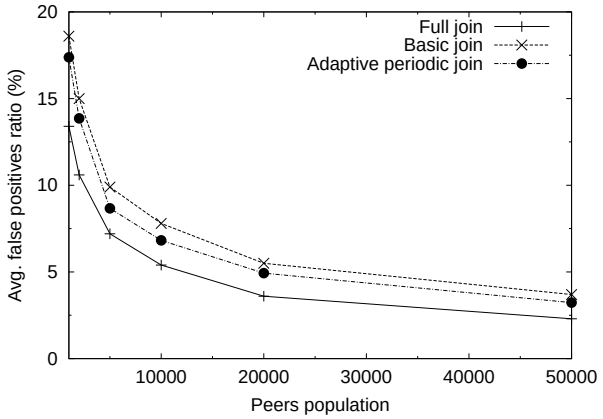


Figure 9.1: False positive ratio for networks of different sizes and small documents (22 tag pairs).

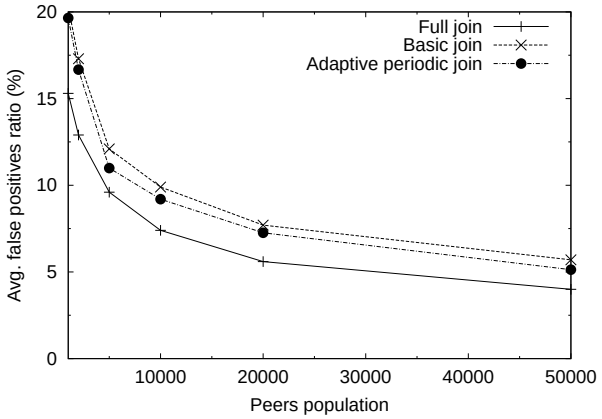


Figure 9.2: False positive ratio for networks of different sizes and large documents (108 tag pairs).

We first observe that the average false positives ratio remains small, typically less than 10% in most cases. This shows that our system delivers documents to all interested peers with only a very small fraction of false positives, that is, with good routing accuracy. Moreover, we computed that the

[1]Note that the number of false positives would not be affected when injecting the messages at another node than the root.

percentage of uninterested peers is 75%, on average and independently of the consumer population. When related to the false positives ratio, this shows that our system succeeds in delivering documents to the integrality of a rather small portion of interested consumers while delivering them to only a small fraction of a large population of uninterested consumers. Also, it illustrates the benefits of our routing protocol over a broadcast, which would yield false positives ratios of approximately 75%. In addition, we observe that the average false positives ratio decreases exponentially with the size of the consumer population, which means that the routing accuracy improves as additional peers join the system. These excellent results are due to the efficiency of the tree topology. By organizing peers based on their interests, documents are filtered out as soon as they reach the boundary of the community of interested consumers. The efficiency of the tree topology improves with the size of the consumer population because of the increasing number of containment relationships shared between the peers.

Unsurprisingly, join algorithms that reorganize the peers more frequently produce network topologies that have lower false positive ratio. As explained in Section 7.3.4, this is directly related to the number of reorganizations that are performed by each algorithm. However, the differences are very small and the benefits of the slight increase in accuracy may not justify the additional overhead of the reorganization process.

Finally, the results obtained with documents of large sizes are slightly higher. This is due to the fact that the larger a document is, the higher the number of peers that are interested in it, and especially, the higher the number of more selective peers that are interested in it. Those peers have higher depths in the tree topology. Thus the larger a document, the deeper and wider its spanning tree is in the tree topology, and the higher the chances to have false positives.

Network characteristics

We now analyze the main characteristics of the networks of various sizes, obtained with each version of the *join* algorithm. We are mainly interested in studying the degree of the peers in the system. Note that in all the networks generated , the root node was always "artificial", i.e., not a real peer with a registered subscription. As previously mentioned, the root node enables to interconnect top-level peers that share no containment relationships with each others, and can be implemented by simply connecting top-level peers with each other through "sibling" links. Consequently, that node was excluded from all the following experiments.

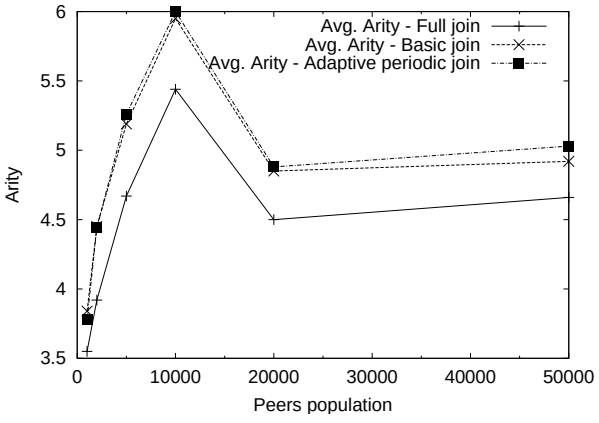


Figure 9.3: Average peer's degree (leaves excluded)

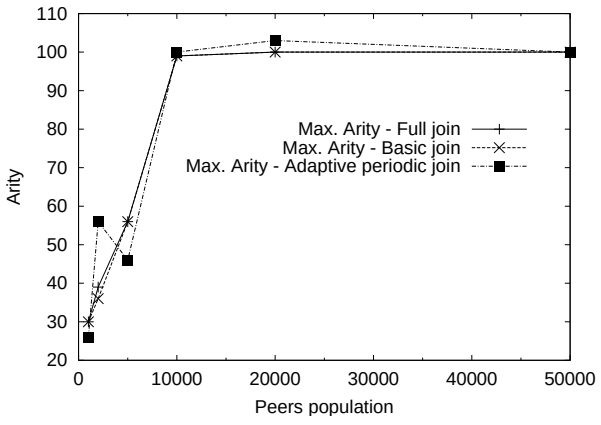


Figure 9.4: Maximum degree reached (root node excluded)

Figure 9.3 shows the average degree of the peers in the system when excluding those that are leaves in the tree hierarchy. We first observe that the average number of connections remains quite low, typically around $5 - 10$. We also observe that the average degree starts growing very fast then goes down to an intermediate value, before growing again slowly. The fact that the curve obtained with the *basic join* join mechanism also follows this evolution proves that the reorganization phase is

not responsible for this behavior. Rather, it is due to the fact that when new peers join an existing system consisting of a small population, there are not many containment relationships between the new peers subscriptions and the subscriptions of the existing peers. Thus the new peers do not find parents other than the most "general" peers (peers with the most general subscription, which have small depths) and the average number of connections increases very fast. When the population starts growing, there are more containment relationships and some new peers connect to more selective peers, which were leaves in the tree hierarchy and that are now taken into account to compute the average degree. Now, when the population grows more, the leaves become too selective. The average peer's degree starts increasing again but at a much lower rate, since there are more internal (non leaf) peers. Finally, we observe that the networks obtained with the *full join* mechanism have a lower degree, on average, than the two others. This is due to the fact that the *full join* algorithm performs connections balancing more efficiently. In addition, the *full join* algorithm performs more reorganizations than the other two algorithms. Indeed, very selective peers that joined the system early have low depths in the tree hierarchy. They are bad candidates for being the parents of new joining peers. If they are reorganized, their depth is increased and they have higher chances to have children.

Figure 9.4 shows the maximum degree reached by the most loaded peer in the system. The evolution of all three curves is directly dependent of the maximum size of an equivalent tree in the system. Indeed, when the population increases, the number of equivalent peers also increases. The result is that equivalence trees grow, and eventually, one has formed with more than two levels, and hence the root of that equivalence tree has reached the maximum degree allowed. The fact that the maximum degree is not exceeded is due, for one part, to the connections balancing technique employed in all three algorithms (new peers connect to the best peer with lowest degree), and for the other part, to the mechanism that was explained in Section 7.3.4 to prevent roots of equivalence from being overloaded. We observe that in the case of the *adaptive periodic join* algorithm, the maximum degree is occasionally exceeded, but the algorithm adapts by trying to reorganize the children of the peers that have exceeded the limit.

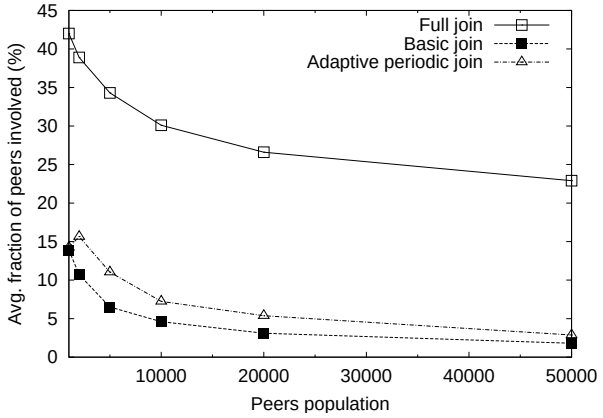


Figure 9.5: Average fraction of peers involved in a join process

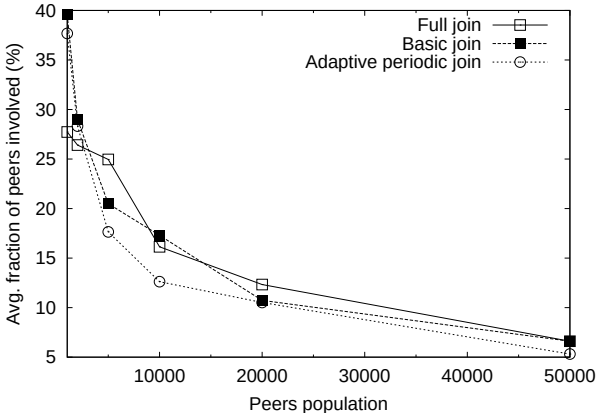


Figure 9.6: Average fraction of peers involved in a full leave process for children of departing peer to re-join the network.

Efficiency of peers management

To assess the performance of peers management, we have measured the average fraction of the peers population that is involved in a join or leave process. Indeed, we have seen that in both cases, the network must be probed recursively so as to find adequate containment relationships. For that purpose, we proceeded as follows. In networks of different sizes generated with each version of the join algorithm, we simulated the joining of 1000 new random peers, with the same version of the

join algorithm as the one that was used to generate it, and starting at the root node of the network (joining here has the highest overhead). After each join, we cancelled it to maintain a stable peers population during the whole experiment (we saved the system state before the join and restored it subsequently). We measured for each join the fraction of the peers that were involved in the process, i.e., that received and processed an instance of the *JOIN* message, and computed the mean value. To study the cost of peer departures, we proceeded similarly except that, for each of the 1000 measurements, we simulated the departure of a random peer. The *full leave* mechanism was used to maintain the most accurate topology. That is, we performed all necessary peers reinsertions in the tree topology, as explained in Section 7.3.4. Finally, to evaluate the real cost of the procedure, we excluded peers that are leaves in the tree topology and those that belong to equivalence trees, whose departure induces no or negligible overhead (but root of equivalence trees were included). Results are shown in Figures 9.5 and 9.6.

Finally, we evaluate the *partial leave* mechanism, which consists in reconnecting the children of the leaving peer at its parent. We have seen that this algorithm has no overhead on the global system, but that routing accuracy may degrade over time. Consequently, we have measured the loss of routing accuracy, in terms of false positives, after a given fraction r of the system has left consecutively. We used documents of size $T = 22$ tag pairs. We proceeded with networks with an initial size of $50,000$ peers, generated with each version of the join mechanism. Results are shown in Figure 9.7.

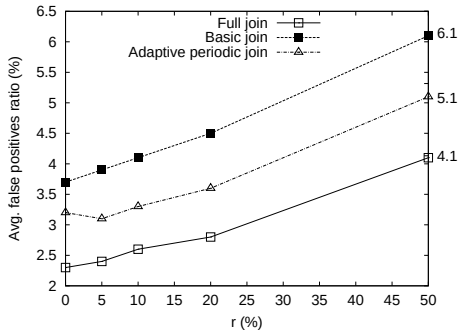


Figure 9.7: Average false positives ratio after a certain fraction $r\%$ of the population has consecutively left the system with the *basic leave* mechanism.

Join mechanism. We first focus on the case of the joining algorithm. Results are shown in Figure 9.5. We first observe that the fraction of peers involved in a join process remain reasonably small (except for the *full join* mechanism) and scales well to large consumer population. This is due to the efficiency of the tree topology which increases with the consumer population. Indeed, the higher the number of peers, the more organized they are in the tree topology, which avoids unnecessary propagation of *JOIN* messages further in the tree. We then outline that the *full join* mechanism performs significantly worse than the two others, the fraction of peers involved is several times higher. This is due to the fact that the *full join* mechanism performs all possible reorganizations , which necessitates *JOIN* messages to be propagated much further in the tree topology (even if the subscription of the new peer has no containment relationships with that of the current peer, as in lines $19 - 23$ in Algorithm 22). In comparison, the *simple* and the *adapt* mechanisms perform significantly less reorganizations and involve less peers in the process.

Full leave mechanism. We now focus on the evaluation of the *full leave* process. Note that the *basic leave* process involves very few peers and is the one that will be used typically. The *full leave* mechanism, in contrast, has significantly more overhead, and should only be used for small systems or when best accuracy is needed. Results are illustrated in Figure 9.6. We first observe that in both cases, the fraction of the system involved in a leave process remains reasonably small and decreases

with the consumer population. This is once again due to the efficiency of the tree topology: the more the consumer population, the more containment relationships there are, and the highest the chances that the children of the leaving peer are able to find another parent of same depth, and thus no more peers need to be re-inserted in the tree topology. Still, the fraction of peers that are involved in a leave process is significantly higher than that for a join (*full join* excluded). Indeed, a leave process requires at least that all the children of the leaving peers be reinserted with the join mechanism in the tree topology. In addition, as previously mentioned, additional peers among the descendants might need to re-evaluate their position as well if P's departure has decreased their depth. However, we have seen that the rejoining of those peers can be handled with the *Basic join* mechanism, which has the lowest overhead. Consequently, the fraction of the peers involved in a leave process remains small. This also explains that the overhead for leaving in the different networks (i.e., generated with different versions of the *join* algorithm) is similar.

Basic leave mechanism. We observe that, unsurprisingly, the false positives ratio increases when employing the *basic leave* algorithm, but at a moderate rate. It does not exceed 6% in all cases. Note that the accuracy improves as soon as some new peers join the system (unless with the *basic join* version), by having some peers that are no longer in their optimal place in the tree topology being reorganized optimally.

Conclusion

As a conclusion, the containment-based proximity metric f_c allows to build a bandwidth-efficient network topology that produces no false negatives and few false positives. In other words, the system delivers documents to all interested peers with only a small fraction of false positives.

The *full join* mechanism yields to the most accurate hierarchy, and to a well balanced topology, but at the cost of a significantly higher overhead due to the reorganization procedure. In fact, it appears that the slight increase in accuracy may not justify the additional overhead of the reorganization process. On the other hand, the *basic join* mechanism has the lowest overhead, but produces less accurate networks with poor load balancing properties. Finally, it appears that the *adaptive periodic join* mechanism reaches a good compromise between the *full* and the *basic join* join mechanisms. Indeed, it has an overhead almost similar to that of the *basic join* algorithm, but produces a well balanced topology and a routing accuracy very close to that of the *full join* mechanism.

9.1.3 Similarity metric

We now focus on the system when peers are organized in a graph according to the proximity metric f_s based on subscription similarities. The parameters of the experiments that we conducted are indicated in table 9.3.

Parameter	Description	**Value range**
n	connectivity (number of semantic neighbors)	$\{5, 7, 10\}$
r	number of random neighbors	1
ρ	random neighbor forwarding probability (%)	$\{0, 50, 100\}$

Table 9.3: Experimental parameters for organization based on similarity

The parameters of the data used in the experiments were indicated in Section 9.1.1, at the difference that we experimented with XML documents of size $T = 22$ and $T = 58$ (small and medium).

Our main focus is to evaluate the accuracy of the routing process in the system. We also analyze the main characteristics of the system in terms of number of connections. We do not assess the performance of peers management, since we have seen that *JOIN* messages are broadcast in (a reasonably large portion of) the whole system during a join or a leave process.

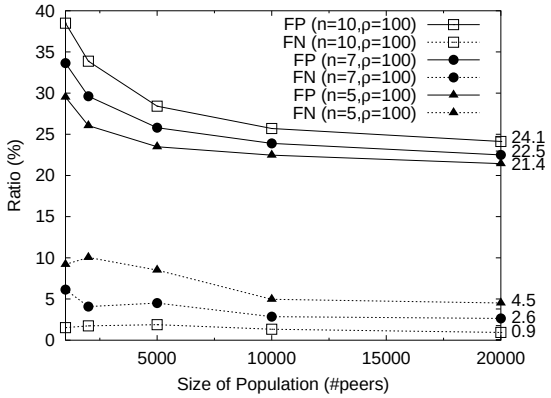


Figure 9.8: False positives and false negatives ratios for networks of different sizes and connectivities, for documents of size 22.

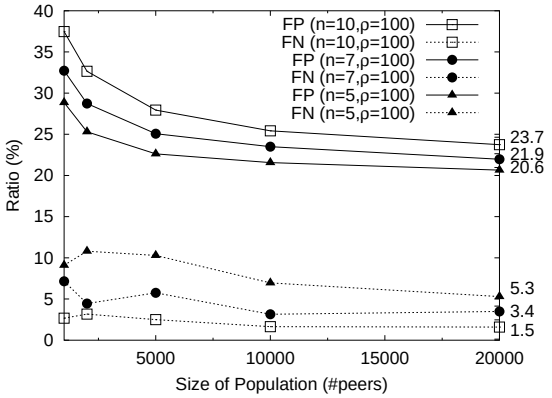


Figure 9.9: False positives and false negatives ratios for networks of different sizes and connectivities and for documents of size 58.

Routing accuracy

Our main focus is to evaluate the routing accuracy of the system. Since the topology based on similarity does not prevent the occurrence of false negatives, we are interesting in quantifying the accuracy of our system both in terms of false positives and false negatives. For that purpose, we proceeded as in the case of the metric based on containment. We first generated networks of different sizes, using different values of n for the number of proximity neighbors and a value of $r = 1$ for the number of random neighbors.

We then injected random documents of different sizes and quantified the routing accuracy. We measured the false positives ratio as the percentage of the peers in the system that received a message that did not match their interests, and the false negatives ratio as the percentage of peers interested in a message that did not receive it. To clarify, let N be the total population. Consider a document e that has been routed. Let FP be the number of peers not interested in e that received it. Then, the false positives ratio is computed as $\frac{FP}{N}$. Now let FN be the number of peers interested in e that did not receive it, and N_I be the total number of peers in the system interested in e. Then, the false negatives ratio is computed as $\frac{FN}{N_I}$. We computed the average values over $1,000$ runs. We experimented with random neighbor forwarding probabilities ρ of 100%, 50% and 0%. In particular the value $\rho = 0\%$ enables us to experiment with a network with no random neighbors, without having to re-generate it.

Figure 9.8 shows the false positives and false negatives ratios for documents of size 22, when varying the size of the population, and for different values of n. Figure 9.9 refers to documents of size 58. Figure 9.10 shows the false positives and negatives ratios for documents of size 22 and for a value of $n = 10$, when varying the size of the population and for different values of the random neighbor forwarding probability ρ. Figure 9.11 shows results with documents of size 58.

We first observe that the average false negatives ratio remains small, typically less than 5% and much less for networks of high connectivities. This shows that on average, for a given document, only a negligible fraction of the population of interested consumers does not receive it. In other words, our system succeeds in delivering documents to almost all interested consumers. The false positives ratio, while significantly higher, still remains at reasonable values, typically around 20%.

Those results illustrate the benefits of our routing protocol over a broadcast, which would yield false positives ratios of approximately 75% (the same as for the containment metric, since we used the same sets of XPath expressions and XML documents). In contrast, our routing protocol, which has almost the same complexity as a broadcast, achieves significantly lower false positives ratio and better bandwidth usage, at the cost of only 5% of false negatives, in average.

We also remark that, unsurprisingly, a higher connectivity favors the false negatives ratio over the false positives ratio. However, results do not differ significantly and remain quite comparable.

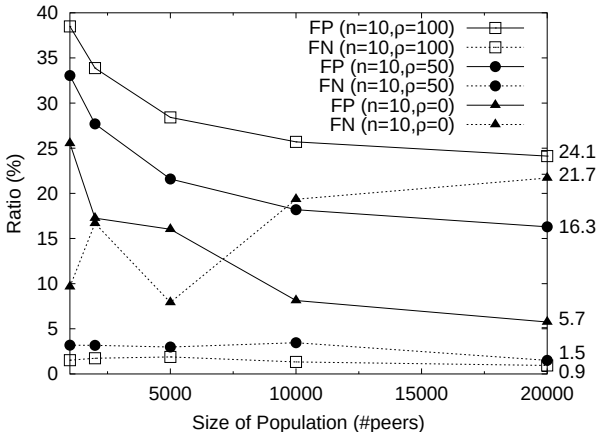


Figure 9.10: False positives and false negatives ratios for networks of different sizes and for different values of ρ, for documents of size 22.

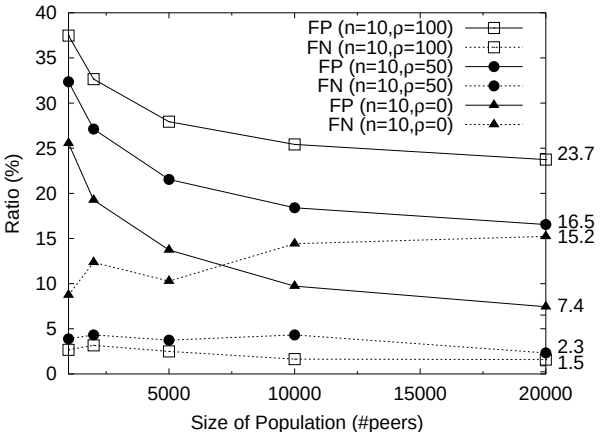


Figure 9.11: False positives and false negatives ratios for networks of different sizes and for different values of ρ, for documents of size 58.

In addition, we also observe that, as expected, the parameter ρ has the opposite effect: a lower value of the parameter ρ favors the false positives ratio over the false negatives ratio. For a value of $\rho = 50\%$, the false positives ratio improves significantly (30% lower, in average) at the cost of a slight increase of the false negatives ratio. In fact, it appears that if obtaining low false negatives ratios is not so vital, a value of $\rho = 50\%$ may be preferred since it enables to obtain a good false positives ratio (16% for $20,000$ peers). On the other hand, it appears that a network where peers have no random neighbors (i.e., $\rho = 0\%$), achieves a very poor false negatives ratio, which, even worse, increases with the consumer population. It achieves, however, excellent false positive ratio. As previously anticipated, this is due to the fact that random neighbors avoid the construction of disconnected semantic communities (because their interests do not compare with the other communities' interests). The phenomenon amplifies with the consumer population. Hence, in most cases, it is desirable that peers have at least one random neighbor.

Besides, we observe that the documents size does not seem to have a major impact on the routing accuracy. A higher size favors the false positives ratio over the false negatives ratio, but results remain comparable. That evolution is not obvious. We think, however, that it is due to the fact that with documents of higher sizes, there are more interested peers in the system. Then, it is more difficult to reach the population of interested peers, and consequently there are more false negatives. On the other hand, since there are more interested peers, documents are propagated to more peers (because of the routing algorithm). Hence, the average false negatives ratio does not increase significantly. Similarly, there are less non-interested peers, and hence, less false positives. However, the documents are propagated to more peers, which prevents the false positives ratio from improving significantly.

Finally, all performance metrics decrease with the size of the consumer population, which shows that the routing accuracy globally improves with the consumer population. This can be explained by the fact that, in larger populations, peers are able to find better neighbors according to the proximity metric f_s and hence reduce the occurrence of false positives and false negatives.

Networks characteristics

We now briefly analyze the main characteristics of the networks of various sizes and connectivities. Since peers are organized in a graph, we are mainly interested in the degree (number of neighbors) of the peers. Recall that each peer in the system has chosen a set of n proximity neighbors and r random neighbors, but can be in turn chosen by some other peers as a proximity or a random neighbor. Hence, the total number of neighbors that a peer has is necessarily higher than $n + r$. In

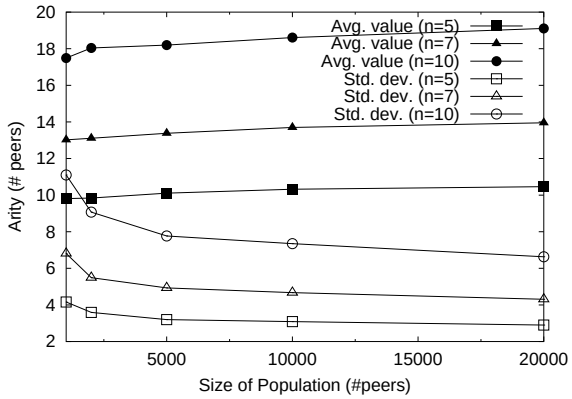


Figure 9.12: Mean value and standard deviation of the peers degree.

addition, as previously mentioned, it is not bounded, a priori. Hence, in each network generated with different values of n, we have measured the average value μ, the maximum value θ and the standard deviation σ of the total number of neighbors of the peers. Results are reported in table 9.4 and Figure 9.12.

	P	$1,000$	$2,000$	$5,000$	$10,000$	$20,000$
n=5	μ	9.81	9.84	10.11	10.32	10.46
	θ	57	70	99	106	133
	σ	4.16	3.59	3.20	3.09	2.90
n=7	μ	13.02	13.11	13.38	13.70	13.95
	θ	88	100	158	172	217
	σ	6.81	5.49	4.93	4.67	4.31
n=10	μ	17.49	18.04	18.2	18.61	19.11
	θ	155	147	217	276	317
	σ	11.10	9.07	7.77	7.35	6.63

Table 9.4: Average value μ, maximum value θ and standard deviation of the peers degree.

We first observe that the average number of connections increases with the size of the population but at a very low rate. This is due to the fact that, when the population grows, peers are able to find better neighbors according to the proximity metric f_s. Besides, the standard deviation has relatively low values and decreases with the consumer population. This shows that for large populations, the number of connections is more fairly distributed amongst the peers. This is again due to the fact that peers are able to find better neighbors when the population grows, as well as the connections balancing techniques that are used in the system.

Finally, we remark that the maximum number of connections increases with the consumer population, but at a moderate rate. Although it reaches relatively high values, in particular in networks with high connectivities ($n = 10$), we believe those are isolated cases, as indicated by the low values of the standard deviation. Besides, although not implemented in the system, it is possible for an overloaded peer to refuse additional connections.

Summary

The organization of peers according to the proximity metric f_s based on subscription similarities enables to build a well balanced and robust network topology that succeeds in delivering documents to almost all interested consumers with only a relatively small fraction of false positives. The system offers high scalability to large consumer populations, both in terms of routing accuracy and connections balancing.

Related Work

Many pub/sub systems use an overlay network of event brokers to implement some form of distributed content based routing, most notably IBM Gryphon [14], Siena [28], Jedi [44] and XNet [37]. As previously mentioned, these systems suffer from various limitations in terms of extensibility, scalability, and cost.

To address some of these issues, several attempts have been made to implement pub/sub systems based on P2P networks.

Scribe [33] and Bayeux [127] are two examples of topic-based pub/sub systems built on top of two overlay network infrastructures. Scribe is built on top of Pastry [104], whereas Bayeux leverages the architecture of Tapestry [126]. These systems provide application-level multicast services using the rendezvous-based communication model. The events and the subscriptions are automatically classified in topics, using an appropriate application-specific schema. They are then associated with an identifier that is a hash of the data content. Routing is then achieved by leveraging the direct routing capabilities of the underlying infrastructure. Basically, events are first sent via the rendezvous node, they are then distributed to interested subscribers. Those systems support large numbers of topics and perform efficient large-scale routing of message. Also, rendezvous nodes can be replicated for fault-tolerance. However, one limitation is that all events must be sent via the rendezvous node which can then become a potential bottleneck. Morever, those systems are incapable of supporting more sophisticated content-based operations.

More recently, some content-based pub/sub systems based on P2P networks have been proposed.

In [91], the authors combine the notion of rendezvous nodes and content-based multicast to implement content based routing in a P2P environment. Events are first guided to a rendezvous node before being disseminated along a multicast tree of interested subscribers. As previously mentionned, one limitation lies in the fact that all events must be sent via the rendezvous node which can become a potential bottleneck.

HOMED [40] is a P2P overlay for distributed pub/sub systems. Peers are organized in a mesh-like structure based on their interests, by assigning to each peer an identifier that represents its subscription. Peers are then organized in a logically binary hypercube according to their identifiers. Routing is achieved by propagating the event along a multicast tree embedded in the hypercube.

Hermes [95] is a distributed event based middleware platform that also use the concept of rendezvous nodes to avoid global broadcasts. Hermes implements a type and attribute based routing scheme that is built on top of a P2P overlay routing network and extends the expressiveness of subscriptions and supports event hierarchies. An event is first routed to the rendezvous node that correponds to its type. The direct routing capabilities of the underlying P2P overlay are used for that purpose (where the ID of the node is the hash of the event type name). The event is then diffused via a distribution tree rooted at the rendezvous node, setup as in the case of our XNet publish subscribe system or SIENA. Hermes also encompasses other functions such as subscription aggregation and fault-tolerance. However, one limitation is that all events must be sent via the rendezvous node, which can then become a potential bottleneck. Another limitation lies in the fact that nodes in a Hermes event hierarchy are likely to be dispersed throughout the network, which can result in large overhead and poor bandwidth usage when propagating an event to all of its descendants.

In [12], the authors address a major issue encountered by most pub/sub systems, which is their

reliance on a fixed infrastructure of brokers. They specifically address the issue that if the network topology has no relationships with the subscriptions registered by the consumers, then the routing process performs poorly: it often involves a large number of routers, and is then barely more efficient than a broadcast. For that purpose, they propose TMS, a self-organizing topology management system. Its principle is similar to that of our semantic P2P overlay. However, the TMS system does not implement a routing algorithm. Also, it is not based on the P2P paradigm. Rather, the authors consider an existing pub/sub system architectured as a network of event brokers, and propose TMS as an additional component of each broker in the system. TMS operates transparently and independently of a given content based routing algorithm, aiming at increasing its efficiency. For that purpose, it rearranges TCP connections between pairs of brokers so as to put consumers with similar interests as close as possible. This obviously enables to reduce the number of TCP hops (i.e. the number of brokers involved) for each event diffusion, and hence improve the overal efficiency and scalability of the system. Also, the system takes into account network-level metrics to ensure that a new rearrangement is not harmful in terms of network-level performance. Finally, the TMS system implements fault-tolerance mechanisms and allows routers to dynamically join and leave the network. One limitation of the TMS system is that each broker must maintain the full history of the last n received events, so as to estimate the proximity between brokers. This can be an issue if very large events transit in the system. In contrast, our proximity metric relies on a *synopsis* of events, which is a compact representation of a history, even with very large number of events. In addition, our proximity metric is perfectly capable of operating without any kind of information about the streamed events (history or synopsis), with only the grammar that is used to express events and subscriptions.

In [46] and [8], the authors propose a scalable protocol for the diffusion of information in P2P (or mobile) networks, which also provides anonymity and mobility for producers and consumers. In their approach, the system is modeled as a logical multi-layer system where each layer l, also referred to as the communication graph at layer l, is a directed acyclic graph. Topic-based pub/sub is then implemented by having each communication graph at layer l represent the pub/sub system for the topic l. Content-based pub/sub is an extension of the topic-based one. Consumers' interests are specified using a set of topics, each of them are then handled by the communication graph at the corresponding layer. This is one limitation of the system. In fact, it is not clear whether consumers interests are directly replaced by, or only mapped to a set of topics in the system (i.e., in the latter case, consumers keep their "real" interests and the system handles the associated set of topics). In the former case, the expressiveness provided to consumers is strongly limited, while the case of a mapping may result in poor bandwidth and processor usage.

Finally, some proposals have been made to implement content based routing on top of the Chord [113] P2P network.

In [114], event propagation and filter updates are similar to the broadcast mechanism proposed in [54], but are attenuated by the use of filters on the edges of the graph and by taking advantage of covering relationship.

In [116], the event schema is a set of typed attributes. Each node stores pieces of information regarding some subscriptions, which are called *subscription ids*. The main idea is to store a subscription id at the nodes of the graph selected by appropriately hashing the values of the attributes of the subscription. Routing is achieved by fetching the subscription ids of the nodes selected by hashing the values of the attributes in the event.

Chapter 10

Conclusions

The pub/sub paradigm has become a hot research topic in the last few years. Indeed, it offers a convenient abstraction for data producer and consumers, as most of the complexity related to addressing and routing is encapsulated within the network infrastructure. Most importantly, the strong decoupling that it allows in time, space and synchronization between the different participants in the system makes it well adapted to large scale distributed information systems.

In this thesis, we have focused on the research, the design and the implementation of pub/sub systems with the specific goal of achieving *scalable* and efficient diffusion of information.

10.1 Contributions

The contributions of this thesis can be summarized as follows:

The XNet XML Content Network. Although some prototypes have been developed by several researchers in the past, the issue of implementing an Internet-scalable pub/sub system remains a big challenge. Starting from this observation, we have designed the XNet XML content network to implement efficient and reliable distribution of structured XML content to very large populations of consumers. For that purpose, our system integrates several novel technologies. The routing protocol, XRoute, makes extensive use of subscription aggregation to limit the size of routing tables while ensuring perfect routing (i.e., an event is forwarded to a link iff it leads to a consumer interested in that event), even in the presence of subscription cancellation. The filtering engine, XTrie, uses a sophisticated algorithm to match incoming XML documents against large populations of tree-structured subscriptions. The XSearch subscription management algorithm enables the system to manage large and highly dynamic consumer populations, by efficiently determining the possible containment relationships between a given subscription and a potentially large set of subscriptions. Finally, our XNet system integrates *reliability* mechanisms to guarantee that its state is consistent with the consumer population. It implements several approaches to fault-tolerance to recover from various types of router and link failures by using the most appropriate scheme depending on various factors, such as the expected duration of the outage or application-specific availability requirements.

It is important to note that the XNet system is a perfectly operational prototype. Not only have we analyzed its efficiency by the means of various simulations, but we have also performed a large scale deployment in the PlanetLab testbed, so as to experiment with the conditions of the real Internet. Experimental results demonstrate that our prototype does not only offer very good performance and scalability under normal operation, but can also quickly recover from system failures.

Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks. To address some of the limitations of traditional pub/sub systems based on server overlays, we have explored a different and novel approach to building a pub/sub system based on the P2P paradigm. The producers and

consumers are organized in a peer-to-peer network that self-adapts upon peer arrival, departure, or failure. Most importantly, our pub/sub system features an extremely simple routing protocol that requires almost no resources and no routing state to be maintained at the peers. We have seen that routing may not be perfectly accurate, but that by organizing the peers in "semantic communities", i.e., by organizing them according to their interests with adequate proximity metrics, we can minimize the routing inaccuracy. We have proposed a containment-based proximity metric that allows to build a bandwidth-efficient network topology that produces no false negatives and very few false positives, and a proximity metric based on subscription similarities that yields a more robust graph structure with small false negatives ratios and very few false positives. Experimental results demonstrate that the routing process is indeed very accurate and highly efficient, and that our system features excellent scalability to large consumer populations, both in terms of routing and peer management overhead.

Finally, we have briefly presented the proximity metric based on subscription similarities that we developed to organize the peers in the aforementioned system in an efficient and robust graph structure. The metric evaluates the proximity between two given subscriptions in terms of filtering, i.e., it evaluates the error that would be induced by filtering XML documents against the other subscription instead of the original. For that purpose, we introduced the notion of a subscription's expansion, a data structure that represents all the possible constraints, on both the structures and the contents, that a document matching that subscription may contain. Besides, our proximity metric can operate in various conditions, that is, it adapts to the amount of knowledge that we have about the data. More precisely, if the *grammar* of the data or a *history* or *synopsis* of previous documents is known, the proximity metric exploits this knowledge to improve correctness and accuracy. In particular, a major innovation is that if the grammar is known, then the proximity metric makes use of the *correlations* between elements that are defined in it, so as to improve accuracy. The evaluation of the system in terms of routing shows that the proximity metric is highly accurate.

The proximity metric can be used in a context independent of the pub/sub paradigm. It is of interest in its own right, and can prove useful in other domains.

10.2 Discussions and future directions

XNet system. As previously mentioned, the XNet system is a perfectly operational prototype that we deployed and methodically evaluated in the PlanetLab testbed. Besides, it should be noted that most of the technologies that are integrated in the XNet system are both of a pragmatic and theoretic interest. Although used in the context of pub/sub, they can be used to address different networking or data management problems.

There are several directions for future work in the XNet system. In particular, we could use the "imperfect" aggregation (i.e., lossy) mechanisms presented in [34] to reduce even further the routing tables sizes in the system, under a given space constraint. The cost is that since aggregation is imperfect, it is likely that routing is not perfectly accurate, in the sense that events may be wrongly delivered, or may fail to be delivered. This problem is especially challenging if we aim at preserving routing accuracy in the presence of subscriptions cancellations. Besides, it would be very interesting to observe how routing accuracy is affected when related to the "degree" of compression applied to the routing tables.

Further, it would be interesting to extend XNet to support "semantic" languages, i.e., that considers the meaning of the data rather than just its type or content.

Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks. The implementation of a working prototype of our pub/sub system based on the P2P paradigm is still under progress. However, the communication protocols have been evaluated by the means of various simulations. Also, we expect that the extremely simple routing protocol used in our system should yield to an efficient implementation.

It should be noted that our pub/sub system based on P2P is very general. It can be used with any subscription languages and any proximity metrics. Besides, our scheme is highly applicable to ad-hocs networks (e.g., sensor networks) where resources are scarce, in terms of participants (CPU, memory), or network (bandwidth). In this context, the simplicity of our routing and joining mechanisms largely makes it for the induced routing inaccuracy.

As part of our ongoing research, we are studying refinements of our proximity metrics that take into account additional factors, such as physical proximity or link bandwidth, in order to minimize latency and maximize throughput.

As a possible direction for future work, we are studying the possibility to use more expressive and powerful grammars than DTDs, such as XML schemas (XSD), in our proximity metric. In particular, DTDs only allow to define rather limited correlations between elements, since they only come in the form of mutual exclusion or element quantifiers. In contrast, XML schemas enable to define more varied and expressive correlations.

Bibliography

[1] An history of the internet. http://academ.hvcc.edu/~hurdand/web01/HistoryLecture.htm.

[2] *Network News Transfer Protocol - A Proposed Standard for the Stream-Based Transmission of News*. Internet Request For Comments (RFC) 977, Internet Engineering Task Force, 1986.

[3] A. Aboulnaga, A.R. Alameldeen, and J.F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 591–600, San Francisco, CA, USA, 2001.

[4] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching Events in a Content-based Subscription System. In *Proceedings of PODC*, pages 53–61, Atlanta, GA, May 1999.

[5] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, New York, NY, USA, 1999. ACM Press.

[6] M. Altherr, M. Erzberg, and S. Maffeis. iBus - a software bus middleware for the java platform. In *Proceedings of the International Workshop on Reliable Middleware Systems*, pages 49–65, October 1999.

[7] M. Altinel and M.J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of VLDB*, pages 53–64, September 2000.

[8] E. Anceaume, A.K. Datta, M. Gradinariu, and G. Simon. Publish/subscribe scheme for mobile networks. In *POMC '02: Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 74–81, New York, NY, USA, 2002.

[9] S. Baehni, P. Eugster, and E. Guerraoui. Data-aware multicast. In *Proceedings of the 5th IEEE International Conference on Dependable Systems and Networks*, June 2004.

[10] R.A. Baeza-Yates and G.H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

[11] R. Baldoni, R. Beraldi, S.T. Piergiovanni, and A. Virgillito. Measuring notification loss in publish/subscribe communication systems. In *PRDC*, pages 84–93, 2004.

[12] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. A self-organizing crash-resilient topology management system for content-based publish/subscribe. In *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, Edinburgh, Scotland, UK, May 2004.

[13] R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito. Modeling publish/subscribe communication systems: towards a formal approach. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 304–311. IEEE, 2003.

[14] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of ICDCS*, May 1999.

[15] G. Banavar, M. Kaplan, K. Shaw, R.E. Strom, D.C. Sturman, and W. Tao. Information flow based event distribution middleware. In *ICDCS Workshop on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, 1999.

[16] K. Bennett and C. Grothoff. GAP – practical anonymous networking. In Roger Dingledine, editor, *Proceedings of Privacy Enhancing Technologies workshop (PET 2003)*. Springer-Verlag, LNCS 2760, March 2003.

[17] K. Bennett, C. Grothoff, T. Horozov, and I. Patrascu. Efficient Sharing of Encrypted Data. In *Proceedings of ASCIP 2002*, pages 107–120. Springer-Verlag, July 2002.

[18] S. Bhola, R.E. Strom, S. Bagchi, Y. Zhao, and J.S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 7–16, Washington, DC, USA, 2002. IEEE Computer Society.

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

[20] K. P. Birman and R.van Renesse. Reliable distributed computing with the isis toolkit. *IEEE Computer Society Press*, 1994.

[21] G. Bricconi, E. Tracanella, and E. Di Nitto. Issues in analyzing the behavior of event dispatching systems. In *IWSSD '00: Proceedings of the 10th International Workshop on Software Specification and Design*, page 95, Washington, DC, USA, 2000. IEEE Computer Society.

[22] G. Bricconi, E. Tracanella, E. Di Nitto, and A. Fuggetta. Analyzing the behavior of event dispatching systems through simulation. In *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*, pages 131–140, London, UK, 2000. Springer-Verlag.

[23] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, December 1998.

[24] A. Carzaniga, E. Di Nitto, D.S. Rosenblum, and A.L. Wolf. Issues in supporting event-based architectural styles. In *3rd International Software Architecture Workshop*, Orlando, Florida, November 1998.

[25] A. Carzaniga, D.R. Rosenblum, and A.L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *Engineering Distributed Objects '99*, Los Angeles, California, May 1999.

[26] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000.

[27] A. Carzaniga, D.S. Rosenblum, and A.L. 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.

[28] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

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

[30] A. Carzaniga, M.J. Rutherford, and A.L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, March 2004.

[31] A. Carzaniga and A.L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, Arizona, October 2001. Springer-Verlag.

[32] A. Carzaniga and A.L. Wolf. Forwarding in a content-based network. In *Proceedings of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, August 2003.

[33] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, October 2002.

[34] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proceedings of VLDB*, August 2002.

[35] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *VLDB Journal*, 11(4):354–379, 2002. Also appeared in Proceedings of ICDE, 2002.

[36] C-Y. Chan, M. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119, 2003.

[37] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of NCA*, Cambridge, MA, April 2003.

[38] R. Chand and P. Felber. Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks. In *EuroPar 2005*, September 2005.

[39] R. Chand and P.A Felber. XNet: A Reliable Content-Based Publish/Subscribe System. In *SRDS 2004, 23rd Symposium on Reliable Distributed Systems*, Florianopolis, Brazil, October 2004.

[40] Y. Choi, K. Park, and D. Park. Homed: A peer-to-peer overlay architecture for large-scale content-based publish/subscribe systems. In *Proceedings of DEBS*, Edinburgh, UK, May 2004.

[41] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, ICSI, Berkeley, CA, USA, July 2000.

[42] International Press Telecommunications Council. News Industry Text Format.

[43] G. Cugola, E. Di Nitre, and G. Picco. Content-based dispatching in a mobile environment. In *Workshop su Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi (WSDAAL)*, 2000.

[44] G. Cugola, E. Di Nitto, and A. Fugetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.

[45] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 261–270, Washington, DC, USA, 1998. IEEE Computer Society.

[46] A.K. Datta, M. Gradinariu, M. Raynal, and G. Simon. Anonymous publish/subscribe in p2p networks. In *IPDPS*, page 74, 2003.

[47] D.Cheriton and W.Zwaenepoel. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, 1985.

[48] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. In *TODS*, volume 28(4), pages 467–516, New York, NY, USA, 2003.

[49] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of ICDE*, San Jose, CA, February 2002.

[50] Y. Diao and M.J. Franklin. High-Performance XML Filtering: An Overview of YFilter. *IEEE Data Engineering Bulletin*, March 2003.

[51] A.L. Diaz and D. Lovell. *XML Generator*. http://www.alphaworks.ibm.com/tech/xmlgenerator, September 1999.

[52] C. Diot, B. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment Issues for the IP Multicast Service and Architecture. *IEEE Network magazine special issue on Multicasting*, February 2000.

[53] D.Powell. Group communication. *Communications of the ACM*, 39(4):50–97, April 1996.

[54] S. El-Ansary, L.O. Alima, P. Brand, and S. Haridi. Efficient broadcast in structured p2p networks. In *Proceedings of IPTPS03*, Berkeley, USA, February 2003.

[55] P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[56] P.Th. Eugster, R. Guerraoui, and Ch.H. Damm. From epidemics to distributed computing. *IEEE Computer, 37(5)*, pages 60–67, May 2004.

[57] F. Fabret, H.A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, and D. Shasha. Filtering Algorithms and Implementations for Very Fast Publish/Subscribe Systems. In *Proceedings of SIGMOD*, pages 115–126, Santa Barbara, California, May 2001.

[58] L. Fiege, G. Mühl, and F.C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.

[59] L. Garces, K. Ross, E. Biersack, P. Felber, and G. Urvoy-Keller. Topology-centric look-up service. In *NGC'03, 5th International Workshop on Networked Group Communications, September 16-19, 2003 - Munich, Germany*, Sep 2003.

[60] GNUnet. http://www.gnu.org/software/gnunet/.

[61] Gnutella. http://www.gnutella.wego.com/ (2000).

[62] T.J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 173–189, London, UK, 2002. Springer-Verlag.

[63] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Version 2.3. Object Management Group, Framingham, MA, USA, 1998.

[64] Object Management Group. *CORBA event service specification*. Version 1.01. OMG Document formal/2002-08-04, 2002.

[65] T.O. Group. *DCE 1.1: Remote Procedure Call.* Technical Standard C706. The Open Group, Cambridge, MA, USA, 1997.

[66] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Proceedings of The International Conference on Distributed Computing Systems, Workshop on Middleware*, Austin, Texas, 1999.

[67] A.K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2003.

[68] E.N. Hanson, C. Carnes, L. Huang, O. Konyala, L. Noronha, S. Parthasarathy, B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275, Sydney, Austrialia, March 1999.

[69] E.N. Hanson, M. Chaabouni, C-H. Kim, and Y-W. Wang. A predicate matching algorithm for database rule systems. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 271–280, Atlantic City, New Jersey, United States, May 1990.

[70] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proceedings of MobiDE*, pages 27–34, May 2001.

[71] I2P. http://www.i2p.net/.

[72] IBM. Gryphon: Publish/subscribe over public networks. Technical report, IBM T.J. Watson Research Center, 2001.

[73] P. Sens J-M. Busca, F. Picconi. Pastis: a Highly-Scalable Multi-User Peer-to-Peer File System. In *EuroPar 2005*, September 2005.

[74] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D/ Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[75] L.V.S. Lakshmanan and S. Parthasarathy. On efficient matching of streaming xml documents and queries. In *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*, pages 142–160, London, UK, 2002. Springer-Verlag.

[76] M. Kirtland M. Horstmann. Dcom architecture. http://www.microsoft.com/com/tech/DCOM.asp, July 1997.

[77] Sun Microsystems, Inc. Java 2 Platform Enterprise Edition. http://java.sun.com/j2ee/.

[78] Sun Microsystems, Inc. Java RMI.

[79] Sun Microsystems, INC. *Network Programming.* Sun Microsystems, Mountain View, CA, March 1990.

[80] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 65–76, New York, NY, USA, 2002. ACM Press.

[81] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, 2002.

[82] S. Mullender, editor. *Distributed Systems*, chapter 7 and 8. Addison-Wesley, 2nd edition, 1993.

[83] N. Bruno and L. Gravano and N. Koudas and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *In Proceedings of ICDE 2003*, pages 139–150, Los Alamitos, Calif., March 2003.

[84] Napster. http://www.napster.com/ (2000).

[85] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proceedings of ACM SIGMOD*, pages 437–448, Santa Barbara, California, May 2001.

[86] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, New York, NY, USA, 1993. ACM Press.

[87] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In *Proceedings of Middleware*, April 2000.

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

[89] P. Eugster. *Type-based publish/subscribe.* PhD thesis, EPFL Lausanne, December 2001.

[90] P.E. Chung, Y. Huang, S. Yajnik, D. Liang, J.C. Shih, C.-Y. Wang, Y.M.Wang. DCOM and CORBA side by side, step by step, and layer by layer. *C++ Report*, 10(1), January 1998.

[91] G. Perng, C. Wang, and M.K. Reiter. Providing content based services in a peer to peer environment. In *Proceedings of DEBS*, Edinburgh, UK, May 2004.

[92] G. Picco, G. Cugola, and A. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *Proceedings of ICDCS*, 2003.

[93] P.R. Pietzuch. Event-based middleware: A new paradigm for wide-area distributed systems? In *6th CaberNet Radicals Workshop*, February 2002.

[94] P.R. Pietzuch. *Hermes: A Scalable Event-Based Middleware.* PhD thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.

[95] P.R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[96] P.R. Pietzuch and J.M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 611–618, Vienna, Austria, July 2002.

[97] Planetlab. http://www.planet-lab.org.

[98] 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.

[99] P.Th. Eugster and R. Guerraoui and Ch.H. Damm. On Objects and Events. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 254–269, 2001.

[100] R. Chand and P.A. Felber. Efficient subscription management in content-based networks. In *Proceedings of DEBS*, Edinburgh, UK, May 2004.

[101] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[102] Rebeca Web Site. http://www.gkec.informatik.tu-darmstadt.de/rebeca/.

[103] D.R. Rosenblum, A.L. Wolf, and A. Carzaniga. Critical considerations and designs for Internet-scale, event-based compositional architectures. In *Workshop on Compositional Software Architectures*, January 1998.

[104] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[105] S. Daniel and J. Ellis and T. Truscott. USENET - A General Access UNIX Network. ftp://ftp.std.com/obi/USENET/a/usenet/uprop.n, 1980.

[106] B. Segall and D. Arnold. Elvin Has Left the Building: A Publish/Subscribe Notification Service with Quenching. In *Proceedings of the 1997 Australian UNIX and Open Systems Users Group Conference*, 1997.

[107] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *AUUG2K*, Canberra, Australia, June 2000.

[108] R. Shah, R. Jain, and F. Anjum. Efficient Dissemination of Personalized Information Using Content-Based Multicast. In *Proceedings of INFOCOM*, New-York, June 2002.

[109] SIENA Web Site. http://www.cs.colorado.edu/users/carzanig/siena/.

[110] D. Skeen. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview. http://www.vitria.com, 1998.

[111] A.C. Snoeren, K. Conley, and D.K. Gifford. Mesh Based Content Routing using XML. In *Proceedings of SOSP*, pages 160–173, Alberta, Canada, October 2001.

[112] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM 2002*, pages 73–88, aug 2002.

[113] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, 2001.

[114] W.W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A.P. Buchman. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of DEBS*, San Diego, USA, June 2003.

[115] A. Tozawa and M. Hagiya. XML schema containment checking based on semi-implicit techniques. In *Proceedings of the Conference on Implementation and Application of Automata (CIAA)*, July 2003.

[116] P. Triantafillou and I. Aekaterinidis. Content-based publish-subscribe over structured p2p networks. In *Proceedings of DEBS*, Edinburgh, UK, May 2004.

[117] A. Virgillito. *Publish/Subscribe Communication Systems: From Models to Applications*. PhD thesis, Universita degli studi di Roma La Sapienza, 2003.

[118] W3C. XML Path Language (XPath) 1.0, November 1999.

[119] W3C. Extensible Markup Language (XML) 1.0, 2nd Edition, October 2000.

[120] W3C. Simple Object Access Protocol (SOAP) 1.2, June 2003. http://www.w3.org/TR/soap/.

[121] W3C. XML Query (XQuery) 1.0, April 2005.

[122] S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[123] Y. Diao and S. Rizvi and M.J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proceedings of VLDB 2004*, Toronto, Canada, August 2004.

[124] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for internet-scale event services. In *WETICE '99: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 78–83, Washington, DC, USA, 1999. IEEE Computer Society.

[125] E.W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of INFOCOM*, San Francisco, March 1996.

[126] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.

[127] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.

Appendix A

Extension of the RTU algorithm to the case of multiple producers

Algorithm 27 — Subscription Substitution

create a null $entry(S)$
for all upstream interface $I^g_{up} \neq I^k$ **do**
  if $\exists S', S' \supset S, Ptr^g_{S'} = null$ **then**
    substitute S by S' at I^g_{up}
  else
    for all S_k that can be substituted by S at I^g_{up} **do**
      substitute S_k by S at I^g_{up}
    end for
  end if
end for
call Algorithm 30: "Subscription Representation".

Algorithm 28 — Routing Table Update

for all upstream interface $I^g_{up} \neq I^k$ **do**
  if $Ptr^g_S \neq null$ **then**
    for all S' ancestor of S in $tree^g(h^g(S))$ **do**
      $R^g_{S'} \leftarrow R^g_{S'} + n_S + r_S$
    end for
    $adv^g_{out} \leftarrow (h^g(S); 0; n_S + r_S)$
  else
    $adv^g_{out} \leftarrow (S; n_S; r_S)$
  end if
  $R^g_S \leftarrow R^g_S + r_S$
  Send adv^g_{out} to upstream interface I^g_{up}
end for
$T^k_S.x \leftarrow T^k_S.x + n_S$
$T^k_S.z \leftarrow T^k_S.z + r_S$

Algorithm 29 — Reinsert function: $reinsert^g()$

for all S_j such that $Ptr^g_{S_j} = S$ **do**
  push S_j in $L^g_{reinsert}$
  Reset $Ptr^g_{S_j}$
end for
delete $entry(S)$
for all $S_j \in L^g_{reinsert}$ **do**
  if $\exists S', S' \supset S_j, Ptr^g_{S'} = null$ **then**
    substitute S_j by S' at I^g_{up}
  end if
end for
for all $S_j \in L^g_{reinsert}$ **do**
  if $Ptr^g_{S_j} = null$ **then**
    $count_{S_j} = \sum_k T^k_{S_j}.x$
    append $(S_j; count_{S_j}; R^g_{S_j})$ to adv^g_{out}
  end if
  if $Ptr^g_{S_j} \notin L^g_{reinsert}$ **then**
    $count_{S_j} = \sum_k T^k_{S_j}.x$
    append $(Ptr^g_{S_j}; 0; count_{S_j} + R^g_{S_j})$ to adv^g_{out}
  end if
end for

Algorithm 30 — Subscription Representation

declare $A^g = 0$ for all upstream interface $I^g_{up} \neq I^k$
for all S_j subscriptions that can be represented by S at I^k **do**
  declare $T_j = \overline{T^k_{S_j}}$
  Represent S_j by S at I^k
  for all upstream interface $I^g_{up} \neq I^k$ **do**
    if $S_j \in tree^g(S)$ **then**
      for all S_k ancestor of S_j in $tree^g(S)$ **do**
        $R^g_{S_k} \leftarrow R^g_{S_k} - T_j$
      end for
    else
      for all S_k ancestor of S_j in $tree^g(h^g(S_j))$ **do**
        $R^g_{S_k} \leftarrow R^g_{S_k} - T_j$
      end for
      if $S_j \notin tree^g(h^g(S))$ **then**
        append $(h^g(S_j); 0; -T_j)$ to adv^g_{out}
        $A^g \leftarrow A^g + T_j$
      end if
      for all S_k ancestor of S in $tree^g(h^g(S))$ **do**
        $R^g_{S_k} \leftarrow R^g_{S_k} + T_j$
      end for
    end if
  end for
  if $\sum_{p \leq n} \overline{T^p_{S_j}} = 0$ **then**
    remove $entry(S_j)$
    for all upstream interface $I^g_{up} \neq I^k$ **do**
      for all S_k such that $Ptr^g_{S_k} = S_j$ **do**
        $Ptr^g_{S_k} \leftarrow Ptr^g_{S_j}$
      end for
    end for
  end if
end for
$T^k_S.x \leftarrow T^k_S.x + n_S$
$T^k_S.z \leftarrow T^k_S.z + r_S$
for all upstream interface $I^g_{up} \neq I^k$ **do**
  for all S_k ancestor of S in $tree^g(h^g(S))$ **do**
    $R^g_{S_k} \leftarrow R^g_{S_k} + n_S + r_S$
  end for
  $R^g_S \leftarrow R^g_S + r_S$
  if $h^g(S) \neq null$ **then**
    $adv^g_{out} \leftarrow (h^g(S); 0; n_S + r_S + A^g)$ [+ appended triples]
  else
    $adv^g_{out} \leftarrow (S; n_S; r_S + A^g)$ [+ appended triples]
  end if
  Send adv^g_{out} to upstream interface I^g_{up}
end for

Algorithm 31 — Cancellation algorithm - inner nodes

declare L
declare L^g_{keep}, $L^g_{reinsert}$, $keep^g$ for all upstream interface $I^g_{up} \neq I^k$
for all $S_j \in adv_{in}(S)$ **do**
  push S_j in L
end for
for all $S_j \in L$ **do**
  if S_j does not have an entry **then**
    create a null entry for S_j, $entry(S_j)$
    $Ptr^g_{S_j} \leftarrow S$ for all upstream interface $I^g_{up} \neq I^k$
  end if
  $T^k_{S_j}.x \leftarrow T^k_{S_j}.x + n_{S_j}$
  $T^k_{S_j}.z \leftarrow T^k_{S_j}.z + r_{S_j}$
  for all upstream interface $I^g_{up} \neq I^k$ **do**
    for all S_k ancestor of S_j in $tree^g(h^g(S_j))$ **do**
      $R^g_{S_k} \leftarrow R^g_{S_k} + n_{S_j} + r_{S_j}$
    end for
    if $S_j \in tree^g(h^g(S))$ **then**
      $keep \leftarrow keep + n_{S_j} + r_{S_j}$
      $R^g_{S_j} \leftarrow R^g_{S_j} + r_{S_j}$
    else
      if $Ptr^g_{S_j} \neq null$ **then**
        $adv^g_{out} \leftarrow (h^g(S_j); 0; n_{S_j} + r_{S_j})$
      else
        $adv^g_{out} \leftarrow (S_j; n_{S_j}; r_{S_j})$
      end if
      $R^g_{S_j} \leftarrow R^g_{S_j} + r_{S_j}$
    end if
  end for
end for
for all upstream interface $I^g_{up} \neq I^k$ **do**
  for all S_i ancestor of S in $tree^g(h^g(S))$ **do**
    $R^g_{S_i} \leftarrow R^g_{S_i} + n_S - T^k_S.z$
  end for
  if $Ptr^g_S \neq null$ **then**
    $adv^g_{out} \leftarrow (h(S); 0; n_S - T^k_S.z + keep^g)$
    if $\forall p, \overline{T^p_S} = 0$ **then**
      for all S_i such that $Ptr^g_{S_i} = S$ **do**
        $Ptr^g_{S_i} = Ptr_S$
      end for
    else
      $R^g_S \leftarrow R^g_S - T^k_S.z$
    end if
  else
    if $\forall p, \overline{T^p_S} \neq 0$ **then**
      call $reinsert^g()$ (Algorithm 29)
      $adv^g_{out} \leftarrow (S; n_S; 0)$ [+ appended triples]
    else
      $adv^g_{out} \leftarrow (S; n_S; -T^k_S.z + keep^g)$
      $R^g_S \leftarrow R^g_S - T^k_S.z$
      $T^k_S \leftarrow (0, 0)$
    end if
  end if
  Send adv^g_{out} to upstream interface I^g_{up}
end for

Appendix B

Extension of the *Crash/Recover* algorithm to the case of multiple producers

Algorithm 32 On receiving $Adv(sn)$ from interface i

1: **if** $0 < sn \leq hr_i$ **then** {*Duplicate advertisement*}
2: Send $Ack(sn)$ down interface i
3: **else if** $sn = hr_i + 1$ **then** {*Expected advertisement*}
4: Update routing table with $XRoute$ and generate $Adv^1_{out}(hs) \cdots Adv^p_{out}(hs)$
5: $hr_i \leftarrow hr_i + 1$
6: **for all** Upstream interface I^k_{up} **do**
7: $hs^k \leftarrow hs^k + 1$
8: $RetrBuf^k \overset{append}{\longleftarrow} Adv^k_{out}(hs^k)$
9: **end for**
10: Backup log and routing table in recovery database
11: Send $Ack(sn)$ down interface i
12: **for all** Upstream interface I^k_{up} **do**
13: Send $Adv^k_{out}(hs^k)$ upstream I^k_{up}
14: **end for**
15: **end if**

Algorithm 33 On receiving $Back$ from upstream interface I^k_{up}

1: Send $RetrBuf^k$ upstream interface I^k_{up}

Algorithm 34 On receiving $Ack(sn^k)$ from upstream interface I^k_{up}

1: **if** $Adv^k_{out}(sn^k)$ is found in $RetrBuf^k$ **then**
2: Remove $Adv^k_{out}(sn^k)$ from log
3: Backup log in recovery database
4: **end if**

Algorithm 35 On recovering from failure

1: Recover routing table and log from recovery database
2: **for all** Upstream interface I^k_{up} **do**
3: Send $RetrBuf^k$ upstream interface I^k_{up}
4: **end for**
5: Send $Back$ downstream all interfaces